EVERYWHERE.

Redis Lua Scripts

Day 0 Advanced Training,

Itamar Haber @itamarhaber #Redis #RedisConf

Evangely Technicalist

redisconf18

redislabs
home of redis

This session is about Redis Lua Scripting

- Redis has an embedded sandboxed Lua v5.1 engine.
- User scripts can be sent for execution by the server.
- When a script runs, it blocks the server and is atomic.
- Scripts have full access to the data.

The next slides are a mix of:

- Lua: syntax, data structures, control, …
- Redis: loading, running scripts & specific considerations
- Use cases for using Lua

Who We Are

redis

Open source. The leading **in-memory database platform**, supporting any high performance operational, analytics or hybrid use case.

redislabs
home of redis

The open source home and commercial provider of **Redis Enterprise (Redis^e)** technology, platform, products & services.

hello I am

Itamar Haber @itamarhaber, Evangely Technicalist, formerly Chief Developer Advocate & Chief OSS Education Officer

Happy 3rd 6379 Portday!

#MERZFTW

redislabs
home of redis

What is Lua https://www.lua.org/about.html

TL;DR it is a scripting language

> “ Lua is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description.
>
> Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed ...

Lua's Hello World

```lua
-- src: http://rosettacode.org/
print "Hello, World!"
```

- Single line comments begin with a double dash
- Single argument function calls need no parenthesis
- "This is" a string, and also 'this one'
- Whitespace is the statement separator/terminator; semicolon optional

Redis' Lua sandbox

As of v2.6, Redis embeds a Lua v5.1 engine for running user scripts.

For stability and security purposes, the engine is sandboxed and exposes only a subset of Lua's full functionality.

The main restrictions put by the sandbox are:

- Only `local` variables are permitted
- `dofile()` and `loadfile()` are disabled
- External libraries can't be used, the `require()` function is disabled

Sandbox built-in libraries

Standard libraries:

- Included: base, string, table and math
- Excluded: io, os and debug

Additional bundled libraries:

- bit bit manipulation routines
- struct encode/decode C-like structures
- cjson encode/decode JSON
- cmsgpack encode/decode MessagePack
- redis API to the Redis server

Running one-off scripts

```
redis> EVAL
        "return('I <' .. tostring(3) .. ' Lua')" 0
"I <3 Lua"
```

- EVAL expects the raw script as input - it is dev command
- The functions `tostring` and `tonumber` are usually implicitly called
- Remember the exotic string concatenation operator `..`
- The script can return a value with the `return` statement

Cached scripts

Redis caches the bytecode of **every script** it executes in order to avoid re-compiling it in subsequent calls.

Cached scripts offer two major performance gains:

- Sent to the server only[*] once
- Parsed to bytecode only[*] once

[*] More accurately is "on every SCRIPT LOAD"

Important: keep an eye on the script cache's size, it may explode.

Loading and running cached scripts

```
redis> SCRIPT LOAD "return 21 * 2.01"
       "935b7b8d888c7affc0bac8a49e63933c915b883f"
redis> EVALSHA
       935b7b8d888c7affc0bac8a49e63933c915b883f 0
(integer) 42
redis> EVALSHA nosuchscriptsha1 0
(error) NOSCRIPT No matching script. Please use
EVAL.
```

- SCRIPT LOAD returns the sha1sum of the script
- EVALSHA expects the sha1sum of the loaded script

```
# The Cache Explosion Scenario - misuse/antipattern
# Caches every script == EVAL cached, but silently
# P.S. this script should be parameterized, more below
redis> SCRIPT FLUSH
OK
redis> EVAL
       "local r = 21 * 2.01 return tostring(r)" 0
"42.21"
redis> EVAL
       "return redis.sha1hex(
       'local r = 21 * 2.01 return tostring(r)')" 0
"f8bc25b220fcd0406938b86d7cf9d29dce7d1ee8"
redis> EVALSHA
       f8bc25b220fcd0406938b86d7cf9d29dce7d1ee8 0
"42.21"
```

