Язык программирования Lua

Обработка ошибок и управление сложностью кода

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

Обработка ошибок. Исключения

В Lua нет привычных `try/catch`. Вместо этого есть функция `error( ... )`, которая “бросает” ошибку, и защищенные вызовы `pcall/xpcall`, которые ее “ловят”.

```
error(message[, level])
```

`error()` бросает ошибку, код дальше не выполняется, а управление передается ближайшему `pcall/xpcall`. Значение `level` сдвигает “виноватую” строку: 1 - виновата строка внутри error (по умолчанию), 2 - виноват вызывающий код, 0 - не добавлять позицию.

Например, ставить значение 2 удобно при проверке аргументов:

```
local function must_be_pos(n)
  if n ≤ 0 then error("n must be > 0", 2) end
end
```

Обработка ошибок. Исключения. pcall()

```
local ok, result_or_err = pcall(risky_function,
123)
if ok then
  print("всё ок:", result_or_err)
else
  print("ошибка:", result_or_err)
end
```

`pcall(f, ...)` запускает функцию `f(...)` “безопасно” и возвращает `true, <результат>` при успехе, или `false, <сообщение об ошибке>` при падении.
Перехваченные ошибки НЕ проходят через обработчик сообщений.

Обработка ошибок. Исключения. xpcall()

`xpcall` работает как `pcall`, но с обработчиком сообщений.

```
local function with_trace(err)
  return debug.traceback(tostring(err), 2)  -- 2: скрыть обёртку
end

local ok, res = xpcall(risky_function, with_trace)
if not ok then io.stderr:write(res, "\n") end
```

```
debug.traceback([thread,] [message [, level]])
```

Возвращает строку с трейсбеком; `level` задаёт, с какого уровня начинать (обычно 2, чтобы не показывать обёртку-обработчик).

Обработка ошибок. Исключения. assert()

Также в Lua есь возможность короткой проверки:

```
assert(v [, message])
```

Если `v` - ложь, то кидает ошибку с `message` (или текстом по умолчанию: `"assertion failed!"`).
Если `v` - истина, то возвращает все свои аргументы.

Используя `assert()` удобно “прокидывать” значения дальше.

```
local fh = assert(io.open("data.txt", "r"))  -- вернёт файловый дескриптор или упадёт
```

Тонкость: аргументы функции вычисляются до вызова `assert`, поэтому не следует класть в неё тяжёлые вычисления.

Управление сложностью кода. Пространство имен

В Lua нет как таковых пространств имен, эту роль выполняет таблица.

Выражение A.B в Lua — это доступ к полю таблицы: A["B"]. Поэтому если сложить функции и константы в одну таблицу, получится «имя.что-то» — по сути, namespace.

Управление сложностью кода. Модули

Модуль - это обычный Lua-файл, который при загрузке возвращает значение (чаще таблицу с функциями).

Загружается через `require`. Первый успешный `require` выполняет файл и кладёт результат в кеш `package.loaded[modname]`; последующие вызовы просто отдают кеш, не выполняя файл еще раз.

В Lua 5.4 `require` также возвращает вторым значением «`loader data`» — метаданные о том, откуда модуль найден (например, путь к файлу).

Управление сложностью кода. Модули

```lua
-- файл: my/math_ex.lua
local M = {}
function M.len2(x,y) return math.sqrt(x*x + y*y)
end
return M  -- экспорт

-- использование файла:
local math_ex, where = require("my.math_ex")
print(math_ex.len2(3,4))  -- 5
print("загружено из:", where)
```

Управление сложностью кода. Экспорт/импорт

Ранее мы уже сказали про функцию импорта - `require`. Модуль выполняется один раз и кэшируется в package.loaded, последующие require возвращают тот же объект.

При экспорте все, что не было возвращено и было объявлено как local, остается приватным.

Источники

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