字符串标准库

Lua语言处理字符串的能力有限，只能创建、比较、获取字符串长度（#）。Lua语言处理字符串的完整能力来自其字符串标准库。

字符串标准库，默认处理1byte字符，比如ASCII，对Unicode并不适用。

字符串标准库中的函数示例：

string.len(s) <--> #s

string.rep(“abc”, 3) --> abcabcabc //可以用于生成一个指定长度的字符串，

string.rep(“a”, 2^20). 方便测试

string.reverse(“a long”) --> gnol a

string.lower(“ A Long Line”) --> a long line

string.upper(“A Long Line”) --> A LONG LINE

一种典型应用，忽略字母大小写差异比较两个字符串：

string.lower(a) < string.lower(a)

string.sub(s, i, j) 提取索引i到j之间的字符串，索引从1开始。可以为负数，从-1开始，表示倒数第一个字符。

string.sub(s, 1, j) 提取前j个字符。

string.sub(s, j, -1) 提取第j个字符开始的后缀。

string.sub(s, 2, -2) 掐头去尾

string.char(97) --> a

string.char(99, 100, 101) --> cde

string.byte(“abc”) --> 97

string.byte(“abc”, 2) --> 98

string.byte(“abc”, -1) --> 99

string.byte(“abc”, 1, 2) --> 97 98

string.byte(s, 1, -1) --> 由s中所有字符代码组成的表（由于Lua限制了栈大小，这限制了函数的返回值的最大个数，默认最大一百万个。因此，这个技巧不能用于大小超过1MB的字符串）

string.format(“x = %d”, 8) --> x = 8

string.format 用于字符串格式化、将数值输出为字符串。Lua语言是通过调用C语言标准库来完成实际工作的，与printf一样。

冒号操作符：

string.sub(s, i, j) <--> s:sub(i, j)

string.upper(s) <--> s:upper( )

string.find(“hello world”, “wor”) --> 7 9

string.find(“hello world”, “war”) --> nil

string.gsub(“hello world”, “l”, “.”) --> he..o wor.d 3

string.gsub(“hello world”, “a”, “.”) --> hello world 0

Unicode 编码

UTF-8使用变长的多个字节来编码一个字符。从第一个字符的数值范围可以判断这个字符占用的字节数。

第一个字节：<128，占一个字节，与ASCII兼容。

第一个字节：194<=x<=223，占两个字节，后续字节的数值大小满足：128<=x<=191

第一个字节：224<=x<=239，占三个字节，后续字节的数值大小满足：128<=x<=191

第一个字节：240<=x<=244，占四个字节，后续字节的数值大小满足：128<=x<=191

对于ASCII码，把它当做ASCII或者UTF-8都没问题。

对于其它UTF-8编码字符，Lua语言中对字符串的操作和函数也能用，不过可能得到并非你所想的结果。

Lua5.3引入了处理UTF-8字符的标准库：utf8。

utf8.len(“嘿嘿”) --> 2

utf8.len(“ab\x93”) --> nil 3

utf8.char(114,233,115,117,109,233) --> résumé

a = utf8.char(114,233,115,117,109,233)

utf8.codepoint(a) --> 109 233

//与string.char和string.byte类似，注意utf8.codepoint中的索引，依然是字节位置，如果想使用字符位置，要用utf8.offset函数

s = ...

utf8.codepoint(s, utf8.offset(s,5)) //获取第5个字符的字节索引

utf8.offset中的索引可以是负数

for i,c in utf8.codes(“Ação”) do //遍历UTF-8字符串中的每个字符，返回字节索引和编码值

print(i, c)

end

--> 1 65

--> 2 231

--> 3 227

--> 4 111

表

Table是Lua语言唯一的、强大的数据结构。Lua使用表来表示package和对象。调用math.sin时，可看成是调用math库中的sin函数，对于Lua来说，其实际含义是：以字符串“sin”为键，检索表math。

Lua中的表，本质是一种辅助数组，associative array，它可以使用任意类型的值作为索引，nil除外。

表，构造器表达式，{ }

a = { }

a[“x”] = 10

b = a

b[“x”] = 20

a[“x”] --> 20

a[10] --> nil

a = nil

b = nil //没有指向表的引用了，垃圾收集器会回收内存

上例中，a[“x”] = nil，可以删除表中元素，与Lua的全局变量一样，这不是巧合，因为Lua就是使用表来存储全局变量的。

表索引

a.name <--> a[“name”]

a = { }

a.x = 10

a[“x”] = a.x = 10

a.y --> nil

对Lua语言而言，这两种形式等价，可混用；对于阅读程序的人而言，这两种形式代表了不同的意图，点分形式说明了表是被当做结构体或对象使用的。

a = { }

x = “y”

a[x] = 10

a.x --> nil

a.y --> 10

由于可以使用任意类型的索引，所以索引表时会遇到相等性比较方面的微妙问题。

i = 10; j = “10”; k = “+10”

a.i a.j a.k 是不一样的。

整型和浮点型作为表索引时，则不区分子类型，只按数值是否相等去判断。

a = { }

a[2.0] = 10

a[2] --> 10

Lua在实现中，当number被用作表索引时，任何能转换成integer的，都要被转成integer存储，不能转换就算了，就按float来，当做检索时，如果传入的float，可以转换成integer，也要转成integer去检索。这样integer的比较速度比单纯的number之间的比较速度要快。这时Lua的一个优化点。