Lua has 8 data types

1. **Nil**: can only have the value `nil`
2. **Boolean**: can be either `true` or `false`
3. **Number**: 32-bit-dp-fp for reals and integers
4. **String**: 8-bit binary-safe strings
5. **Function** object: functions are 1st classers
6. **Userdata** object: C-bindings (n/a in our context)
7. **Thread** object: coroutines implementation (ditto)
8. **Table** object: associative array that's used for everything

Note: find out the type with Lua's `type()` function.

```lua
-- The table data type
local empty = {}
local associative = {
  ['key1'] = 'value1',
  [42] = true }
associative['newkey'] = {
  ['another_key'] = { 'table' }, }
local indexed = { 6, 'afraid', 7 }  -- 1-based index!
indexed[#indexed+1] = 'because'
table.insert(indexed, 7)
table.remove(indexed)
return indexed[4]
$ redis-cli --eval localtables.lua
"because"
```

Script parameters

Anti-pattern: hardcoding dynamic values in the script, causing the cache to literally explode.

Instead, parametrize your scripts.

Redis distinguishes between two types of parameters:

1. Keys are the names of all keys accessed by the script
2. Arguments are anything excluding any names of keys

Again, why two types of parameters?

Generating key names in the script, programmatically and/or based on the data set, **is not advised**.

While Redis does not enforce it, doing so may lead to unknown results, and complexifies tracking software issues.

Additionally, this usually makes a script totally incompatible with the Redis cluster, unless special care is taken. And even then it is not advised.

```
# KEYS and ARGV are prepopulated indexed tables
$ redis-cli EVAL
  "return { ARGV[1], ARGV[2], ARGV[3] }"
  0 foo bar baz
1) "foo"
2) "bar"
3) "baz"
$ redis-cli EVAL
  "return { KEYS[1], { ARGV[1], ARGV[2] } }"
  1 foo bar baz
1) "foo"
2) 1) "bar"
   2) "baz"
```

Accessing the Redis server from a script

```
-- Executes a Redis PING
local reply = redis.call('PING')
return reply

$ redis-cli EVAL "$(cat ping.lua)" 0
PONG
```

As simple as that :)

```lua
--[[ This adds a the same members (ARGV) to two
regular sets (KEYS[1], KEYS[2]) ]]--
local set1, set2 = KEYS[1], KEYS[2]
local rep1 = redis.call(
    'SADD', set1, unpack(ARGV))
local rep2 = redis.call(
    'SADD', set2, unpack(ARGV))
return rep1 + rep2
```

```
$ redis-cli --eval msadd.lua s s1 , e1 e2
(integer) 4
$ redis-cli --eval msadd.lua s s2 , e3 e4
(integer) 4
```

Prophecies about the #1 cause for your future...

1. Runtime errors: is forgetting about Lua's `unpack()`, a.k.a the "splat" operator
2. bugs: will be `redis.call()` not returning what you thought it should/could
3. "$!@$^%#@%#!!!!": redis-cli --eval , put a space before and after comma!

Data type conversion: redis.call() to Lua

The Redis type	Is converted to Lua's
Integer reply	Number
Bulk string reply	String
Multi bulk reply	Table, may be nested
Status reply	Table with a single field, ok, set in it
Error reply	Table with a single field, err, set in it
Nil	Boolean false

Data type conversion: Lua -> Redis (i.e. return)

The Lua type	**Is converted to Redis'**
Number	Integer reply (here be dragons)
String	Bulk string reply
Table (indexed)	Multi bulk reply (truncated 1st nil, if any)
Table with a single ok field	Status reply
Table with a single err field	Error reply

Data type conversion: Lua -> Redis, redis.call

The Lua type	Is converted to Redis'
Nil	Nil bulk reply
Boolean false	Nil bulk reply
Boolean true	Integer reply with value of 1 **Note:** has no corresponding conversion

4+1 more conversions conversation pieces

1. When intending to return/pass to Redis an integer from a Lua number, make sure that you're doing that.
   Helpful but dirty trick:

   ```
       tonumber(string.format('%.0f', 3.14))
   ```
2. RESP has no floating points, so you'll need to return them as strings
3. There is no simple way to have nils inside indexed arrays because Lua table semantics, so when Redis converts a Lua array into Redis protocol the conversion is stopped if a nil is encountered.

