把 Lisp 代码塞进 OpenResty

Urn 语言简介

wangkeyi@huobi.com

2018-11-18

OpenResty 主要用来分流和限流，其中有一小部分的逻辑是用 Urn 写的。

我们的 OpenResty 每天处理大概 40亿次请求，其中 1% 会被 Reject 掉。

现在主要工作是用 Clojure 搞的，所以对 Urn 这种风格和定位与 Clojure 都比较接近的语言很喜欢。

回顾 Lisp 历史，最早的 GC、动态类型、非常早的 OOP。

John McCarthy、Mit 6001: SICP。

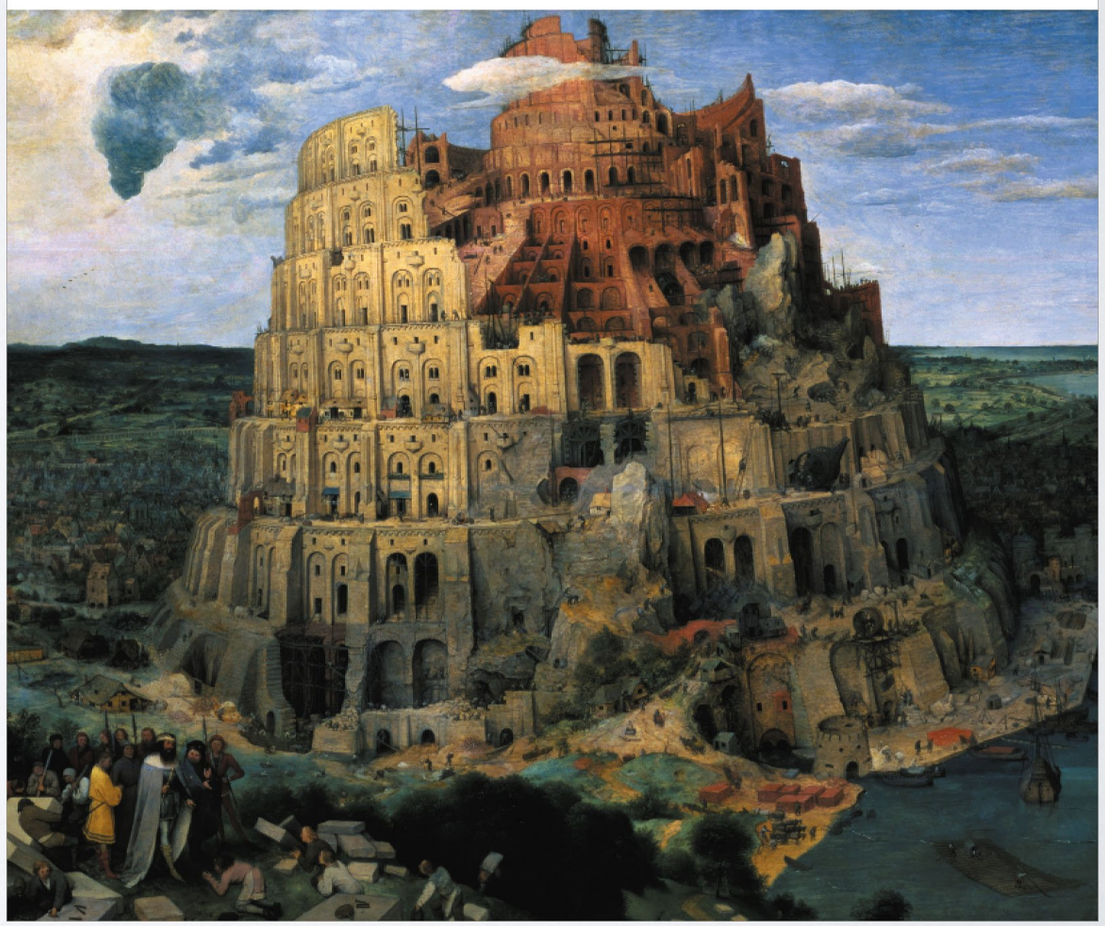
PaulGraham
viewed 卖了 5000w 刀
介绍文化类吹水书籍：黑客与画家
超越平凡、百年语言

"Brimming with contrarian insight and practical wisdom."
—Andy Hertzfeld, co-creator of the Macintosh computer

