[**OpenResty学习指南（二）**](https://www.cnblogs.com/luozhiyun/p/12284264.html)

**数据结构table**[**#**](https://www.cnblogs.com/luozhiyun/p/12284264.html#2350539921)

table并没有区分开数组、哈希、集合等概念，而是揉在了一起。

Copy

local color = {first = "red", "blue", third = "green", "yellow"}

print(color["first"]) --> output: red

print(color[1]) --> output: blue

print(color["third"]) --> output: green

print(color[2]) --> output: yellow

print(color[3]) --> output: nil

**table 库函数[#](https://www.cnblogs.com/luozhiyun/p/12284264.html" \l "2413540602)**

**获取元素个数[#](https://www.cnblogs.com/luozhiyun/p/12284264.html" \l "3929046758)**

对于序列，用table.getn 或者一元操作符 # ，就可以正确返回元素的个数。

Copy

$ resty -e 'local t = { 1, 2, 3 }

print(table.getn(t)) ' # 返回3

不是序列的 table，就无法返回正确的值。

Copy

$ resty -e 'local t = { 1, a = 2 }

print(#t) ' #返回1

所以不要使用函数 table.getn 和一元操作符 # 。

我们可以使用 table.nkeys 来获取 table 长度，返回的是 table 的元素个数，包括数组和哈希部分的元素。

Copy

local nkeys = require "table.nkeys"

print(nkeys({})) -- 0

print(nkeys({ "a", nil, "b" })) -- 2

print(nkeys({ dog = 3, cat = 4, bird = nil })) -- 2

print(nkeys({ "a", dog = 3, cat = 4 })) -- 3

**删除指定元素[#](https://www.cnblogs.com/luozhiyun/p/12284264.html" \l "3432931437)**

第二个我们来看table.remove 函数，它的作用是在 table 中根据下标来删除元素，也就是说只能删除 table 中数组部分的元素

Copy

$ resty -e 'local color = {first = "red", "blue", third = "green", "yellow"}

table.remove(color, 1)

for k, v in pairs(color) do

print(v)

end'

这段代码会把下标为 1 的 blue 删除掉。

如果要删除哈希部分，把 key 对应的 value 设置为 nil 即可。

**元素拼接函数**[**#**](https://www.cnblogs.com/luozhiyun/p/12284264.html#464021732)

table.concat 以按照下标，把 table 中的元素拼接起来。只能拼接数组部分

Copy

$ resty -e 'local color = {first = "red", "blue", third = "green", "yellow"}

print(table.concat(color, ", "))'

使用table.concat函数后，它输出的是 blue, yellow，哈希的部分被跳过了。

**插入一个元素**[**#**](https://www.cnblogs.com/luozhiyun/p/12284264.html#3615477504)

table.insert 函数，可以下标插入一个新的元素，自然，影响的还是 table 的数组部分。

Copy

$ resty -e 'local color = {first = "red", "blue", third = "green", "yellow"}

table.insert(color, 1, "orange")

print(color[1])

'

color 的第一个元素变为了 orange。当然，你也可以不指定下标，这样就会默认插入队尾。

**优化**[**#**](https://www.cnblogs.com/luozhiyun/p/12284264.html#2690049427)

**预先生成数组[#](https://www.cnblogs.com/luozhiyun/p/12284264.html" \l "3093532460)**

预先生成一个指定大小的数组，避免每次新增和删除数组元素的时候，都会涉及到数组的空间分配、resize 和 rehash。  
如：

Copy

local new\_tab = require "table.new"

local t = new\_tab(100, 0)

for i = 1, 100 do

t[i] = i

end

**循环使用单个 table**[**#**](https://www.cnblogs.com/luozhiyun/p/12284264.html#1343298999)

table.clear 函数它会把数组中的所有数据清空，但数组的大小不会变。  
如下：

Copy

local local\_plugins = {}

function load()

core.table.clear(local\_plugins)

local local\_conf = core.config.local\_conf()

local plugin\_names = local\_conf.plugins

local processed = {}

for \_, name in ipairs(plugin\_names) do

if processed[name] == nil then

processed[name] = true

insert\_tab(local\_plugins, name)

end

end

return local\_plugins

local\_plugins 这个数组，是 plugin 这个模块的 top level 变量。在 load 这个加载插件函数的开始位置， table 就会被清空，然后根据当前的情况生成新的插件列表。

