Язык программирования Lua

Объектно-ориентированное и прототипно-ориентированное программирование. Виртуальная машина. Управление памятью.

гр. 5030102/20201
Смирнова А. П.
Грушин А. Д.

Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) — парадигма, в которой программа описывается через объекты: наборы данных (состояние) и методов (поведение).

Классическое ООП предполагает наличие классов, наследования и полиморфизма.

В Lua нет встроенного понятия «класс», но все необходимые механизмы для ООП есть: таблицы, метатаблицы, функции и замыкания. На их основе можно построить как классическое, так и прототипное ООП.

ООП. Таблицы как объекты.

В Lua объектом обычно делают таблицу: поля таблицы — это свойства и методы объекта.

```
local user = {
  name = "Alice",
  greet = function(self)
    print("Hello, " .. self.name)
  end
}
```

Тут user — объект, name — его состояние, а greet — метод.

ООП. Класс через метатаблицу

Распространённый паттерн — делать «класс» таблицей-прототипом и использовать метатаблицу с полем __index для делегирования:

```
local Point = {}
Point.__index = Point  -- прототип для всех объектов Point

function Point:new(x, y)
  local obj = { x = x, y = y }
  return setmetatable(obj, self) -- если поле не найдено в obj,
оно ищется в Point через __index
end

function Point:len()
  return math.sqrt(self.x^2 + self.y^2)
end

local p = Point:new(3, 4)
print(p:len())  -- методный вызов с неявным self = p. Вывод: 5
```

Point выступает в роли «класса» и прототипа по умолчанию.

ООП. Инкапсуляция и «приватные» данные

Прямой поддержки модификаторов доступа (private, public) в Lua нет, но инкапсуляцию можно реализовать через локальные переменные и замыкания:

```lua
local function newCounter()
  local count = 0        -- «приватное» состояние
  local obj = {}

  function obj:inc()
    count = count + 1
    return count
  end

  return obj
end

local c = newCounter()
print(c:inc())  -- 1
print(c:inc())  -- 2
```

Переменная count недоступна снаружи, но жива в замыкании метода inc, что соответствует идее скрытого состояния объекта.

Прототипно-ориентированное программирование

Прототипно-ориентированное программирование — разновидность ООП, в которой нет строго разделения на классы и экземпляры. Вместо этого есть объекты-прототипы, от которых создаются новые объекты путём клонирования или делегирования.

Особенности прототипного подхода:

- Любой объект может выступать прототипом для других.
- «Наследование» реализуется цепочкой делегирования: если поле не найдено в объекте, оно ищется в его прототипе.
- Изменения прототипа автоматически отражаются во всех объектах, которые на него ссылаются.

Lua исторически ближе к прототипному ООП: «класс» — это всего лишь таблица-прототип; наследование строится через __index.

Прототипно-ориентированное программирование. Делегирование через __index

Простой пример прототипов:

```
local Enemy = {
  hp = 100,
  attack = 10
}  -- Enemy — прототип, содержащий общие поля.

local enemy1 = setmetatable({ name = "Slime" }, {
__index = Enemy })
local enemy2 = setmetatable({ name = "Orc", attack =
20 }, { __index = Enemy }) -- enemy1 и enemy2
делегируют отсутствующие поля в Enemy через __index.

print(enemy1.hp)      -- 100 (берётся из прототипа)
print(enemy2.hp)      -- 100
enemy1.hp = 50  -- При записи enemy1.hp = 50 создаётся
собственное поле, не влияющее на прототип и другие
объекты.
print(enemy1.hp)      -- 50 (собственное поле)
print(enemy2.hp)      -- 100 (остался у прототипа)
```

Виртуальная машина

Lua — компилируемый язык: исходный код сначала переводится в байткод, который затем исполняется виртуальной машиной.

Этапы выполнения программы:

1. Парсер читает исходный текст, строит синтаксическое дерево.
2. Компилятор генерирует байткод для виртуальной машины Lua.
3. Интерпретатор виртуальной машины пошагово выполняет инструкции байткода.

Такой подход обеспечивает:

- переносимость (один и тот же байткод работает на разных платформах);
- компактность и скорость загрузки;
- возможность анализа и оптимизации кода на уровне байткода.

Виртуальная машина. Регистровая архитектура

Виртуальная машина Lua — регистровая (начиная с Lua 5.0): каждая функция работает с фиксированным набором виртуальных регистров, а большинство инструкций имеют вид «операция над регистрами».

Особенности:

- Меньше инструкций по сравнению с чисто стековой VM, что даёт более компактный и быстрый байткод.
- Легче реализовать оптимизации на уровне регистров.
- Каждый вызов функции создаёт новый «кадр», содержащий набор регистров, локальные переменные и ссылки на внешние (upvalues).

Программист обычно не работает напрямую с байткодом, но понимание регистровой модели помогает объяснить особенности производительности и области видимости в Lua.

Виртуальная машина. Lua и C-API

Lua часто встраивается в приложения на C/C++. В таком случае виртуальная машина живёт внутри процесса и предоставляет стековый интерфейс (Lua stack) для взаимодействия с кодом на C.

- Внутри VM — регистровая архитектура для байткода.
- На границе с C — стековая модель (lua_push*, lua_gettable, lua_call и т.д.).

Это позволяет:

- эффективно вызывать функции Lua из C и наоборот;
- передавать данные между мирами (числа, строки, таблицы, userdata);
- расширять язык через C-модули, оставаясь при этом на общей виртуальной машине.

Управление памятью

Все сложные объекты Lua (таблицы, строки, функции, сопрограммы, userdata) хранятся в куче и управляются сборщиком мусора.

Особенности модели памяти:

- Программист не освобождает память явно (нет free или delete).
- Объект живёт, пока на него есть хотя бы одна сильная ссылка.
- Когда объект становится недостижимым, сборщик мусора может освободить его память.

Примитивы вроде чисел и булевых значений передаются по значению, но строки в Lua иммутабельны и обычно интернируются (одна копия одинаковой строки разделяется между всеми использованиями).

Сборщик мусора

Lua использует инкрементный сборщик мусора на основе алгоритма «mark-and-sweep» (пометка и зачистка), с поддержкой поколений объектов.

Основные идеи:

- На фазе «mark» сборщик обходит граф объектов, начиная с корней (глобальные переменные, стек, регистры) и помечает достижимые объекты.
- На фазе «sweep» он проходит по всем объектам и освобождает те, которые не были помечены.
- Инкрементность означает, что работа сборщика разбивается на небольшие шаги и распределяется во времени, чтобы уменьшить паузы.
- Поколенческий режим оптимизирует работу с учётом того, что большинство объектов живут недолго, а старые объекты «подозрительны» реже.

Параметры работы GC и режимы можно настраивать функцией collectgarbage.

Сборщик мусора

Функция collectgarbage позволяет управлять сборщиком мусора из Lua-кода:

```
collectgarbage("collect") -- запустить сборку
collectgarbage("stop") -- остановить GC
collectgarbage("restart") -- возобновить GC
collectgarbage("setpause", 200)
collectgarbage("setstepmul", 200)
```

Для userdata можно определить метаметод __gc в метатаблице и освободить внешние ресурсы (файлы, дескрипторы, память в C) при уничтожении объекта.

Источники

При создании этой презентации использовалась информация из документации с официального сайта языка Lua (https://www.lua.org).