PAUL GRAHAM

HACKERS & PAINTERS

BIG IDEAS FROM THE COMPUTER AGE



今天要介绍的 Urn 是一种 Lisp 方言

其实从大会预告到现在这么长时间，感兴趣的同学可能已经对 Urn 有了一定的了解，玩过 Lisp 的同学可能花几小时就可以基本掌握了。
听了之前几个比较深入的主题，可以在这个略水的时间段稍稍休息一会儿。

Urn:
A Lisp implementation for Lua

https://urn-lang.com

编译型语言，目标代码最小化。不会带上没有用到的 runtime。

Lisp 的语法基本上就是这个样子了，特点是括号多 ...

Basic Syntax:

> (define foo "bar") out = "bar"	> foo = "bar"
> (print! "hello, world") hello, world out = nil	> print("hello,world") hello, world
> (+ 1 2 3) out = 6	> 1 + 2 + 3 6
> (if (> 2 3) . "bigger" . "smaller") out = "smaller"	> if 2 > 3 then >> return "bigger" >> else >> return "smaller" >> end smaller
> (defun add [a b] . (+ a b)) out = function: 0x7fded8d906f0	> function add (a, b) >> return (a + b) >> end

这一页介绍 String 上的操作，以 split 为例

如果这里写 “更好的 Lua” 会不会引战 ...?

更顺手的 Lua

如果你接触 Lua 时间不长，并且之前熟悉别的语言，那么看到这个页面之后可能会有一些迷茫

这就比较自然了，需要注意的是，这是按 Lua pattern 进行切分的，与正则表达式有一些区别。

http://lua-users.org/wiki/SplitJoin

```
(import core/string str)
(str/split "hello,world" ",")

;; out = ("hello" "world")
```

```
local ngx_re = require "ngx.re"

local res, err = ngx_re.split("a,b,c,d", "(,)")
```

更多功能，比如：字符串模板，见文档

毕竟 ngx.re 这个模块也是要搜一搜才知道的 ……

更顺手的 Lua

原生的 Lua 很多比较基础的功能是需要大家人肉补全的。
这一页介绍一些 List 上的操作。

```
> (elem? 256 '(32 64 128 256))
out = true
```

Urn 是区分 list 与 table 的。

```
function has_value (tab, val)
    for index, value in ipairs(tab) do
        if value == val then
            return true
        end
    end
    return false
end
> has_value ({32, 64, 128, 256, 512}, 256)
true
```

```
> (append '(1 2) '(3 4))
out = (1 2 3 4)
```

```
for i = 1, #t1 do
   t2[#t2+1] = t1[i]
end
```

```
> (flatten '((1 2) (3 4) '(5 6)))
out = (1 2 3 4 5 6)
```

Urn 实现了一个 Set 模块，不过缺少差集和补集等操作。可以替代一些场景下的 table 操作

更顺手的 Lua

顺便说明一下 import 的有限引入以及别名的细节。

```
(import data/set (set-of intersection))

(intersection (set-of 1 2 3) (set-of 3 4 5))
;; out = «hash-set: 3»

(intersection (set-of 1 2 3) (set-of 3 4 5) (set-of 7 8 9))
;; out = «hash-set: »
```

```
(import data/set s)

(s/union (s/set-of 1 2 3) (s/set-of 4 5 6))
;; out = «hash-set: 1 2 3 4 5 6»

(union '(1 2 3 ) '( 3 3 3 4 5))
;; out = (1 2 3 4 5)
```