**table 池[#](https://www.cnblogs.com/luozhiyun/p/12284264.html" \l "4116898581)**

lua-tablepool，可以用缓存池的方式来保存多个 table，以便随用随取。

Copy

local tablepool = require "tablepool"

local tablepool\_fetch = tablepool.fetch

local tablepool\_release = tablepool.release

local pool\_name = "some\_tag"

local function do\_sth()

local t = tablepool\_fetch(pool\_name, 10, 0)

-- -- using t for some purposes

tablepool\_release(pool\_name, t)

end

**缓存[#](https://www.cnblogs.com/luozhiyun/p/12284264.html" \l "2469235596)**

OpenResty 中有两个缓存的组件：shared dict 缓存和 lru 缓存。前者只能缓存字符串对象，缓存的数据有且只有一份，每一个 worker 都可以进行访问，所以常用于 worker 之间的数据通信。后者则可以缓存所有的 Lua 对象，但只能在单个 worker 进程内访问，有多少个 worker，就会有多少份缓存数据。





**shared dict[#](https://www.cnblogs.com/luozhiyun/p/12284264.html" \l "4068091247)**

Copy

$ resty --shdict='dogs 1m' -e 'local dict = ngx.shared.dogs

dict:set("Tom", 56)

print(dict:get("Tom"))'

需要事先在 Nginx 的配置文件中，声明一个内存区 dogs，然后在 Lua 代码中才可以使用。

共享字典中还有一个 get\_stale 的读取数据的方法，相比 get 方法，多了一个**过期数据**的返回值：

Copy

resty --shdict='dogs 1m' -e 'local dict = ngx.shared.dogs

dict:set("Tom", 56, 0.01)

ngx.sleep(0.02)

local val, flags, stale = dict:get\_stale("Tom")

print(val)'

在上面的这个示例中，数据只在共享字典中缓存了 0.01 秒，在 set 后的 0.02 秒后，数据就已经超时了。这时候，通过 get 接口就不会获取到数据了，但通过 get\_stale 还可能获取到过期的数据。

因为，在 shared dict 中存放的是缓存数据，即使缓存数据过期了，也并不意味着源数据就一定有更新。

**lru 缓存[#](https://www.cnblogs.com/luozhiyun/p/12284264.html" \l "2798050286)**

lru 缓存的接口只有 5 个：new、set、get、delete 和 flush\_all。

如何使用：

Copy

resty -e 'local lrucache = require "resty.lrucache"

local cache, err = lrucache.new(200)

cache:set("dog", 32, 0.01)

ngx.sleep(0.02)

local data, stale\_data = cache:get("dog")

print(stale\_data)'

可以看到，在 lru 缓存中， get 接口的第二个返回值直接就是 stale\_data，而不是像 shared dict 那样分为了 get 和 get\_stale 两个不同的 API。

**lua-resty-mlcache[#](https://www.cnblogs.com/luozhiyun/p/12284264.html" \l "480318787)**

lua-resty-mlcache用 shared dict 和 lua-resty-lrucache ，实现了多层缓存机制。

初始化：

Copy

local mlcache = require "resty.mlcache"

local cache, err = mlcache.new("cache\_name", "cache\_dict", {

lru\_size = 500, -- size of the L1 (Lua VM) cache

ttl = 3600, -- 1h ttl for hits

neg\_ttl = 30, -- 30s ttl for misses

})

if not cache then

error("failed to create mlcache: " .. err)

end

这段代码的开头引入了 mlcache 库，并设置了初始化的参数。第一个参数是缓冲名，第二个参数是字典名，第三个参数是个字典，里面是12个选填参数。

使用：

Copy

local function fetch\_user(id)

return db:query\_user(id)

end

local id = 123

local user , err = cache:get(id , nil , fetch\_user , id)

if err then

ngx.log(ngx.ERR , "failed to fetch user: ", err)

