Lua Pitfalls

Dmitry Kotelnikov
IPONWEB

Lua Pitfalls

- For beginners in Lua
- Especially with habits for other language
- Subtle errors, serious consequences
- Based on real events
- No claim to completeness
- Imagine a mistake to remember the pitfall

Goal

- People bypass couple of pitfalls
- Save time and money for important things

IPONWEB

- Custom-built real-trading platform
- Dozens of Lua developers
- Business logic in Lua, tests in Perl
- Compact code
- Rapid integration of the newcomers
- Billions of the requests
- Opportunity for a mistake

nil in tables

- Well-known
- Simple, but different
- Intentional remove is obvious, unintentional not so
- Subtle difference

```
local fruits = {lemon = 1, melon = 1}
fruits.melon = nil
print(fruits.melon)
```

- Let's make an error to understand

Big Bang

- Habit of Perl + MongoDB
- Actually Lua + MongoDB

```
mongo.remove({
    collection = 'user',
    _id = self.user_id,
})
```

Big Bang

- Habit of Perl + MongoDB
- Actually Lua + MongoDB

```
mongo.remove({
    collection = 'user',
    _id = self.user_id,
})
```

- I removed all users
- Feel bad for a week or two
- Good! Now I'll remember -)

little excuse

In Perl + MongoDB query removes nothing

```
db.user.remove({_id: null})
```

Bypass the pitfall

- Improve workflow
- Check if absence and undefined value is the same
- Deny queries without _id

Global vars

- Habits for Apache
- Global object to store user data
- Synthetic tests passed
- Small traffic tests passed
- Statistic checks passed
- Actually large number of incorrect data

Big Bang

- Coroutines + Yields
- Requests overlap

```
function Controller:handle()
    -- accidentally or by intention missed 'local'
    user = {}
    user.gender = get_from_cookie('gender')
    -- then yield, and so go to the other request
    user.history = get_from_mongo(
        self.id, 'history')
    -- then resume and probably get other gender
    log(self.id, user.gender)
end
```

- Users change gender many times per day

Bypass the pitfall

- Pass context manually

```
function Request:handle()
    self.context = {}
    self.context.profiler = Profiler:new()
    self.context.user = User:new()
    local targeting = Targeting:new(self.context)
    ...
end
```

- Make global environment read-only

```
_ENV = {}
setmetatable(_ENV, {
    __index = _G,
    __newindex = function (t, k, v) error('read only') end
})

-- saves from typos like
local user_id
..
uesr_id = 5
local targeting = Targeting:new(uesr_id)
```

Just ideas

- Restore _ENV

```
function Request:handle()
    _ENV = setmetatable({}, {__index = _G})
    PROFILER = Profiler:new()
    ...
end
function my_yield(coroutine)
    local old_env = _ENV
    coroutine.yield()
    _ENV = old_env
end
```

- Index by coroutine.running()

```
COROUTINE_CONTEXT[coroutine.running()][name]
```

- Set and get coroutine environment

```
function Request:handle()
    setfenv(0, setmetatable({}, {__index = _G}))
    getfenv(0).PROFILER = Profiler:new()
    ...
end
```

Function declaration order

- Well-known that function is first-class value
- Simple, but unusual
- Must be declared before usage
- Do not reflect because of single entry point
- Coding Style wars

Bang #1

```
-- before
local function normalize()
    ...
end
function parse_url(str)
    ...
    return normalize(url)
end

-- after
function parse_url(str)
    ...
    return _normalize(url)
end
function _normalize()
    ...
end

-- bang
function parse_url(str)
    ...
    return _normalize(url)
end
local function _normalize()
    ...
end
```

Bang #2

```
-- before
function parse_domain(str)
    ...
    return domain
end
local function _normalize()
    ...
end
function parse_url(str)
    ...
    return _normalize(url)
end

-- bang
function parse_domain(str)
    ...
    return _normalize(domain)
end
local function _normalize()
    ...
end
function parse_url(str)
    ...
    return _normalize(url)
end
```

Bypass the pitfall

- Introduce Coding Style
- Order independent and really private

```
local _normalize
function parse_url(str)
    ...
    return _normalize(url)
end
function _normalize()
    ...
end
```

- Order independent and simple

```
function parse_url(str)
    ...
    return _normalize(url)
end
function _normalize()
    ...
end
```

Accident comma

- Poorly visible

```
local name = 'Subtle comma'
local color = 'white'
local height = 10
local width = 10,
log('document',
      name,
      color,
      height,
      width,
      date
)
```

- Introduce code formatter

```
local name = 'Subtle comma'
local color = 'white'
local height = 10
local width = 10, log('document', name, color, height, width, date)
```

Pitfalls with typing

- Keys may have any type
- Condition is false only given nil or false

```
# bad
my $id_allowed = {5 => 0, 7 => 1, __key_type => 'number'};
# good
my $id_allowed = lua_table({5 => lua_false(), 7 => lua_true()},
'number');
```

- Be carefull in converting data to/from other language
- Don't use automatic type detection
- Don't use "fast" solutions
- Use objects

Pitfalls with typing

- There is no type array
- Use objects

```
-- bad
local array = {7, 8, __as_array = true}
-- good
local array = json_array({7, 8})
```

- Value may change the type
- Use validation or force typing

```
-- bad
local request = from_json(body)
if (request.id == 5) do ... end

-- good
request.id = tonumber(request.id)
```

List of pitfalls

- Nil key is the same as absent key
- Global vars mix data between coroutines
- Misspell of variable makes it global
- Function declaration order is crucial
- Accident comma silently breaks code
- Table stores keys of any type
- Condition is false only given nil or false
- There is no type array
- Variable may change the type

- Function may change passed table
- Arrays indices start at one, not zero
- nil in arrays breaks array length
- gsub does not make in-place replacement

Final recommendations

- Learn the differences between languages
- Share your knowledge
- Use Coding Style
- Make code more reliable than needed
- Have fun -)

Thanks!

Dmitry Kotelnikov
IPONWEB
kotelnikovdv@gmail.com