要限制引入的范围，因为可能会污染默认的命名空间。

提一下 Urn 中另外一些比较好玩的模块。

更顺手的 Lua

data/bitset

math/bignum

data/graph

core/match

math/matrix

math/rational

io/term

一个经常举的例子：定义 if。
这个东西如果用函数定义的话，t 和 B 在函数调用的时候就会被提前求值了。

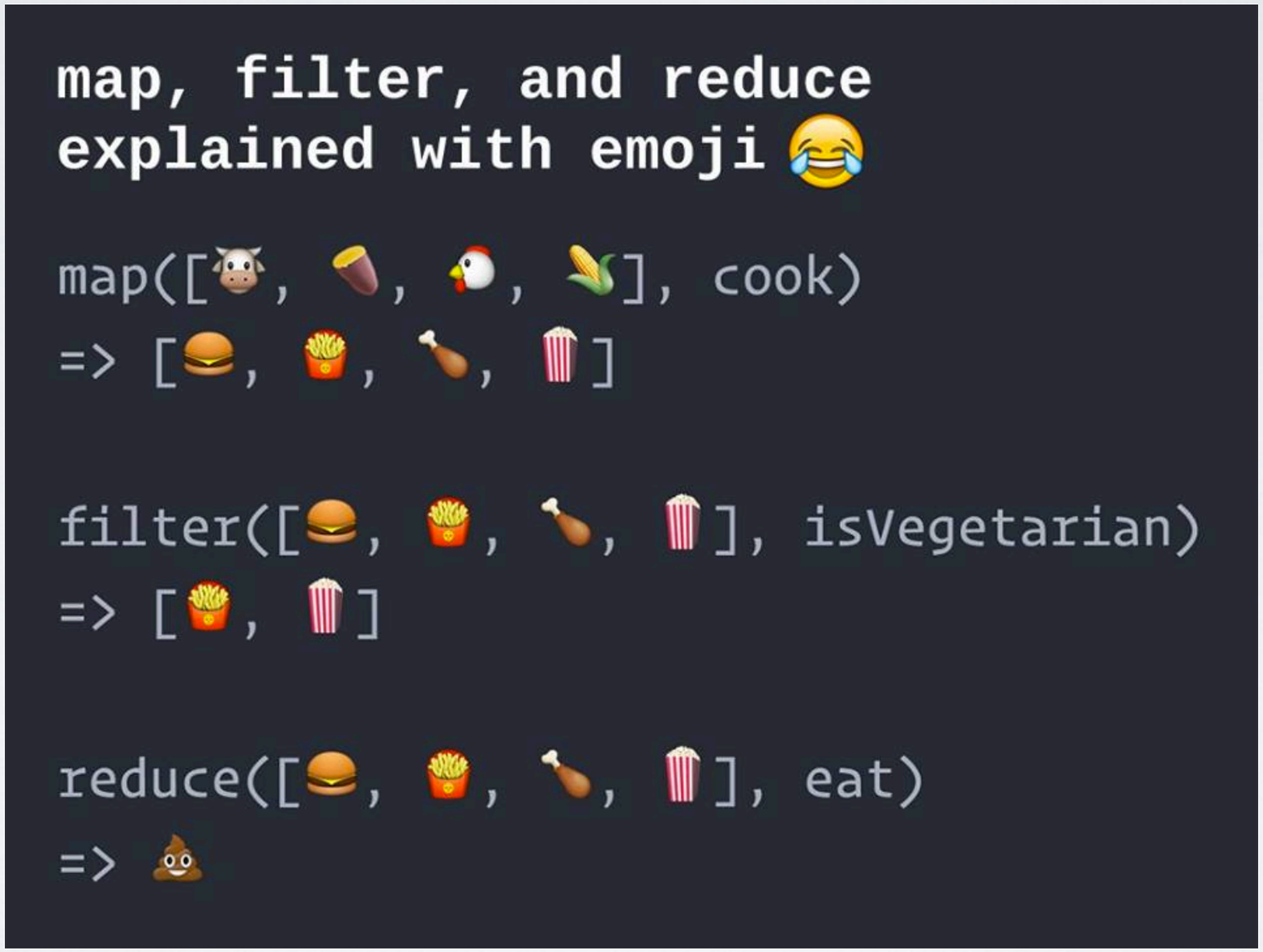
宏

`和‘的不同之处

```
(defmacro if (c t f)
  `(cond (,c ,t) (true ,f)))
```

常用的高阶函数，map/filter/reduce 等

高阶函数



filter 和 map 都可以由 reduce 实现

高阶函数

(reduce f z xs)

reduce 是比较强大的一个 “模式”，对我们写程序的时候最经常遇到的情形进行了抽象。
使我们可以脱离循环和状态，来描述程序逻辑。
这也是函数式编程最显著的特点：描述数据的模式，而非对数据的操作步骤。

高阶函数

```
(reduce
  (lambda [acc x] (+ acc x) )
  0
  '(1 2 3))

;; out = 6
```

```
> (reduce + '(1 2 3))
out = 6
```