return

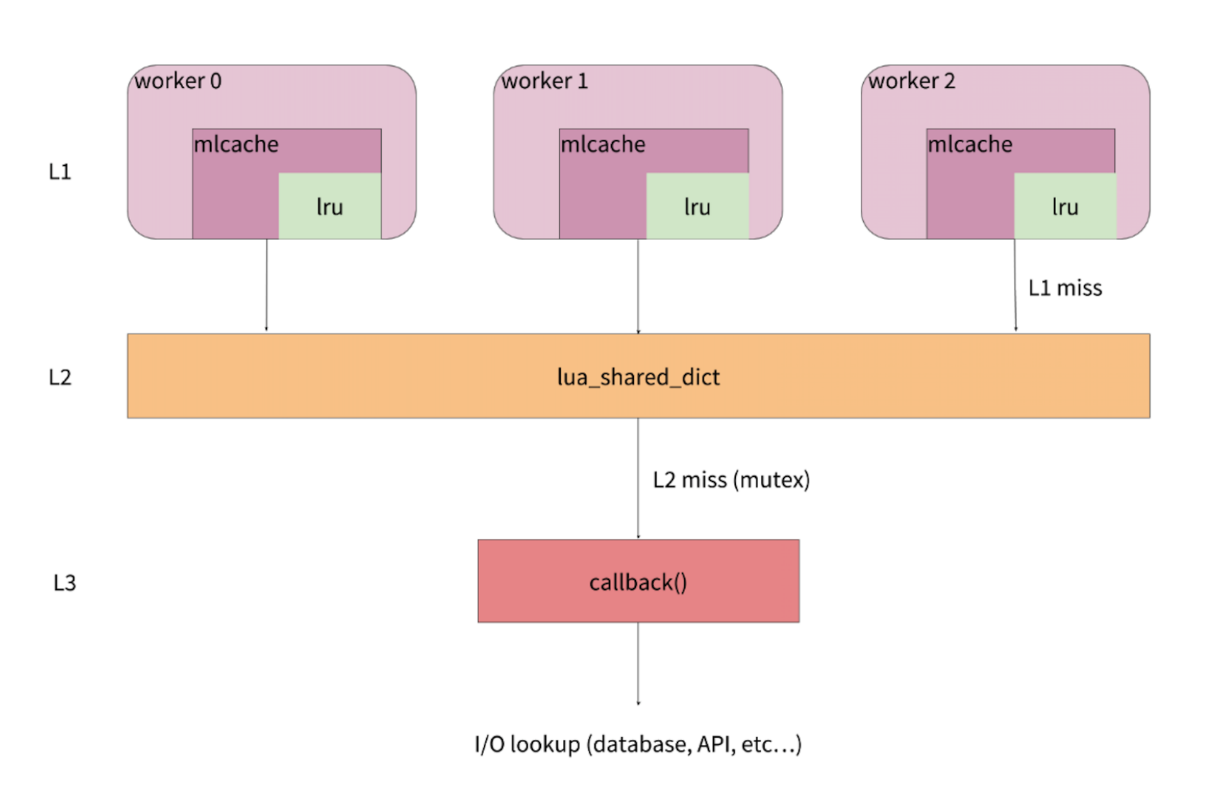
end

if user then

print(user.id) -- 123

end

下面再看看这个库的架构与实现：



L1 缓存就是 lua-resty-lrucache。每一个 worker 中都有自己独立的一份，有 N 个 worker，就会有 N 份数据，自然也就存在数据冗余。

L2 缓存是 shared dict。所有的 worker 共用一份缓存数据，在 L1 缓存没有命中的情况下，就会来查询 L2 缓存。

L3 则是在 L2 缓存也没有命中的情况下，需要执行回调函数去外部数据库等数据源查询后，再缓存到 L2 中。

整体而言，从请求的角度来看：

1. 首先会去查询 worker 内的 L1 缓存，如果 L1 命中就直接返回。
2. 如果 L1 没有命中或者缓存失效，就会去查询 worker 间的 L2 缓存。如果 L2 命中就返回，并把结果缓存到 L1 中。
3. 如果 L2 也没有命中或者缓存失效，就会调用回调函数，从数据源中查到数据，并写入到 L2 缓存中，这也就是 L3 数据层的功能。

需要做数据序列化的情况：

Copy

local mlcache = require "resty.mlcache"

local cache, err = mlcache.new("my\_mlcache", "cache\_shm", {

l1\_serializer = function(i)

return i + 2

end,

})

local function callback()

return 123456

end

local data = assert(cache:get("number", nil, callback))

assert(data == 123458)

在 new 中，我们设置的 l1\_serializer 函数会在设置 L1 缓存前，把传入的数字加 2，也就是变成 123458。