Lua: A Guide for Redis Users

You’ve heard that Redis has an embedded scripting language, but haven’t given it a try yet? Here’s a tour of what you need to understand to use the power of Lua with your Redis server.

Hello, Lua!

Our first Redis Lua script just returns a value without actually interacting with Redis in any meaningful way:

This is as simple as it gets. The first line sets up a local variable with our message, and the second line returns that value from the Redis server to the client. Save this file locally as `hello.lua` and run it like so:

Connectivity Problems?

This `redis-cli` example assumes that you're running a Redis server locally. If you're working with a remote server like RedisGreen, you'll need to specify host and port information. Find the **Connect** button on your RedisGreen dashboard to quickly copy the login info for your server.

See also: Could not connect to Redis at 127.0.0.1:6379: Connection refused.

Running this will print “Hello, world!”. The first argument of `EVAL` is the complete lua script — here we’re using the `cat` command to read the script from a file. The second argument is the number of Redis keys that the script will access. Our simple “Hello World” script doesn’t access any keys, so we use `0`.

Accessing Keys and Arguments

Suppose we’re building a URL-shortener. Each time a URL comes in we want to store it and return a unique number that can be used to access the URL later.

We'll use a Lua script to get a unique ID from Redis using `INCR` and immediately store the URL in a hash that is keyed by the unique ID:

```
local link_id = redis.call("INCR", KEYS[1])
redis.call("HSET", KEYS[2], link_id, ARGV[1])
return link_id
```

We're accessing Redis for the first time here, using the `call()` function. `call()`'s arguments are the commands to send to Redis: first we `INCR <key>`, then we `HSET <key> <field> <value>`. These two commands will run sequentially — Redis won't do anything else while this script executes, and it will run extremely quickly.

We're accessing two Lua tables, `KEYS` and `ARGV`. Tables are associative arrays, and Lua's only mechanism for structuring data. For our purposes you can think of them as the equivalent of an array in whatever language you're most comfortable with, but note these two Lua-isms that trip up folks new to the language:

- Tables are one-based, that is, indexing starts at 1. So the first element in `mytable` is `mytable[1]`, the second is `mytable[2]`, etc.
- Tables cannot hold nil values. If an operation would yield a table of `[ 1, nil, 3, 4 ]`, the result will instead be `[ 1 ]` — the table is *truncated* at the first nil value.

When we invoke this script, we need to also pass along the values for the `KEYS` and `ARGV` tables:

```
redis-cli EVAL "$(cat incr-and-stor.lua)" 2 links:counter links:urls
http://malcolmgladwellbookgenerator.com/
```

This time when calling `EVAL`, after the script we provide `2` as the number of `KEYS` that will be accessed, then we list our `KEYS`, and finally we provide values for `ARGV`. Normally when we build apps with Redis Lua scripts, the

Redis client library will take care of specifying the number of keys, but we’re showing the actual command sent to Redis here for completeness.

Just to make things clear, here’s our original script again, this time with `KEYS` and `ARGV` expanded:

```
local link_id = redis.call("INCR", "links:counter")
redis.call("HSET", "links:urls", link_id,
"http://malcolmgladwellbookgenerator.com")
return link_id
```

When writing Lua scripts for Redis, every key that is accessed should be accessed only by the `KEYS` table. The `ARGV` table is used for parameter-passing — here it’s just the value of the URL we want to store.

Conditional Logic: increx and hincrex

Our example above saves the link for our URL-shortener, but we also need to track the number of times a URL has been accessed. To do that we’ll keep a counter in a hash in Redis. When a user comes along with a link identifier, we’ll check to see if it exists, and increment our counter for it if it does:

```
if redis.call("HEXISTS", KEYS[1], ARGV[1]) == 1 then
return redis.call("HINCRBY", KEYS[1], ARGV[1], 1)
else
return nil
end
```

Each time someone clicks on a shortlink, we run this script to track that the link was shared again. We invoke the script using `EVAL` and pass in `links:visits` for our single key and the link identifier returned from our previous script as the single argument.

The script would look almost the same without hashes. Here’s a script which increments a standard Redis key only if it exists:

```
if redis.call("EXISTS",KEYS[1]) == 1 then
return redis.call("INCR",KEYS[1])
else
return nil
end
```

SCRIPT LOAD and EVALSHA

Remember that when Redis is running a Lua script, it will not run anything else. The best scripts simply extend the existing Redis vocabulary of small atomic data operations with the smallest bit of logic necessary. Bugs in Lua scripts can lock up a Redis server altogether — best to keep things short and easy to debug.

Even though they're usually quite short, we need not specify the full Lua script each time we want to run one. In a real application you'll instead register each of your Lua scripts with Redis when your application boots (or when you deploy), then call the scripts later by their unique SHA-1 identifier.

```
redis-cli SCRIPT LOAD "return 'hello world'"
=> "5332031c6b470dc5a0dd9b4bf2030dea6d65de91"

redis-cli EVALSHA 5332031c6b470dc5a0dd9b4bf2030dea6d65de91 0
=> "hello world"
```

An explicit call to `SCRIPT LOAD` is usually unnecessary in a live application since `EVAL` implicitly loads the script that is passed to it. An application can attempt to `EVALSHA` optimistically and fall back to `EVAL` only if the script is not found.

