redis-lua Documentation

Release 2.0.8

Julien Kauffmann

October 12, 2016

Contents

1 Quick start **3**
1.1 Step-by-step analysis . . . 3

2 What's the magic at play here ? **5**

3 One step further **7**

4 What happens when I make a mistake ? **9**

5 What's next ? **11**

6 Table of contents **13**
6.1 Basic usage . . . 13
6.2 Advanced usage . . . 14
6.3 API . . . 16

7 Indices and tables **19**

redis-lua is a pure-Python library that eases usage of LUA scripts with Redis. It provides script loading and parsing abilities as well as testing primitives.

CHAPTER 1

Quick start

A code sample is worth a thousand words:

```
from redis_lua import load_script

# Loads the 'create_foo.lua' in the 'lua' directory.
script = load_script(name='create_foo', path='lua/')

# Run the script with the specified arguments.
foo = script.get_runner(client=redis_client)(
    members={'john', 'susan', 'bob'},
    size=5,
)
```

1.1 Step-by-step analysis

Let's go through the code sample step by step.

First we have:

```
from redis_lua import load_script
```

We import the only function we need. Nothing too specific here.

The next lines are:

```
# Loads the 'create_foo.lua' in the 'lua' directory.
script = load_script(name='create_foo', path='lua/')
```

These lines looks for a file named *create_foo.lua* in the *lua* directory, relative to the current working directory. This example actually considers that using the current directory is correct. In a production code, you likely want to make sure to use a more reliable or absolute path.

The `load_script` function takes the name of the script to load, without its *.lua* extension and a path to search from. It supports sub-directories which means that specifying *subdir/foo* in the path *lua/* will actually look for the file *lua/subdir/foo.lua*. It returns a `Script` instance.

In production code, you will likely want to load several scripts at once. To this intent, you can either use `load_scripts` which takes a list of names or `load_all_scripts` which loads all the scripts it can find in a given directory. The former gives you better, explicit control over which files you load and the latter in nice in case where you want to load every script.

Finally we have:

```
# Run the script with the specified arguments.
foo = script.get_runner(client=redis_client)(
    members={'john', 'susan', 'bob'},
    size=5,
)
```

This basically tells the specified Redis instance to execute the script with the specified parameters and to give us back the result. Note how *redis_lua* translated all the arguments and the return value for you transparently. We will see in a moment what it takes to reach that level of user-friendlyness.

The `Script.get_runner` method takes either a Redis connection or a pipeline and returns a callable. This callable expects in turn the named arguments to call the script with and returns the script's return value.

CHAPTER 2

What's the magic at play here ?

You may wonder how it is possible for *redis_lua* to possibly know how to translate the *members* and *size* arguments to something meaningful in LUA.

Let's take a look at the *create_foo.lua* file:

```
%arg size integer
%arg members list
%return dict

local foo_id = redis.call('INCR', 'foo:last_id')
local foo_root_key = string.format('foo:%s', foo_id)
local foo_members_key = foo_root_key .. ':members'
local foo_size_key = foo_root_key .. ':size'

redis.call('SET', foo_size_key, size)
redis.call('SADD', foo_members_key, unpack(members))

return cjson.encode({
    id=foo_id,
    members=members,
    size=size,
})
```

Notice the *%arg* and *%return* lines ? This is where the magic happens.

redis_lua extends the language of LUA scripts with new instructions to instrument your scripts. A *%arg* instruction declares a named variable and its type so that it gets converted for you automatically when calling the script. A *%return* statement declares the expected return type of the LUA script so that you don't have to parse it yourself.

You will never have to write things like this anymore:

```
local size = tonumber(ARGV[1]) -- So size goes first. I'll have to remember
                               -- that when calling the script.
local members = cjson.decode(ARGV[2]) -- Don't forget to increment ARGV !
```

CHAPTER 3

One step further

Aren't you tired of duplicating logic in your LUA scripts just because the *require* instruction isn't available in Redis LUA ?

Well, you won't have to do that anymore: *redis_lua* not only supports named arguments but also recursive *%include* directives ! Split your scripts in as many files as you want and simply assemble them using the appropriate *%include* statements ! *redis_lua* will take care of concatening the different scripts for you automatically.

CHAPTER 4

What happens when I make a mistake ?

At this point, you might be afraid that extending the language with new instructions and include capabilities causes you troubles when something wrong happens in your LUA script. Fear no longer: *redis_lua* not only extends the langage but also improves the debugging and testing tools.

Let's introduce a problem in our script and look at the resulting exception:

```
redis_lua.exceptions.ScriptError: Script attempted to access unexisting global variable 'foo_size_key
LUA Traceback (most recent script last):
  Script "<user-code>", line 8
    local foo_size_key = foo_size_key .. ':size'
```

The LUA scripting error was detected as usual by *redispy* but *redis_lua* was able to enhance it with more contextual information: the script in which the error occured, the line at which it occured and the offending code. Had the error happened in a sub-script (via an *%include* directive), the traceback of the different scripts would have been shown as well. Simple, reliable and efficient.

CHAPTER 5

What's next ?

Check out the *API* for more details about the available functions.