高阶函数

```
> (map f '(1 2 3))
out = (2 4 6)
```

```
(defun my-map [f xs]
  (reduce (lambda [acc x]
            (append acc `(,(f x))))
          '()
          xs))


> (my-map f '(1 2 3))
out = (2 4 6)
```

curry 化的方法，以及链式调用。

这明显是从 Clojure 里面借鉴过来的。

高阶函数

```
> ((cut * 2 <>) 3)
out = 6
```

‘<> ;; Urn 里面这个东西叫做 “slot”，作用是在 curry 化的时候做占位符。

```
> ((cut * <> <>) 2 3)
out = 6
```

一个 list 中包含多个 slot 的情形

```
(-> '(1 2 3)
  (map succ <>)
  (map (cut * <> 2) <>))

;; out = (4 6 8)
```

链式调用

从 Clojure 里面借鉴过来的另一个好东西。

模式匹配

一个 naive 的 fibonacci 数列第 n 项，体会一下最简单的模式匹配。

```
(defun fib [n]
  (case n
    [0  1 ]
    [1  1 ]
    [?x (+ (fib (- x 1))
           (fib (- x 2)))]))
```

```
> (map fib (range :from 0 :to 6))
out = (1 1 2 3 5 8 13)
```

模式匹配

```
(defun last2 [l]
  (case l
    [()         nil]
    [(?x ?y)    `(,x ,y)]
    [(_ . ?xs)  (last2 xs)]))
```

```
> (last2 '[1 2 3 4 5])
out = (4 5)
```

对列表结构进行模式匹配

这只能说是受了 Clojure 的影响。

Urn 的 Immutable 特性没有像 Clojure 那样激进。

像 let 和函数参数 还是可变的。

Immutable

```
(define x :mutable 1)
;; x = 1
```

toplevel 上定义的东西，如果不加修饰，默认是 immutable 的。

setq! 的用途。

简单说说 immutable 的好处。

```
(set! x 2)
;; x = 2
```

```
> (define x 1)
out = 1
> (set! x 2)
[ERROR] Cannot rebind immutable definition 'x'
  => <stdin>:[1:1 .. 1:10]
 1 | (set! x 2)
   | ^^^^^^^^^^
```

与 Lua 交互

```
(define x {
    :size   20
    :range  {"name" "foo"}})
```

```
;; x.size
(print! (.> x :size))

;; x.range.name
(print! (.> x :range :name))

;; x.size = 2
(.<! x :size 2)

;; x.range.name = "bar"
(.<! x :range :name "bar")
```

与 Lua 交互

```
(define mysql (require "resty.mysql"))
```

```
(let [(mysql (require "resty.mysql"))]
     do-something)
```

```
(with [mysql (require "resty.mysql")]
     do-something)
```

```
local mysql = require "resty.mysql"
```

与 Lua 交互

```
((.> ngx :print) data)
```

```
ngx.print(data)
```

```
(define ngx-print (.> ngx :print))
(ngx-print data)
```

与 Lua 交互

(call ngx :print data)

ngx.print(data)

与 Lua 交互

```
local db = mysql:new()
db:connect{
            host = "127.0.0.1",
            port = 3306,
            database = "test",
            user = "monty",
            password = "mypass"
          }

local res, err, errno, sqlstate =
        db:query("select * from cats order by id asc")

print(res)
```

与 Lua 交互

```
(with [db (self mysql :new)]
  (self db :connect
        {
            host        "127.0.0.1"
            port        3306
            database    "test"
            user        “monty"
            password    "mypass"
        })

(destructuring-bind
  [(?res _ _)
     (list (self db :query "select * from cats order by id asc"))]

  (print! res)))
```

与 Lua 交互

```
(define cjson (require "cjson"))
(define cjson-encode (.> cjson :encode))

(defun greeting (ngx)
  (call ngx :say
    (format nil
      "HELLO-FROM-LISP: {msg}"
      :msg (cjson-encode {
             "foo" "bar"
             }))))

{
  "hello"  greeting
}
```

```
init_by_lua_block {
    lisp = require("/path/to/target/out")
}


location /lisp {
    default_type text/html;

    content_by_lua_block {
        lisp.hello (ngx);
    }
}
```

Recap

* 更顺手的 Lua

* 函数式特性

* 与 OpenResty 的交互

利用函数式编程，快速实现正确的功能:D