If you're a Ruby programmer, take a look at Shopify's Wolverine, which simplifies the loading and storing of Lua scripts for Ruby apps. For PHP programmers, Predis supports adding Lua scripts to be called just as though they were normal Redis commands. If you use these or other tools to standardize your interaction with Lua, let me know — I'd be interested to find out what else is out there.

When to use Lua?

Redis support for Lua overlaps somewhat with `WATCH`/`MULTI`/`EXEC` blocks, which group operations so they are executed together. So how do you choose to use one over the other? Each operation in a `MULTI` block needs to be independent, but with Lua, later operations can depend on the results of earlier operations. Using Lua scripts can also avoid race conditions that can starve slow clients when `WATCH` is used.

From what we've seen at RedisGreen, most apps that use Lua will also use MULTI/EXEC, but not vice versa. Most successful Lua scripts are tiny, and just implement a single feature that your app needs but isn't a part of the Redis vocabulary.

Visiting the Library

The Redis Lua interpreter loads seven libraries: base, table, string, math, debug, cjson, and cmsgpack. The first several are standard libraries that allow you to do the basic operations you'd expect from any language. The last two let Redis understand JSON and MessagePack — this is an extremely useful feature, and I keep wondering why I don't see it used more often.

Web apps with public APIs tend to have JSON lying around all over. So maybe you have a bunch of JSON blobs stored in normal Redis keys and you want to access some particular values inside of them, as though you had stored them as a hash. With Redis JSON support, that's easy:

```
if redis.call("EXISTS", KEYS[1]) == 1 then
local payload = redis.call("GET", KEYS[1])
return cjson.decode(payload)[ARGV[1]]
```

```
else
return nil
end
```

Here we check to see if the key exists and quickly return nil if not. Then we get the JSON value out of Redis, parse it with `cjson.decode()`, and return the requested value.

```
redis-cli set apple '{ "color": "red", "type": "fruit" }'
=> OK

redis-cli eval "$(cat json-get.lua)" 1 apple type
=> "fruit"
```

Loading this script into your Redis server lets you treat JSON values stored in Redis as though they were hashes. If your objects are reasonably small, this is actually quite fast, even though we have to parse the value on each access.

If you're working on an internal API for a system that demands performance, you're likely to choose MessagePack over JSON, as it's smaller and faster. Luckily with Redis (as in most places), MessagePack is pretty much a drop-in replacement for JSON:

```
if redis.call("EXISTS", KEYS[1]) == 1 then
local payload = redis.call("GET", KEYS[1])
return cmsgpack.unpack(payload)[ARGV[1]]
else
return nil
end
```

Doing the Numbers

Lua and Redis have different type systems, so it's important to understand how values may change when crossing the Redis-Lua border.

When a number comes from Lua back to a Redis client, it becomes an integer — any digits past the decimal point are dropped:

```
local indiana_pi = 3.2
return indiana_pi
```

When you run this script, Redis will return an integer of 3 — you lose the interesting pieces of pi. Seems simple enough, but things get a bit more tricky when you start interacting with Redis in the middle of the script. An example:

```
local indiana_pi = 3.2
redis.call("SET", "pi", indiana_pi)
return redis.call("GET", "pi")
```

The resulting value here is a string: `"3.2"` Why? Redis doesn't have a dedicated numeric type. When we first `SET` the value, Redis saves it as a string, losing all record of the fact that Lua initially thought of the value as a float. When we pull the value out later, it's still a string.

Values in Redis that are accessed with `GET`/`SET` should be thought of as strings except when numeric operations like `INCR` and `DECR` are run against them. These special numeric operations will actually return integer replies (and manipulate the stored value according to mathematical rules), but the "type" of the value stored in Redis is still a string value.

Gotchas: A Summary

These are the most common errors that we see when working with Lua in Redis:

- Tables are one-based in Lua, unlike most popular languages. The first element in the KEYS table is `KEYS[1]`, the second is `KEYS[2]`, etc.
- A nil value terminates a table in Lua. So `[ 1, 2, nil, 3 ]` will

automatically become `[1, 2]`. Don’t use nil values in tables.

- `redis.call` will raise exception-style Lua errors, while `redis.pcall` will automatically trap any errors and return them as tables that can be inspected.

- Lua numbers are converted to integers when being sent to Redis — everything past the decimal point is lost. Convert any floating point numbers to strings before returning them.

- Be sure to specify all the keys you use in your Lua scripts in the `KEYS` table, otherwise your scripts will probably break in future versions of Redis.

- Lua scripts are just like any other operation in Redis: nothing else runs while they’re being executed. Think of scripts as a way to expand the vocabulary of the Redis server — keep them short and to-the-point.

Further Reading

There are lots of great resources for Lua and Redis online — here are a few I use:

- EVAL Docs
- RedisGreen’s Lua Script Library
- Lua Reference Manual
- Lua Tutorial Directory

Last updated 29 Jul 2015. Originally posted 18 Mar 2013 by Brian P O'Rourke

← Back to docs