

luerl(3)

Jean Chassoul, Robert Virding

2018-2024

Name

luerl - The basic interface to the Luerl system

Interface functions - New Version

The **Lua State** parameter is the state of a Lua VM instance. It must be created with the **luerl:init()** call and be carried from one call to the next.

As it is possible in Lua to create self-referencing data structures, indeed the standard libraries have many instances of this, then using the functions which decode their return values will generate an error when they would cause an infinite loop during the decoding. An simple example is the top level table which contains a key **_G** which references the top-level table.

Note that Lua **Chunks** (see definition below) can travel between different States. They are precompiled bits of code, independent of State. That you can ‘carry around’ this is no unique to Luerl but a low-level implementation detail of the standard Lua language, for more on chunks read the official Lua 5.3 reference manual.

Spec Definitions

Binary means an Erlang binary string.

Chunks means a portion of precompiled bytecode.

State means a Lua State, this *is* a Lua VM instance.

Path means a file system path and file name.

KeyPath means an Erlang list of **atoms** representing nested names, e.g. [table,pack] for table.pack.

Keys means Lua table keys, the keys of a key-value structure.

CompileOptions means a list of compiler options. Currently supported options are ‘return’, which returns the errors and warnings, and ‘report’ which will log the errors and warnings.

LuaCallReturn = {ok, Result, State} | {lua__error, Error, State}

This is the return value from evaluating a Lua call.

Functions

luerl:init() -> State

Get a new Lua State = a fresh Lua VM instance.

luerl:gc(State) -> State

Runs the garbage collector on a state and returns the new state.

luerl:load(String|Binary[, CompileOptions], State) -> {ok, Function, State} | CompileError

Parse a Lua chunk as string or binary, and return a compiled chunk ('form').

luerl:loadfile(FileName[, CompileOptions], State) -> {ok, Function, State} | CompileError

Parse a Lua file, and return a compiled chunk ('form').

**luerl:path_loadfile([Path,], FileName[, CompileOptions], State)
-> {ok,Function,FullName,State} | {error, Reason}**

Search Path until the file FileName is found. Parse the file and return a compiled chunk ('form'). If Path is not given then the path defined in the environment variable `LUA_LOAD_PATH` is used.

luerl:load_module(KeyPath, ErlangModule, State) -> State

Load ErlangModule and install its table at KeyPath which is **NOT** encoded.

luerl:load_module_dec(EncodedKeyPath, ErlangModule, State) -> State

Load ErlangModule and install its table at KeyPath which is encoded.

luerl:do(String|Binary|Form, State) -> {ok, Result, NewState} | {lua__error, Error, State} | CompileError

Evaluate a Lua expression and return its result which is **NOT** decoded, and the new Lua State.

```
luerl:do_dec(String|Binary|Form, State) -> {ok, Result, NewState}
| {lua_error, Error, State} | CompileError
```

Evaluate a Lua expression and return its result which is automatically decoded, and the new Lua State.

```
luerl:dofile(Path, State) -> {ok, Result, NewState} | {lua_error,
Error, State} | CompileError
```

Load and execute the Lua code in the file and return its result which is **NOT** decoded, and the new Lua State. Equivalent to doing `luerl:do("return dofile('FileName')")`.

```
luerl:dofile_dec(Path[, State]) -> {ok, Result, NewState} |
{lua_error, Error, State} | CompileError
```

Load and execute the Lua code in the file and return its result which is automatically decoded, and the new Lua State.

```
luerl:call(FuncRef, ArgRefs, State) -> {ok, Result, State}
```

```
luerl:call_chunk(FuncRef, ArgRefs, State) -> {ok, Result, State}
| {lua_error, Error, State}
```

Call a compiled chunk or function. Use the `call_chunk`, call has been kept for backwards compatibility.

```
luerl:call_function(FuncRef | FuncPath, ArgRefs, State) -> {ok,
Result, State} | {lua_error, Error, State}
```

Call a function already defined in the state. **Result** is **NOT** decoded.

```
luerl:call_function_enc(KeyPath, Args, State) -> {ok, Result,
State} | {lua_error, Error, State}
```

Call a function already defined in the state. **KeyPath** is a list of keys to the function. **KeyPath** and **Args** are automatically encoded, while **Result** is **NOT** decoded.

```
luerl:call_function_dec(KeyPath, Args, State) -> {ok, Result,
State} | {lua_error, Error, State}
```

