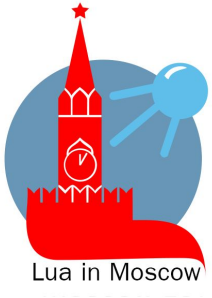


DSL на Lua как конечный автомат

Харитонова Екатерина, LogicEditor

DSL

DSL (**D**omain-**S**pecific **L**anguage / Предметно-Ориентированный Язык) - язык программирования с ограниченными выразительными возможностями, ориентированный на некую конкретную предметную область.
Должен быть прост в освоении и использовании за счет поддержки минимума возможностей, необходимых для своей предметной области.

Внутренний DSL - язык, созданный на базе существующего языка общего назначения. Корректный код в синтаксисе языка общего назначения, использующий лишь необходимое подмножество возможностей языка.

М. Фаулер “Предметно-ориентированные языки”

Требования

- Удобный синтаксис
- Валидация, диагностика
- Удобство поддержки:
  - Легко добавлять новые выражения
  - Легко добавлять новые задачи

Синтаксис внутреннего DSL на Lua

```
namespace:method "title"
{
  data = "here";
}
namespace:method "title"
{
    subnode:method "subnode title"
    {
        subnode_data = "here";
    };
}
```

```
.method2 "title2"
[[
    text data
]]

.metod3 “title”
```

Синтаксис внутреннего DSL на Lua

```
[branch "master"]
        remote = origin
        merge = refs/heads/master
```

```
git:branch "master"
{
  remote = "origin";
  merge = "refs/heads/master";
}
```

Синтаксический сахар

```
print "a"        ~        print ("a")

table:key ( … )        ~        table.key (table, ...)
```

Синтаксис внутреннего DSL на Lua

```
git:branch "master"
{
  remote = "origin";
  merge = "refs/heads/master";
}
```

Без синтаксического сахара:

```
_G["git"].branch (git, "title")
({
  remote = "origin";
  merge = "refs/heads/master";
})
```

```
local namespace = _G["git"]
local method = namespace["branch"]
local tmp = method(namespace, “master”)
tmp ({remote = "origin"; merge =
"refs/heads/master";})
```

```
git.branch = function (self, name)
    return function (data)
        ….
    end
end
```

Метод __index

__index вызывается при обращении к несуществующему ключу таблицы

```
t = { }

setmetatable(
    t,
    { __index = function(table, key)
        local v = 1
        table[key] = v
        return v
    end,
})

print(t.a, t.b, t.c)          -->    1      1      1
```

Метод __call

__call вызывается при вызове талицы в качестве функции

```
t = { }
setmetatable(
    t,
    { __call = function(t, ...)
        print(“hello”)
    end,
})

t()          -->   hello
```

```
local proxy = function(namespace)
    local self = { }
    return setmetatable(
        self,
        {
            __index = function(t, k)
                -- do smth
            end;
            __call = function(t, ...)
                -- do smth

            end;
        }
    )

setmetatable(_G, { __index = function(t, k) return proxy(k) end })
```

```
git:branch "master"
{
  remote = "origin";
  merge = "refs/heads/master";
}

Без синтаксического сахара:
_G["git"].branch (git, "title")
({
  remote = "origin";
  merge = "refs/heads/master";
})
```

Старый подход

1. Загрузка данных в дерево
2. Обход дерева:
    a. Валидация (+ Диагностика)
    b. Использование данных DSL

Недостатки

- Ограничение в выразительных возможностях языка
- Необходимость дорабатывать базовую логику при добавлении новых типов выражений
- Загрузка в дерево любых данных

Синтаксис внутреннего DSL на Lua

Выражение на DSL:

```
git:branch "master"
{
  remote = "origin";
  merge = "refs/heads/master";
}
```

Без синтаксического сахара:

```
_G["git"].branch (git, "title")
({
  remote = "origin";
  merge = "refs/heads/master";
})
```

А вот так он исполняется:

