Package: luajr (via r-universe)

July 2, 2024

Type Package

Title 'LuaJIT' Scripting

Version 0.1.8

Description An interface to 'LuaJIT' <https://luajit.org>, a just-in-time compiler for the 'Lua' scripting language <https://www.lua.org>. Allows users to run 'Lua' code from 'R'.

URL https://github.com/nicholasdavies/luajr, https://nicholasdavies.github.io/luajr/

BugReports https://github.com/nicholasdavies/luajr/issues

License MIT + file LICENSE

Encoding UTF-8

SystemRequirements GNU make

Suggests Rcpp, crayon, knitr, rmarkdown, testthat (>= 3.0.0)

RoxygenNote 7.3.1

VignetteBuilder knitr

Config/testthat/edition 3

NeedsCompilation yes

Author Mike Pall [aut, cph] (Author of the embedded LuaJIT compiler), Lua.org, PUC-Rio [cph] (Copyright holders over portions of Lua source code included in LuaJIT), Nicholas Davies [cre, ctb, cph] (Author of the R package wrapper, <https://orcid.org/0000-0002-1740-1412>)

Maintainer Nicholas Davies <nicholas.davies@lshtm.ac.uk>

Repository CRAN

Date/Publication 2024-07-01 14:50:02 UTC

Contents

`luajr-package` *luajr: LuaJIT Scripting*

Description

'luajr' provides an interface to LuaJIT, a just-in-time compiler for the Lua scripting language. It allows users to run Lua code from R.

The R API

- `lua()`: run Lua code
- `lua_func()`: make a Lua function callable from R
- `lua_shell()`: run an interactive Lua shell
- `lua_open()`: create a new Lua state
- `lua_reset()`: reset the default Lua state
- `lua_parallel()`: run Lua code in parallel

Further reading

For an introduction to 'luajr', see `vignette("luajr")`

Author(s)

Maintainer: Nicholas Davies <nicholas.davies@lshtm.ac.uk> (ORCID) (Author of the R package wrapper) [contributor, copyright holder]

Authors:

- Mike Pall (Author of the embedded LuaJIT compiler) [copyright holder]

Other contributors:

- Lua.org, PUC-Rio (Copyright holders over portions of Lua source code included in LuaJIT) [copyright holder]

See Also

Useful links:

- https://github.com/nicholasdavies/luajr
- https://nicholasdavies.github.io/luajr/
- Report bugs at https://github.com/nicholasdavies/luajr/issues

---

`lua` *Run Lua code*

Description

Runs the specified Lua code.

Usage

```
lua(code, filename = NULL, L = NULL)
```

Arguments

`code`	Lua code block to run.
`filename`	If non-`NULL`, name of file to run.
`L`	Lua state in which to run the code. `NULL` (default) uses the default Lua state for **luajr**.

Value

Lua value(s) returned by the code block converted to R object(s). Only a subset of all Lua types can be converted to R objects at present. If multiple values are returned, these are packaged in a `list`.

Examples

```
twelve <- lua("return 3*4")
print(twelve)
```

lua_func *Make a Lua function callable from R*

Description

Takes any Lua expression that evaluates to a function and provides an R function that can be called to invoke the Lua function.

Usage

```
lua_func(func, argcode = "s", L = NULL)
```

Arguments

func	Lua expression evaluating to a function.
argcode	How to wrap R arguments for the Lua function.
L	Lua state in which to run the code. NULL (default) uses the default Lua state for **luajr**.

Details

The R types that can be passed to Lua are: NULL, logical vector, integer vector, numeric vector, string vector, list, external pointer, and raw.

The parameter argcode is a string with one character for each argument of the Lua function, recycled as needed (e.g. so that a single character would apply to all arguments regardless of how many there are).

In the following, the corresponding character of argcode for a specific argument is referred to as its argcode.

For NULL or any argument with length 0, the result in Lua is **nil** regardless of the corresponding argcode.

For logical, integer, double, and character vectors, if the corresponding argcode is 's' (simplify), then if the R vector has length one, it is supplied as a Lua primitive (boolean, number, number, or string, respectively), and if length > 1, as an array, i.e. a table with integer indices starting at 1. If the code is 'a', the vector is always supplied as an array, even if it only has length 1. If the argcode is the digit '1' through '9', this is the same as 's', but the vector is required to have that specific length, otherwise an error message is emitted.

Still focusing on the same vector types, if the argcode is 'r', then the vector is passed *by reference* to Lua, adopting the type luajr.logical_r, luajr.integer_r, luajr.numeric_r, or luajr.character_r as appropriate. If the argcode is 'v', the vector is passed *by value* to Lua, adopting the type luajr.logical, luajr.integer, luajr.numeric, or luajr.character as appropriate.

For a raw vector, only the 's' type is accepted and the result in Lua is a string (potentially with embedded nulls).

For lists, if the argcode is 's' (simplify), the list is passed as a Lua table. Any entries of the list with non-blank names are named in the table, while unnamed entries have the associated integer key in

the table. Note that Lua does not preserve the order of entries in tables. This means that an R list with names will often go "out of order" when passed into Lua with `'s'` and then returned back to R. This is avoided with argcode `'r'` or `'v'`.

If a list is passed in with the argcode `'r'` or `'v'`, the list is passed to Lua as type `luajr.list`, and all vector elements of the list are passed by reference or by value, respectively.