CHAPTER 6

Table of contents

6.1 Basic usage

6.1.1 Files layout

It is recommended to create dedicated folder in your project for your LUA scripts.

A good convention is to name it *lua* as it is both obvious and short to type.

You can then organize your scripts in whichever structure you want: with or without subfolders, one for each function or several functions per file. It's really up to you. Just make sure you stay consistent.

A common pattern to get a variable that points to that root folder for scripts is to write:

```
LUA_SCRIPTS_PATH = os.path.join(
    os.path.dirname(os.path.abspath(__file__)),
    'lua',
)
```

6.1.2 Initialization

In a typical case, you want to load all your scripts in the initialization of your executable or service. The easiest way to achieve that is to simply call:

```
from redis_lua import load_all_scripts

scripts = load_all_scripts(path=LUA_SEARCH_PATH)
```

6.1.3 Calling scripts

Calling a script is easy:

```
result = scripts['foo'].get_runner(client=client)(
    my_key='my_key',
    my_arg='my_arg',
)
```

`result` will contain the result as given by the LUA script.

6.2 Advanced usage

redis_lua does not only provide helpers to deal with LUA scripts but also extends the LUA syntax to add new features.

6.2.1 Keys, arguments and return values

redis_lua helps you calling LUA scripts by allowing you to name your keys and arguments.

To do that, you can use the *%key* and *%arg* instructions. Both are pretty similar but don't have the exact same syntax.

Keys

The *%key* instruction takes one mandatory parameter that is the name of the key. Keys and arguments share the same namespace which means you can only have one key or argument with a given name. To avoid ambiguity on the Python side, it is recommended that you suffix your keys with *_key* as to make it obvious you are dealing with a Redis key.

Here is an example of usage of the *%key* instruction:

```
%key player_key
```

This exposes a *player_key* function argument on the Python side which expects to be set with a Redis key (a string).

Arguments

The *%arg* instructions takes one mandatory parameter which is the name of the argument, like *%key*, and one optional parameter, which is the type of the argument.

Here is an example of usage of the *%arg* instruction:

```
%arg count int
```

This exposes a *count* function argument on the Python side which expects to be set with a Python integer value.

Here is a list of the supported types:

Aliases	Python type	LUA type
int	int	number
integer	int	number
string	str	string
str	str	string
bool	bool	number
boolean	bool	number
dict	dict	array (dict)
dictionary	dict	array (dict)
list	list	array
array	list	array

If no type is specified, the argument is transfered as-is to the script using the default argument conversion of *pyredis*. It is unspecified what this conversion does exactly.

Return values

The *%return* statement indicates the expected return type of the script when converting the value for return on the Python side. The user is responsible for providing a value that can correctly be cast to the registered return type.

Here is an example of usage of the *%return* instruction:

```
%return dict
```

This cause the value returned by the script to be interpreted as a JSON-encoded dictionary and converted implicitely into a Python *dict*.

Here is a list of the expected LUA types for each type:

Aliases	Python type	LUA type
int	int	number
integer	int	number
string	str	string
str	str	string
bool	bool	number
boolean	bool	number
dict	dict	JSON-encoded array (dict)
dictionary	dict	JSON-encoded array (dict)
list	list	JSON-encoded array
array	list	JSON-encoded array

On the LUA side, you may want to use the following pattern for the `list` and `dict` return types:

```
return cjson.encode({
   a=1,
   b="2",
   c={
     d=42,
   },
})
```

Warning: There can be at most **one** *%return* statement in a given script.

6.2.2 Script inclusion

One of the main problems of Redis LUA scripts is that it doesn't support the LUA `require` keyword. To circumvent that limitation, the LUA script parsing logic in *redis_lua* handles `%include` statements, like so:

```
-- The "foo.lua" script in the same folder defines the "create_foo()"
-- function.

%include "foo"

local t = create_foo(1, "a");
```

%include takes a single argument, which is the complete name (with any relative path component) of the LUA script to include, without its `.lua` extension.

So if you have two scripts `foo/a.lua` and `bar/b.lua` each in a different subfolder of the `lua` directory, you can include `bar/b.lua` in `foo/a.lua` by simply adding the following *%include* statement:

```
%include '../bar/b'
```

Warning: For the inclusion system to work properly, all scripts must either been have loaded by the same call, or by different calls but using the same script cache.

Multiple inclusion

By default, *redis-lua* allows multiple inclusions of the same file several times. This can cause issues when including different scripts that include the same subscripts which conflict with each other.

To prevent side-effects caused by multiple inclusion of the same scripts, you can use the following statement, anywhere in the script:

```
%pragma once
```

Note: This behavior is new since version 2.0.0.

In previous versions, the default behavior was as-if *%pragma once* was defined implicitely in each script.

6.3 API

6.3.1 Script loading functions

These functions are the most common entry points for loading LUA scripts on disk.

`redis_lua.`**`load_all_scripts`**(*path*, *cache=None*)
Load all the LUA scripts found at the specified location.

Parameters

- **`path`** – A path to search into for LUA scripts.
- **`cache`** – A cache of scripts to use to fasten loading. If some *names* are in the *cache*, then they are taken from it.