Call a function already defined in the state. **KeyPath** is a list of keys to the function. **KeyPath** and **Args** are automatically encoded and **Result** is automatically decoded.

```
luerl:call_method(ObjRef, Method, ArgRefs, State) -> {ok, Result, State} | {lua_error, Error, State}
```

Call a method already defined in the state. Result is **NOT** decoded.

```
luerl:call_method_enc(KeyPath, Method, Args, State) -> {ok, Result, State} | {lua_error, Error, State}
```

Call a method already defined in the state. KeyPath is a list of keys to the method. KeyPath, Method and Args are automatically encoded, while Result is **NOT** decoded.

```
luerl:call_method_dec(KeyPath, Method, Args, State) -> {ok, Result, State} | {lua_error, Error, State}
```

Call a method already defined in the state. KeyPath is a list of keys to the method. KeyPath, Method and Args are automatically encoded and Result is automatically decoded.

```
luerl:get_table_keys(KeyPath, State) -> {ok, Result, State} | {lua_error, Error, State}
```

Gets a value inside the Lua state. KeyPath and Result are **NOT** encoded/decoded.

```
luerl:get_table_keys_dec(KeyPath, State) -> {ok, Result, State} | {lua_error, Error, State}
```

Gets a value inside the Lua state. KeyPath is automatically encoded and Result is decoded.

```
luerl:set_table_keys(KeyPath, Value, State) -> {ok, State} | {lua_error, Error, State}
```

Sets a value inside the Lua state. KeyPath and Value are **NOT** encoded.

```
luerl:set_table_keys_dec(KeyPath, Value, State) -> {ok, Result, State} | {lua_error, Error, State}
```

Sets a value inside the Lua state. KeyPath and Value are automatically encoded and Result is decoded.

```
luerl:get_table_key(Table, Key, State) -> {ok, Result, State} | {lua_error, Error, State}
```

Gets the value of a key in a table. Table and Key are **NOT** encoded and Result is **NOT** decoded.

```
luerl:set_table_key(Table, Key, Value, State) -> {ok, State} |  
{lua_error, Error, State}
```

Sets the value of a key in a table. Table, Key and Value are **NOT** encoded.

```
luerl:get_stacktrace(State) -> [{FuncName,{file,FileName},{line,Line}}]
```

Return a stack trace of the current call stack in the state.

```
luerl:encode(Term, State) -> {LuerlTerm,State}
```

Encode the Erlang representation of a term into Luerl form updating the state when necessary.

```
luerl:encode_list([Term], State) -> {[LuerlTerm],State}
```

Encode a list of Erlang term representations into a list of Luerl forms updating the state when necessary.

```
luerl:decode(LuerlTerm, State) -> Term
```

Decode a term in the Luerl form into its Erlang representation.

```
luerl:decode_list([LuerlTerm], State) -> [Term]
```

Decode a list of Luerl terms into a list of Erlang representations.

```
luerl:put_private(Key, Term, State) -> State.
```

Puts a private value under key that is not exposed to the runtime.

```
luerl:get_private(Key, State) -> Term.
```

Get a private value for the given key.

```
luerl:delete_private(Key, State) -> Term.
```

Deletes the private value for the given key.

Passing String in the Erlang Shell

Here we are going to look at passing in command strings into the `luerl:do/2` function, especially when these strings contain Lua strings. The problem is to make sure that the strings in the Lua commands are processed correctly.

First just doing it from the standard Lua shell as the source of truth:

```
>return "aéb\235c"  
aéb?c  
>string.byte("aéb\235c", 1, 20)
```

97 195 169 98 235 99

Now doing it from Erlang where I need to use `\\` to get a `\` into the string:

```
1> St = luerl:init(), ok.  
ok  
2> f(S), S = "return 'aéb\\235c'".  
[114,101,116,117,114,110,32,39,97,233,98,92,50,51,53,99,39]  
3> f(R), {ok,R,_} = luerl:do(S, St), io:write(R).  
[<<97,195,169,98,235,99>>]
```

Now doing it from Elixir using the `~C` sigil to get the handling of the string right:

```
iex(1)> st = Luerl.init(); :ok  
:ok  
iex(4)> s = ~C"return 'aéb\235c'"  
[114,101,116,117,114,110,32,39,97,233,98,92,50,51,53,99,39]  
iex(3)> {ok,r,_} = Luerl.do(st, s); r  
[<<97, 195, 169, 98, 235, 99>>]
```

So it works as expected. We use `io:write` to write out the characters in the binary without any parsing or trying to be smart.