

How Lmod Loads a Modulefile, Part 2

Robert McLay

August. 2, 2022

Outline



- ▶ Part 1 took us to `Master:load()` and `loadModuleFile(t)`
- ▶ This part will start with `Master:load()` and `loadModuleFile(t)`
- ▶ This talk will be about handing off your modulefile to the Lua interpreter and Lmod routines.

Outline (II)

- ▶ `main()`: How `main()` parses the command line
- ▶ ...
- ▶ `Master:load(mA)`: Where the heavy lifting is done
- ▶ `loadModuleFile()`: Where the modulefile is actually evaluated

Master: load(): getting ready to call loadModuleFile(t)

- ▶ src: src/Master.lua
- ▶ There is no duplicate sn so we can load the modulefile *foo*.

```
elseif (fn) then
  frameStk:push(mname)
  mt = frameStk:mt()
  mt:add(mname,"pending")
  local status = loadModuleFile{file = fn, shell = shellNm,
                                mList = mList, reportErr = true}
  -- part 2
```

After loading modulefile

```
-- Now fn is loaded

mt = frameStk:mt() -- Why?

-- A modulefile could load the same named module over top of the current modulefile
-- Say modulefile abc/2.0 loads abc/.cpu/2.0. Then in the case of abc/2.0
-- the filename won't match.
if (mt:fn(sn) == fn and status) then
    mt:setStatus(sn, "active")
    hook.apply("load",{fn = mname:fn(), modFullName =
        mname:fullName(), mname = mname})
end
frameStk:pop()
loaded = true
```

Notes about loadModuleFile(t) function

- ▶ Almost all module commands will eval one or more modules
- ▶ This means that loadModuleFile(t) will be called.
- ▶ Does not: module list, module -t avail
- ▶ Does: module load, module unload, module help, module whatis, ...
- ▶ Even if your site uses a system spider cache, you will re-eval modulefiles that change \$MODULEPATH (New)

Let's load the modulefile w/ `loadModuleFile(t)`

```
function loadModuleFile(t)
-----
-- Don't complain if t.file has disappeared when mode() == "unload"
-----
-- Check for infinite loop on mode() == "load"
-----
-- Read modulefile into string "whole"

if (t.ext == ".lua") then
    -- Read complete Lua modulefile into "whole"
else
    -- Convert TCL modulefile into Lua
end

-----

-- Use sandbox to evaluate modulefile

-----

-- Report any errors and error out

-----

-- Mark lmodBrk is LmodBreak() is called inside moduleFile
return not lmodBrk
end
```

- Note that we passed an anonymous table in Master.lua.
- The result is that there is a single argument "t" to the function.

Reading entire modulefile

```
if (t.ext == ".lua") then
  -- Read in lua module file into a [[whole]] string.
  local f = io.open(t.file)
  if (f) then
    whole = f:read("*all")
    f:close()
  end
end
```


Converting TCL to Lua

- ▶ This will be another talk. But TL; DR
- ▶ The TCL interpreter run on the TCL modulefile
- ▶ But `setenv FOO BAR` is converted to the string `setenv("FOO", "BAR")`
- ▶ `prepend-path PATH /...` \Rightarrow `prepend_path("PATH", "/...")`
- ▶ etc

TCL conversion example

----- Input: -----

```
global env
proc ModulesHelp
puts stderr "Help message..."

set modulepath_root $env(MODULEPATH_ROOT)
set moduleshome      "$modulepath_root/TACC"

module load Linux
module try-add cluster
module load TACC-paths
```

----- Output: -----

```
load("Linux")
try_load("cluster")
load("TACC-paths")
```

What is the sandbox?

- ▶ Lua offers the ability to evaluate a string containing Lua code
- ▶ *WITH* the possibility of a limited choice of functions.
- ▶ In other words, Modulefile evaluation has a limited set of functions.
- ▶ They cannot call internal routines Lmod functions.
- ▶ I did this once testing Lmod and realized the problem.
- ▶ Sites may run their own special function
- ▶ BUT: They must be registered with the sandbox
- ▶ Sites can use the SitePackage.lua or /etc/lmod/lmod_config.lua to register their own functions
- ▶ Note that the sandbox controls what the modulefiles can call
- ▶ Once inside however all functions can be called.

status, msg = sandbox_run(whole)

```
local function l_run5_2(untrusted_code)
  local untrusted_function, message = load(untrusted_code, nil, 't', sandbox_env)
  if not untrusted_function then return nil, message end
  return pcall(untrusted_function)
end
```

```
-----
-- Define two version: Lua 5.1 or 5.2. It is likely that
-- The 5.2 version will be good for many new versions of
-- Lua but time will only tell.
sandbox_run = (_VERSION == "Lua 5.1") and l_run5_1 or l_run5_2
```

What is sandbox_env (src/sandbox.lua?)

```
local sandbox_env = {  
  assert    = assert,  
  loadfile  = loadfile,  
  require   = require,  
  ipairs    = ipairs,  
  next      = next,  
  pairs     = pairs,  
  pcall     = pcall,  
  tonumber  = tonumber,  
  tostring  = tostring,  
  type      = type,  
  load      = load_module,  
  load_any  = load_any,  
  
  --- PATH functions ---  
  prepend_path      = prepend_path,  
  append_path       = append_path,  
  remove_path       = remove_path,  
  
  --- Set Environment functions ----  
  setenv            = setenv,  
  pushenv           = pushenv,  
  unsetenv          = unsetenv,  
  
  ...  
}
```

► Modulefiles can only execute functions in sandbox_env

Modulefile Evaluation: Who is in charge?