For external pointers, the argcode is ignored and the external pointer is passed to Lua as type **userdata**.

When the function is called and Lua values are returned from the function, the Lua return values are converted to R values as follows.

If nothing is returned, the function returns `invisible()` (i.e. `NULL`).

If multiple arguments are returned, a list with all arguments is returned.

Reference types (e.g. `luajr.logical_r`) and vector types (e.g. `luajr.logical`) are returned to R as such. A `luajr.list` is returned as an R list. Reference and list types respect R attributes set within Lua code.

A **table** is returned as a list. In the list, any table entries with a number key come first (with indices 1 to n, i.e. the original number key's value is discarded), followed by any table entries with a string key (named accordingly). This may well scramble the order of keys, so beware. Note in particular that Lua does not guarantee that it will traverse a table in ascending order of keys. Entries with non-number, non-string keys are discarded. It is probably best to avoid returning a **table** with anything other than string keys, or to use `luajr.list`.

A Lua string with embedded nulls is returned as an R raw type.

Value

An R function which can be called to invoke the Lua function.

Examples

```
squared <- lua_func("function(x) return x^2 end")
print(squared(7))
```

---

lua_open *Create a new Lua state*

Description

Creates a new, empty Lua state and returns an external pointer wrapping that state.

Usage

```
lua_open()
```

Details

All Lua code is executed within a given Lua state. A Lua state is similar to the global environment in R, in that it is where all variables and functions are defined. **luajr** automatically maintains a "default" Lua state, so most users of **luajr** will not need to use `lua_open()`.

However, if for whatever reason you want to maintain multiple different Lua states at a time, each with their own independent global variables and functions, `lua_open()` can be used to create a new Lua state which can then be passed to `lua()`, `lua_func()` and `lua_shell()` via the `L` parameter. These functions will then operate within that Lua state instead of the default one. The default Lua state can be specified explicitly with `L = NULL`.

Note that there is currently no way (provided by **luajr**) of saving a Lua state to disk so that the state can be restarted later. Also, there is no `lua_close` in **luajr** because Lua states are closed automatically when they are garbage collected in R.

Value

External pointer wrapping the newly created Lua state.

Examples

```
L1 <- lua_open()
lua("a = 2")
lua("a = 4", L = L1)
lua("print(a)") # 2
lua("print(a)", L = L1) # 4
```

---

lua_parallel *Run Lua code in parallel*

Description

Runs a Lua function multiple times, with function runs divided among multiple threads.

Usage

```
lua_parallel(func, n, threads, pre = NA_character_)
```

Arguments

`func`	Lua expression evaluating to a function.
`n`	Number of function executions.
`threads`	Number of threads to create, or a list of existing Lua states (e.g. as created by `lua_open()`), all different, one for each thread.
`pre`	Lua code block to run once for each thread at creation.

Details

This function is experimental. Its interface and behaviour are likely to change in subsequent versions of luajr.

`lua_parallel()` works as follows. A number `threads` of new Lua states is created with the standard Lua libraries and the `luajr` module opened in each (i.e. as though the states were created using `lua_open()`). Then, a thread is launched for each state. Within each thread, the code in `pre` is run in the corresponding Lua state. Then, `func(i)` is called for each `i` in `1:n`, with the calls spread across the states. Finally, the Lua states are closed and the results are returned in a list.

Instead of an integer, `threads` can be a list of Lua states, e.g. `NULL` for the default Lua state or a state returned by `lua_open()`. This saves the time needed to open the new states, which takes a few milliseconds.

Value

List of `n` values returned from the Lua function `func`.

Safety and performance

Note that `func` has to be thread-safe. All pure Lua code and built-in Lua library functions are thread-safe, except for certain functions in the built-in **os** and **io** libraries (search for "thread safe" in the Lua 5.2 reference manual).

Additionally, use of luajr reference types is **not** thread-safe because these use R to allocate and manage memory, and R is not thread-safe. This means that you cannot safely use `luajr.logical_r`, `luajr.integer_r`, `luajr.numeric_r`, `luajr.character_r`, or other reference types within `func`. `luajr.list` and `luajr.dataframe` are fine, provided the list entries / dataframe columns are value types.

There is overhead associated with creating new Lua states and with gathering all the function results in an R list. It is advisable to check whether running your Lua code in parallel actually gives a substantial speed increase.

Examples

```
lua_parallel("function(i) return i end", n = 4, threads = 2)
```

`lua_reset` *Reset the default Lua state*

Description

Clears out all variables from the default Lua state, freeing up the associated memory.

Usage

```
lua_reset()
```

Details

This resets the default Lua state only. To reset a non-default Lua state `L` returned by `lua_open()`, just do `L <- lua_open()` again. The memory previously used will be cleaned up at the next garbage collection.

Value

None.

Examples

```
lua("a = 2")
lua_reset()
lua("print(a)") # nil
```

`lua_shell` *Run an interactive Lua shell*

Description

When in interactive mode, provides a basic read-eval-print loop with LuaJIT.

Usage

```
lua_shell(L = NULL)
```

Arguments

`L`	Lua state in which to run the code. `NULL` (default) uses the default Lua state for **luajr**.

Details

Enter an empty line to return to R.

As a convenience, lines starting with an equals sign have the `"="` replaced with `"return "`, so that e.g. entering `=x` will show the value of `x` as returned to R.

Value

None.

Index