luerl_old(3)

Jean Chassoul, Robert Virding

2018-2024

Name

luerl - The old original interface to the Luerl system

Interface functions

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.

Functions

eval and **do** functions differ only in what they return. The **do** functions return results and a new Lua State, the **eval** functions return a tuple starting on 'ok' or 'error', then the result, or cause of error.

```
do --> {Result, State}

eval --> {ok, Result} | {error, Reason}
```

luerl:eval(String|Binary|Form, State) -> {ok, Result} | {error, Reason, StackTrace}.

Evaluate a Lua expression passed in as a string or binary, and return its result.

luerl:evalfile(Path, State) -> {ok, Result} | {error, Reason, StackTrace}.

Load and execute a file, and return the result.

luerl:do(String|Binary|Form, State) -> {Result, NewState}.

Evaluate a Lua expression and return its result, and the new Lua State.

luerl:dofile(Path, State) -> {Result, NewState}.

Load and execute the Lua code in the file and return its result, and the new Lua State. Equivalent to doing luerl:do("return dofile('FileName')").

luerl:load(String|Binary[, CompileOptions], State) -> {ok,Function,NewState} | {error, Reason}.

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

luerl:loadfile(FileName[, CompileOptions], State) -> {ok,Function,NewState} | {error, Reason}.

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 encoded.

luerl:load_module1(KeyPath, ErlangModule, State) -> State.

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

luerl:init() -> State.

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

luerl:call(Form, Args, State) -> {Result,State}

luerl:call_chunk(Form, Args, State) -> {Result,State}

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

luerl:call_function(KeyPath, Args, State) -> {Result,NewState}

Call a function already defined in the state. `KeyPath` is a list of names to the function. `KeyPath`, `Args` and `Result` are automatically encoded/decoded.

luerl:call_function1(KeyPath, Args, State) -> {Result,NewState}

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

luerl:call_method(MethPath, Args, State) -> {Result,NewState}.

Call a method already defined in the state. `MethPath` is a list of names to the method. `MethPath`, `Args` and `Result` are automatically encoded/decoded.

luerl:call_method1(MethPath, Args, State) -> {Result,NewState}

Call a method already defined in the state. `MethPath` is a list of keys to the method. `Keys`, `Args` and `Result` are **NOT** encoded/decoded.

luerl:stop(State) -> GCedState.

Garbage collects the state and (todo:) does away with it.

luerl:gc(State) -> State.

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

luerl:set_table(KeyPath, Value, State) -> State.

Sets a value inside the Lua state. Value is automatically encoded.

luerl:set_table1(KeyPath, Value, State) -> State.

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

luerl:get_table(KeyPath, State) -> {Result,State}.

Gets a value inside the Lua state. `KeyPath` and `Result` are automatically encoded.

luerl:get_table1(KeyPath, State) -> {Result,State}.

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

You can use this function to expose an function to the Lua code by using this interface:
`fun(Args, State) -> {Results, State}`

Args and Results must be a list of Luerl compatible Erlang values.