表构造器 Table constructor

表构造器，用来创建和初始化表，是Lua语言中独有的、灵活的、有用的机制。

空构造器：{ }

列表构造器：days = { “Sunday”, “Monday”, “Tuesday” }

days[1] --> Sunday

记录构造器：a = { x=10, y=20}

a.x --> 10

w = { x=0, y=0 }

x = {“a”, “b”}

w[1] = “another field”

x.f = w

w[“x”] --> 0

w[1] --> “another field”

x.f[1] --> “another field”

两种构造器混用：

polyline = { color=”blue”, thickness=2, npoints=4,

{ x=0, y=0},

{ x=-10, y=0},

{ x=-10, y=1},

{ x=0, y=1}

}

上述示例展示了嵌套表、嵌套构造器。

polyline[2].x --> -10

这两种构造器有自己的局限，使用它们时，不能用负数索引初始化表元素、也不能使用不符合标志符规范（Identifier）的东西作为索引。

通用构造器：

opnames = { [“+”] = “add”, [“-“] = “sub”, [20] = “-“, [-1] = “+” }。

{ x=0, y=0 } <--> { [“x”]=0, [“y”]=0 }

{ “r”, “g”, “b” } <--> { [1]=”r”, [2]=”g”, [3]=”b” }

构造器中，最后一个元素的逗号在不在都行，跟C语言中的数组初始化一样。这使得开发人员在编写构造器时，不需要对最后一个元素进行特殊处理。

最后，表构造器中的逗号，可以用分号代替，这主要是为了兼容旧版的Lua，但基本不用。

数组、列表、序列

a = { }

for i=1, 10 do

a[i] = io.read( )

end

Lua中#可以用来计算数组/列表的长度，计算方法是取最后一个不为nil的索引值，这种方法只有在列表中不存在空洞(hole)时才有效，我们称这种不含nil的数组为序列（sequence）。

for i=1, #a do

print(a[i])

end

将长度操作符用于存在空洞列表的行为是Lua语言最具争议的内容之一。很多人建议，在用#操作存在空洞的列表时直接抛出异常，也有人建议，扩展长度操作符的语义，另一种常见的建议是，让#操作符返回列表中全部元素的数量。程序中大多数列表，其实是序列，在多数情况下，使用长度操作符#是安全的。当确实需要处理存在空洞的列表时，可以将列表的长度显示地保存起来。

遍历表 5.4 P50

对表而言，用pairs迭代器遍历表中的键值对。出现顺序随机。

对列表而言，可以用ipairs迭代器。按顺序。

对序列而言，可以用for循环，for i=1, #t do ......

安全访问

如果想确认某个库中的某个函数是否存在，得先确认这个库确实存在。

if lib and lib.foo then ... 如果直接用if lib.foo then ... 那么当lib不存在时，会抛出异常，而不是返回nil。

当表嵌套较深时，这种写法就很冗长，例如：

zip = company and company.director and company.director.address and

company.director.address.zipcode

该写法冗长且低效，对于访问一个有效元素，它需要进行6次访问，而不是3次。

对于这种情景，C#提供了安全访问操作符：？.

zip = company ?. director ?. address ?. zipcode

Lua语言中，我们可以用以下方法模拟安全访问操作符：

E = { }

zip = ( ( ( company or E ).director or E ).address or E ).zipcode

表中的每个字段名都只被使用了一次，保证了尽可能少的对表进行访问（3次）。同时，还避免了向语言中引入新的操作符。虽然可读性和效率还是差了点，但Lua是为精简而设计的，增加操作符就意味着增加处理代码。它是一种足够好的替代方案。

表标准库

对于数组/列表和序列：

t = {10, 20, 30}

table.insert(t, 1, 15) --> {15, 10, 20, 30} //索引值在1和#t+1 之间，否则会报错

table.insert(t, 40) --> 在t的最后插入一个元素

table.remove(2) //删除索引为2处的元素，并将元素向前移动，填充空洞。

table.remove( ) //删除最后一个元素

表标准库，更应该称为：列表库或序列库。可以用上述两个函数轻松实现栈、队列、双端队列。一些操作会涉及元素的移动，而table标准库中的函数使用C语言完成，所以涉及循环的性能开销也并不是太昂贵。

Lua5.3引入了一个更通用的函数 table.move(a, f, e, t)，调用它可以将表a中从f到e的元素移动到位置t上。

table.move(a, 1, #a, 2)

a[1] = newElement //在a的开头插入一个元素

table.move(a, 2, #a, 1)

a[#a] = nil //删除a中的第一个元素

table.move(a, 1, #a, 1, { } ) --> 将a拷贝给{ }，并返回它（引用）

table.move(a, 1, #a, #b+1, b) --> 将a的所有元素复制到b的末尾，并返回b（引用）