

# Lua 迭代器

## 1. 迭代器与 Closure

在 Lua 中，**迭代器通常为函数**，每调用一次函数，即返回集合中的“下一个”元素。每个迭代器都需要在每次成功调用之间保持一些状态，这样才能知道它所在的位置和下一次遍历时的位置。从这一点看，Lua 中 closure 机制为此问题提供了语言上的保障，见如下示例：



```
1 function values(t)
2     local i = 0
3     return function()
4         i = i + 1
5         return t[i]
6     end
7 end
8 t = {10, 20, 30}
9 it = values(t)
10 while true do
11     local element = it()
12     if element == nil then
13         break
14     end
15     print(element)
16 end
17 --另外一种基于 foreach 的调用方式(泛型 for)
18 t2 = {15, 25, 35}
19 for element in values(t2) do
20     print(element)
21 end
22 --输出结果为:
23 --10
24 --20
25 --30
26 --15
27 --25
28 --35
```



## 2. 泛型 for 的语义：

上面示例中的迭代器有一个明显的缺点，即每次循环时都需要创建一个新的 closure 变量，否则第一次迭代成功后，再将该 closure 用于新的 for 循环时将会直接退出。

这里我们还是先详细的讲解一下 Lua 中泛型(for)的机制，之后再给出一个无状态迭代器的例子，以便于我们的理解。如果我们的迭代器实现为无状态迭代器，那么就不必为每一次的泛型(for)都重新声明一个新的迭代器变量了。

泛型(for)的语法如下：


```
for <var-list> in <exp-list> do
  <body>
end
```

为了便于理解，由于我们在实际应用中 **<exp-list>** 通常只是包含一个表达式(**expr**)，因此简单起见，这里的说明将只是包含一个表达式，而不是表达式列表。现在我们先给出表达式的原型和实例，如：

```
1 function ipairs2(a)
2   return iter,a,0
3 end
```


该函数返回 3 个值，第一个为实际的迭代器函数变量，第二个是一个恒定对象，这里我们可以理解为待遍历的容器，第三个变量是在调用 **iter()** 函数时为其传入的初始值。

下面我们再看一下 **iter()** 函数的实现，如：



```
1 local function iter(a, i)
2   i = i + 1
3   local v = a[i]
4   if v then
5     return i, v
6   else
7     return nil, nil
8   end
9 end
```

在迭代器函数 **iter()** 中返回了两个值，分别对应于 **table** 的 **key** 和 **value**，其中 **key**(返回的 **i**) 如果为 **nil**，泛型(**for**)将会认为本次迭代已经结束。下面我们先看一下实际用例，如：



```
1 function ipairs2(a)
2   return iter,a,0
3 end
4
5
6 local function iter(a, i)
7   i = i + 1
8   local v = a[i]
9   if v then
10    return i, v
11  else
12    return nil, nil
13  end
14 end
15
```

```

16 a = {"one", "two", "three"}
17 for k,v in ipairs2(a) do
18     print(k, v)
19 end
20 --输出结果为:
21 --1      one
22 --2      two
23 --3      three

```



这个例子中的泛型(**for**)写法可以展开为下面的基于 **while** 循环的方式，如：



```

1 local function iter(a, i)
2     i = i + 1
3     local v = a[i]
4     if v then
5         return i, v
6     else
7         return nil, nil
8     end
9 end
10
11 function ipairs2(a)
12     return iter,a,0
13 end
14
15 a = {"one", "two", "three"}
16 do
17     local _it,_s,_var = ipairs2(a)
18     while true do
19         local var_1,var_2 = _it(_s,_var)
20         _var = var_1
21         if _var == nil then --注意，这里只判断迭代器函数返回的第一个是否为 nil。
22             break
23         end
24         print(var_1,var_2)
25     end
26 end
27 --输出结果同上。

```



### 3. 无状态迭代器的例子：

这里的示例将实现遍历链表的迭代器。



```
1 local function getnext(list, node)  --迭代器函数。
2     if not node then
3         return list
4     else
5         return node.next
6     end
7 end
8
9 function traverse(list)  --泛型(for)的 expression
10     return getnext,list,nil
11 end
12
13 --初始化链表中的数据。
14 list = nil
15 for line in io.lines() do
16     line = { val = line, next = list}
17 end
18
19 --以泛型(for)的形式遍历链表。
20 for node in traverse(list) do
21     print(node.val)
22 end
```



这里使用的技巧是将链表的头结点作为恒定状态(`traverse` 返回的第二个值)，而将当前节点作为控制变量。第一次调用迭代器函数 `getnext()` 时，`node` 为 `nil`，因此函数返回 `list` 作为第一个结点。在后续调用中 `node` 不再为 `nil` 了，所以迭代器返回 `node.next`，直到返回链表尾部的 `nil` 结点，此时泛型(`for`)将判断出迭代器的遍历已经结束。

最后需要说明的是，`traverse()` 函数和 `list` 变量可以反复的调用而无需再创建新的 `closure` 变量了。这主要是因为迭代器函数(`getnext`)实现为无状态迭代器。

### 4. 具有复杂状态的迭代器：

在上面介绍的迭代器实现中，迭代器需要保存许多状态，可是泛型(`for`)却只提供了恒定状态和控制变量用于状态的保存。一个最简单的办法是使用 `closure`。当然我们还以将所有的信息封装到一个 `table` 中，并作为恒定状态对象传递给迭代器。虽说恒定状态变量本身是恒定的，即在迭代过程中不会换成其它对象，但是该对象所包含的数据是否变化则完全取决于迭代器的实现。就目前而言，由于 `table` 类型的恒定对象已经包含了所有迭代器依赖的信息，那么迭代器就完全可以忽略泛型(`for`)提供的第二个参数。下面我们就给出一个这样的实例，见如下代码：



```
1 local iterator
2 function allwords()
3     local state { line = io.read(), pos = 1 }
4     return iterator, state
5 end
6 --iterator 函数将是真正的迭代器
7 function iterator(state)
8     while state.line do
9         local s,e = string.find(state.line,"%w+",state.pos)
10        if s then
11            state.pos = e + 1
12            return string.sub(state.line,s,e)
13        else
14            state.line = io.read()
15            state.pos = 1
16        end
17    end
18    return nil
19 end
```