The Fourth, the Fifth

4. The conversion of Lua's associative arrays, i.e. tables that are not arrays/lists, is similarly non-trivial, so when Redis converts an associative Lua array into Redis protocol the conversion is stopped when the first key in the array is encountered.
5. Assuming you know what redis.call() will return for a command can lead to ~~embarrassing mistakes~~ software issues.

```
# What's the time and what's with determinism?
$ redis-cli EVAL
            "return redis.call('TIME')" 0
1) "1524107683"
2) "501253"
$ redis-cli EVAL "local t = redis.call('TIME')
            return redis.call('SET', KEYS[1], t[1])"
            1 now
ERR Write commands not allowed after non deterministic
commands. Call redis.replicate_commands()...
$ redis-cli EVAL "redis.replicate_commands()
            redis.set_repl(redis.REPL_ALL) -- AOF|SLAVE|NONE
            local t = redis.call('TIME')
            return redis.call('SET', KEYS[1], t[1])"
            1 now
OK
```

redislabs
home of redis

```lua
-- Control structure: the if statement
if nil then
    -- some logic here
elseif false then
    -- optionally more logic here
else
    -- this always happens
end

return 'if execution completed successfully'
```

```
-- Control structures: while and repeat loops
while false do
   -- your code here
end

repeat
   -- all this space is yours
until 0 -- Note: everything else is true, also 0

return 'like black hole, nothing gets out of an endless
loop'
```

redislabs
home of redis

```lua
-- Control structure: two type of a for loop
local sum1 = 0
for i = 1, #ARGV, 1 do
   sum1 = sum1 + tonumber(ARGV[i])
end
local sum2 = 0
for i, v in pairs(ARGV) do
   sum2 = sum2 + tonumber(v)
end
if sum1 ~= sum2 then
   return redis.error_reply('Something meh here')
end
return redis.status_reply('All good \\o/')
```

```
$ redis-cli EVAL 'while 0 do end' 0
^C
$ redis-cli PING
(error) BUSY Redis is busy running a script. You can
only call SCRIPT KILL or SHUTDOWN NOSAVE.
$ redis-cli SCRIPT KILL
OK
$ redis-cli CONFIG GET lua-time-limit
1) "lua-time-limit"
2) "5000"
$ redis-cli EVAL 'while 0 do break end return "phew"' 0
"phew"
```

```lua
-- Use case: reduce client -> server bandwidth
-- LMPOP key count
local key = KEYS[1]
local count = table.remove(ARGV, 1)
local reply = {}
for _ = 1, count do
    local ele = redis.call('LPOP', key)
    if ele then
        table.insert(reply, ele)
    end
end
return reply
```

```lua
-- Use case: reduce server -> client bandwidth
-- ZSUMRANGE key start stop
local total = 0
local data = redis.call(
    'ZRANGE', KEYS[1], ARGV[1], ARGV[2], 'WITHSCORES')
while #data > 0 do
   -- ZRANGE replies with an array in which
   -- the scores are at even indices
   total = total + tonumber(table.remove(data))
   table.remove(data)
end
return total
```

Use Case: pure functions, better transactions and composable commands

Lua scripts are intended to be pure functions.

If you must, store a state in a key in the database, but always explicitly pass its name.

Lua scripts, being atomic and quite powerful, are often a nicer alternative to using WATCH/MULTI/EXEC/DISCARD and retry. And most times also run faster.

Put differently: Lua lets **compose** commands using the existing API.

So much more important stuff left to cover...

- Dealing with errors by calling redis.pcall()
- The Redis Lua stepping debugger, aka ldb
- Controlling script effects and replication
- Common pitfalls and considerations, also in testing
- More Lua patterns

Do not fear though! Most is available at online at: https://redis.io/commands/eval, https://redis.io/topics/ldb, and https://www.lua.org/manual/5.1/manual.html