- ▶ Once `sandbox_run()` is called then Lua is in charge
- ▶ Not Lmod!
- ▶ All normal Lua statements are run by Lua.
- ▶ Lmod never sees `if () stmts etc.`
- ▶ Lmod code is called when executing `setenv()`, `prepend_path()` etc.

Example: StdEnv.lua

```
local hroot    = pathJoin(os.getenv("HOME") or "", "myModules")
local userDir  = pathJoin(hroot, "Core")
if (isDir(userDir)) then
    prepend_path("MODULEPATH", userDir)
end
haveDynamicMPATH()
```

- ▶ Recent change in Lmod 8.7.4: Dynamic Spider Cache
- ▶ Lmod remember modules that change \$MODULEPATH
- ▶ If ~/myModules doesn't exist then Lmod will never know that this module could change \$MODULEPATH
- ▶ That is why this module needs haveDynamicMPATH().

Suppose your modulefile calls `setenv()`

- ▶ Module command functions are implemented in `src/modfuncs.lua`
- ▶ Below is the `modfunc.lua` function `setenv()`
- ▶ So the arguments are check to be strings and/or numbers with `l_validateArgsWithValue()`
- ▶ Then the start of the work happens with `mcp:setenv(...)`

```
function setenv(...)
  dbg.start{"setenv(",l_concatTbl(...," ","),")"}
  if (not l_validateArgsWithValue("setenv",...)) then return end

  mcp:setenv(...)
  dbg.fini("setenv")
  return
end
```


mcp:setenv() what is it gonna do?

- ▶ `mode() == "load"` \Rightarrow `MasterControl:setenv()`
- ▶ `mode() == "unload"` \Rightarrow `MasterControl:unsetenv()`
- ▶ `mode() == "show"` \Rightarrow `MC_Show:setenv()`
- ▶ `mode() == "refresh"` \Rightarrow `MasterControl:quiet()`
- ▶ Note that there is NOT if block making these choices
- ▶ It depends on which object type `mcp` was build as.

There are ten different types of MasterControl classes

- ▶ MC_Access.lua ⇒ whatis, help
- ▶ MC_CheckSyntax.lua ⇒ for checking the syntax of a modulefile.
- ▶ MC_ComputeHash lua ⇒ Computing the hash value used in user collections.
- ▶ MC_DependencyCk.lua ⇒ reloading modules to report any problems with the dependencies.
- ▶ MC_Load.lua ⇒ loading modules
- ▶ MC_Mgrload ⇒ the loading method when loading a collection (ignore load() type functions inside modulefiles)
- ▶ MC_Refresh ⇒ reload all currently loaded modules but only run set_alias() set_shell_function() and nothing else
- ▶ MC_Show ⇒ Just print out commands
- ▶ MC_Spider ⇒ How to process modules when in spider mode
- ▶ MC_Unload ⇒ How Lmod unloads modules (setenv ⇒ unsetenv, prepend_path() ⇒ remove_path() etc)

MasterControl:setenv()

```
function M.setenv(self, name, value, respect)
  dbg.start{"MasterControl:setenv(",name,"; ",value,"; ",
          respect,"")
  l_check_for_valid_name("setenv",name)

  if (value == nil) then
    LmodErrormsg="e_Missing_Value", func = "setenv", name = name
  end

  if (respect and getenv(name)) then
    dbg.print"Respecting old value"

    dbg.fini("MasterControl:setenv")
    return
  end

  local frameStk = FrameStk:singleton()
  local varT      = frameStk:varT()
  if (varT[name] == nil) then
    varT[name] = Var:new(name)
  end
  varT[name]:set(tostring(value))
  dbg.fini("MasterControl:setenv")
end
```

Lmod changing your Environment in lmod.in.lua

```
-- Output all newly created path and variables.  
Shell:expand(varT)  
  
-- Expand any execute{} cmds  
if (Shell:real_shell())then  
  Exec:exec():expand()  
end
```

- ▶ Assuming no errors
- ▶ Then Lmod prints out the changes to your env.
- ▶ With the above code.
- ▶ All changes are in varT

module unload foo

```
function M.unsetenv(self, name, value, respect)
  name = name:trim()
  dbg.start{"MasterControl:unsetenv(",name,"; ",value,")"}

  l_check_for_valid_name("unsetenv",name)

  if (respect and getenv(name) ~= value) then
    dbg.print"Respecting old value"
    dbg.fini("MasterControl:unsetenv")
    return
  end

  local frameStk = FrameStk:singleton()
  local varT      = frameStk:varT()
  if (varT[name] == nil) then
    varT[name] = Var:new(name)
  end
  varT[name]:unset()

  -- Unset stack variable if it exists
  local stackName = l_createStackName(name)
  if (varT[stackName]) then
    varT[name]:unset()
  end
  dbg.fini("MasterControl:unsetenv")
end
```

► Here we unsetenv() by calling varT[name]:unset()

module show foo

```
function M.setenv(self, name,value)
    l_ShowCmd("setenv", name, value)
end
```

- In src/MC_Show.lua the setenv command is printed.

Next time

- ▶ How the translation from TCL to Lua is handled
- ▶ Why it is not perfect.

Conclusions

- ▶ Lmod provide a way to evaluate your modulefile.
- ▶ It does so through the `sandbox_run()` function
- ▶ And hands off the evaluation to Lua.

Future Topics

- ▶ Next Meeting: September 6th 9:30 US Central (14:30 UTC)
- ▶ How the translation from TCL to Lua is handled