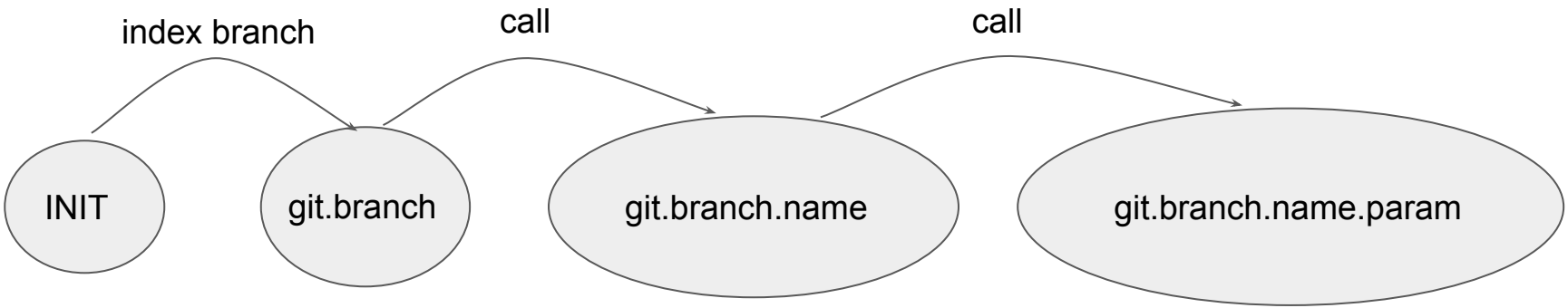
```
local namespace = _G["git"]
local method = namespace["branch"]
local tmp = method(namespace, “master”)
tmp ({remote = "origin"; merge = "refs/heads/master";})
```

FSM

Конечный автомат — абстрактный автомат, число возможных внутренних состояний которого конечно.

Это абстрактная машина, которая может находиться в точно одном из конечного числа состояний в любой момент времени. FSM определяется списком его состояний, его начальным состоянием и условиями для каждого перехода.

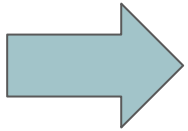
Описание конструкций языка как FSM



DSL на Lua как конечный автомат

Proxy объект (умеет обращаться к описанию нашего DSL)

Описание DSL как конечного автомата (конфиг, который задает все возможные состояния, переходы, обработчики перехода)



profit!

```
states = {
    ["git.branch"] = {
      type = "index";
      "git.branch.name";
      value = "branch";
    };

    ["git.branch.name"] = {
      type = "call";
      "git.branch.name.param";
      handler = function(self, t, _, name)
          t.name = name
      end;
    };
```

```
    ["git.branch.name.param"] ={
        type = "call";
        false; -- Terminal state

        handler = function(self, t, param)
            io.write("[branch \"", t.name, "\"]\n")
            for k, v in pairs(param) do
                io.write("\t", k, " = ", v)
            end
        end;
    };
```

```
cat "this" " is " "fun"
```

```
states = {
    ["call"] = {
        type = "call";

        "call"; -- A self-reference
        false; -- Terminal state


        handler = function(self, t, text)
            io.write(text)
        end;
    }
```

```
["foo.bar.name.param"] = {
    type = "call";
    false; -- Terminal state
        handler = function(self, t, param)
            param.id = "foo:bar"
            param.name = t.name
            param.data = param
            return param
        end;
    };
```

[Создание DSL на Lua (декабрь 2017)]

```
git:branch "master"
{
  remote = "origin";
  merge = "refs/heads/master";
}

tree = {
    [1] = {
        name = "master";
        id = "git:branch";
        data = {
            remote = "origin";
            merge = "refs/heads/master";
        }
    }
```

Итог

- Удобный синтаксис: минимум лишнего
- Не нужно писать интерпретатор
- Легко добавлять новые способы использования языка, используя уже готовый код
- Гибкость синтаксиса (возможность поддерживать совершенно разные структуры в DSL)

Спасибо за внимание

Вопросы?

ek@logiceditor.com