LUA - DEBUGGING

Lua provides a debug library, which provides all the primitive functions for us to create our own debugger. Even though, there is no in-built Lua debugger, we have many debuggers for Lua, created by various developers with many being open source.

The functions available in the Lua debug library are listed in the following table along with its uses.

S.N.	Method & Purpose
1.	**debug** Enters interactive mode for debugging, which remains active till we type in only cont in a line and press enter. User can inspect variables during this mode using other functions.
2.	**getfenv***object* Returns the environment of object.
3.	**gethook***optionalthread* Returns the current hook settings of the thread, as three values − the current hook function, the current hook mask, and the current hook count.
4.	**getinfo***optionalthread, functionorstacklevel, optionalflag* Returns a table with info about a function. You can give the function directly, or you can give a number as the value of function, which means the function running at level function of the call stack of the given thread − level 0 is the current function *getinfoitself*; level 1 is the function that called getinfo; and so on. If function is a number larger than the number of active functions, then getinfo returns nil.
5.	**getlocal***optionalthread, stacklevel, localindex* Returns the name and the value of the local variable with index local of the function at level of the stack.Returns nil if there is no local variable with the given index, and raises an error when called with a level out of range.
6.	**getmetatable***value* Returns the metatable of the given object or nil if it does not have a metatable.
7.	**getregistry** Returns the registry table,a pre-defined table that can be used by any C code to store whatever Lua value it needs to store.

8.	**getupvalue***function, upvalueindex* This function returns the name and the value of the upvalue with index up of the function func. The function returns nil if there is no upvalue with the given index.
9.	**setfenv***functionorthreadoruserdata, environmenttable* Sets the environment of the given object to the given table. Returns object.
10.	**sethook** ***optionalthread, hookfunction, hookmaskstringwith " c " and/or " r " and/or " l " , optionalinstructioncount*** Sets the given function as a hook. The string mask and the number count describes when the hook will be called. Here, c, r and l are called every time Lua calls, returns, and enters every line of code in a function respectively.
11.	**setlocal***optionalthread, stacklevel, localindex, value* Assigns the value to the local variable with index local of the function at level of the stack. The function returns nil if there is no local variable with the given index, and raises an error when called with a level out of range. Otherwise, it returns the name of the local variable.
12.	**setmetatable***value, metatable* Sets the metatable for the given object to the given table *whichcanbenil*.
13.	**setupvalue***function, upvalueindex, value* This function assigns the value to the upvalue with index up of the function func. The function returns nil if there is no upvalue with the given index. Otherwise, it returns the name of the upvalue.
14.	**traceback***optionalthread, optionalmessagestring, optionallevelargument* Builds an extended error message with a traceback.

The above list is the complete list of debug functions in Lua and we often use a library that uses the above functions and provides easier debugging. Using these functions and creating our own debugger is quite complicated and is not preferable. Anyway, we will see an example of simple use of debugging functions.

```
function myfunction ()
   print(debug.traceback("Stack trace"))
   print(debug.getinfo(1))
   print("Stack trace end")
```

```
   return 10
end

myfunction ()
print(debug.getinfo(1))
```

When we run the above program, we will get the stack trace as shown below.

```
Stack trace
stack traceback:
 test2.lua:2: in function 'myfunction'
 test2.lua:8: in main chunk
 [C]: ?
table: 0054C6C8
Stack trace end
```

In the above sample program, the stack trace is printed by using the debug.trace function available in the debug library. The debug.getinfo gets the current table of the function.

Debugging − Example

We often need to know the local variables of a function for debugging. For that purpose, we can use getupvalue and to set these local variables, we use setupvalue. A simple example for this is shown below.

```
function newCounter ()
   local n = 0
   local k = 0

   return function ()
      k = n
      n = n + 1
      return n
   end

end

counter = newCounter ()

print(counter())
print(counter())

local i = 1

repeat
   name, val = debug.getupvalue(counter, i)

   if name then
      print ("index", i, name, "=", val)

      if(name == "n") then
         debug.setupvalue (counter,2,10)
      end

      i = i + 1
   end -- if

until not name

print(counter())
```

When we run the above program, we will get the following output.

```
1
2
index 1 k = 1
index 2 n = 2
```

```
11
```

In this example, the counter updates by one each time it is called. We can see the current state of the local variable using the getupvalue function. We then set the local variable to a new value. Here, n is 2 before the set operation is called. Using setupvalue function, it is updated to 10. Now when we call the counter function, it will return 11 instead of 3.

Debugging Types

- Command line debugging
- Graphical debugging

Command Line Debugging

Command line debugging is the type of debugging that uses command line to debug with the help of commands and print statements. There are many command line debuggers available for Lua of which a few are listed below.

- **RemDebug** − RemDebug is a remote debugger for Lua 5.0 and 5.1. It lets you control the execution of another Lua program remotely, setting breakpoints and inspecting the current state of the program. RemDebug can also debug CGILua scripts.
- **clidebugger** − A simple command line interface debugger for Lua 5.1 written in pure Lua. It's not dependent on anything other than the standard Lua 5.1 libraries. It was inspired by RemDebug but does not have its remote facilities.
- **ctrace** − A tool for tracing Lua API calls.
- **xdbLua** − A simple Lua command line debugger for the Windows platform.
- **LuaInterface - Debugger** − This project is a debugger extension for LuaInterface. It raises the built in Lua debug interface to a higher level. Interaction with the debugger is done by events and method calls.
- **RIdb** − This is a remote Lua debugger via socket, available on both Windows and Linux. It can give you much more features than any existing one.
- **ModDebug** − This allows in controlling the execution of another Lua program remotely, set breakpoints, and inspect the current state of the program.

Graphical Debugging

Graphical debugging is available with the help of IDE where you are provided with visual debugging of various states like variable values, stack trace and other related information. There is a visual representation and step by step control of execution with the help of breakpoints, step into, step over and other buttons in the IDE.

There are number of graphical debuggers for Lua and it includes the following.

- **SciTE** − The default windows IDE for Lua provides multiple debugging facilities like breakpoints, step, step into, step over, watch variables and so on.
- **Decoda** − This is a graphical debugger with remote debugging support.
- **ZeroBrane Studio** − Lua IDE with integrated remote debugger, stack view, watch view, remote console, static analyzer, and more. Works with LuaJIT, Love2d, Moai, and other Lua engines; Windows, OSX, and Linux. Open source.
- **akdebugger** − Debugger and editor Lua plugin for Eclipse.
- **luaedit** − This features remote debugging, local debugging, syntax highlighting, completion proposal list, parameter proposition engine, advance breakpoint management *includingconditionsystemonbreakpointsandhitcount*, function listing, global and local variables listing, watches, solution oriented management.