Returns A list of scripts that were found, in arbitrary order.

`redis_lua.`**`load_scripts`**(*names*, *path*, *cache=None*)
Load several LUA scripts.

Parameters

- **`names`** – An iterable of LUA scripts names to load. If some names contain backslashes, those will be replaced with forward slashes silently.
- **`path`** – A path to search into for LUA scripts.
- **`cache`** – A cache of scripts to use to fasten loading. If some *names* are in the *cache*, then they are taken from it.

Returns A dict of scripts that were found.

Warning All scripts whose load succeeds are added in the cache immediately. This means that if some script fails to load and the function call throws, the cache will still have been updated.

`redis_lua.`**`load_script`**(*name*, *path*, *cache=None*, *ancestors=None*)
Load a LUA script.

Parameters

- **`name`** – The name of the LUA script to load, relative to the search *path*, without the '.lua' extension. If the name contains backslashes, those will be replaced with forward slashes silently.
- **`path`** – A path to search into for LUA scripts.

- **cache** – A cache of scripts to use to fasten loading. If *name* is in the *cache*, then the result is the same as calling *cache[name]*.
- **ancestors** – A list of names to consider as ancestors scripts.

Returns A *Script* instance.

6.3.2 Script running functions

These functions are using to execute LUA code on Redis servers.

redis_lua.**run_code**(*client*, *content*, *path=None*, *kwargs=None*, *cache=None*)
Run a LUA *script* on the specified *redis* instance.

Parameters

- **client** – The Redis or pipeline instance to execute the script on.
- **content** – The LUA code to execute.
- **path** – The path to search for for scripts inclusion. Can be *None*, in which case, any included script must exist in the cache.
- **kwargs** – A dict of arguments to call the script with.
- **cache** – The script cache to use to fasten script inclusion.

Returns The return value, as given by the script.

6.3.3 Script instances

All loaded scripts are wrapped into a *redis_lua.script.Script* instance.

class redis_lua.script.**Script**(*name*, *regions*)

get_line_info(*line*)
Get the line information for the specified line.

Parameters **line** – The line.

Returns The (real_line, real_line_count, line, line_count, region) tuple or *ValueError* if no such line exists.

classmethod **get_line_info_for_regions**(*regions*, *included_scripts*)
Get a list of tuples (first_real_line, real_line, real_line_count, first_line, line, line_count, region) for the specified list of regions.

Params regions A list of regions to get the line information from.

Params included_scripts A set of scripts that were visited already.

Returns A list of tuples.

get_real_line_content(*line*)
Get the real line content for the script at the specified line.

Parameters **line** – The line.

Returns A line content.

get_runner(*client*)
Get a runner for the script on the specified *client*.

Parameters **client** – The Redis instance to call the script on.

Returns The runner, a callable that takes the script named arguments and returns its result. If *client* is a pipeline, then the runner returns another callable, through which the resulting value must be passed to be parsed.

get_scripts_for_line(*line*)
Get the list of (script, line) by order of traversal for a given line.

Parameters **line** – The line.

Returns A list of (script, line) that got traversed by that line.

runner(*client*, ***kwargs*)
Call the script with its named arguments.

Returns The script result.

6.3.4 Low-level script functions

These functions are useful for people that want to perform more advanced operations, such as parsing a Script file manually from another source than a file.

redis_lua.**read_script**(*name*, *path*, *encoding=None*)
Read a LUA script.

Parameters

- **name** – The name of the LUA script to load, relative to the search *paths*, without the '.lua' extension. *name* may contain forward slash path separators to indicate that the script is to be found in a sub-directory.
- **path** – A path to search into for LUA scripts.
- **encoding** – The encoding to use to read the file. If none is specified, UTF-8 is assumed.

Returns The content of the script.

Raises If no such script is found, a `ScriptNotFoundError` is thrown.

redis_lua.**parse_script**(*name*, *content*, *path=None*, *cache=None*, *ancestors=None*)
Parse a LUA script.

Parameters

- **name** – The name of the LUA script to parse.
- **content** – The content of the script.
- **path** – The path of the script. Can be *None* if the script was loaded from memory. In this case, any included script must exist in the cache.
- **cache** – A dict of scripts that were already parsed. The resulting script is added to the cache. If the currently parsed script exists in the cache, it will be overriden.
- **ancestors** – A list of scripts that were called before this one. Used to detect infinite recursion.

Returns A *Script* instance.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

G

get_line_info() (redis_lua.script.Script method), 17
get_line_info_for_regions() (redis_lua.script.Script class method), 17
get_real_line_content() (redis_lua.script.Script method), 17
get_runner() (redis_lua.script.Script method), 17
get_scripts_for_line() (redis_lua.script.Script method), 18

L

load_all_scripts() (in module redis_lua), 16
load_script() (in module redis_lua), 16
load_scripts() (in module redis_lua), 16

P

parse_script() (in module redis_lua), 18

R

read_script() (in module redis_lua), 18
run_code() (in module redis_lua), 17
runner() (redis_lua.script.Script method), 18

S

Script (class in redis_lua.script), 17