

Lua 入门

Lua 是一个小巧的脚本语言。是巴西里约热内卢天主教大学（Pontifical Catholic University of Rio de Janeiro）里的一个研究小组，由 Roberto Ierusalimschy、Waldemar Celes 和 Luiz Henrique de Figueiredo 所组成并于 1993 年开发。其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。Lua 由标准 C 编写而成，几乎在所有操作系统和平台上都可以编译、运行。Lua 并没有提供强大的库，这是由它的定位决定的。所以 Lua 不适合作为开发独立应用程序的语言。Lua 有一个同时进行的 JIT 项目，提供在特定平台上的即时编译功能。

Lua 脚本可以很容易的被 C/C++ 代码调用，也可以反过来调用 C/C++ 的函数，这使得 Lua 在应用程序中可以被广泛应用。不仅仅作为扩展脚本，也可以作为普通的配置文件，代替 XML、ini 等文件格式，并且更容易理解和维护。标准 Lua 5.1 解释器由标准 C 编写而成，代码简洁优美，几乎在所有操作系统和平台上都可以编译和运行；一个完整的标准 Lua 5.1 解释器不足 200KB。而本书推荐使用的 LuaJIT 2 的代码大小也只有不足 500KB，同时也支持大部分常见的体系结构。在目前所有脚本语言引擎中，LuaJIT 2 实现的速度应该算是最快的之一。这一切都决定了 Lua 是作为嵌入式脚本的最佳选择。

Lua 语言的各个版本是不相兼容的。因此本书只介绍 Lua 5.1 语言，这是为标准 Lua 5.1 解释器和 LuaJIT 2 所共同支持的。LuaJIT 支持的对 Lua 5.1 向后兼容的 Lua 5.2 和 Lua 5.3 的特性，我们也会在方便的时候予以介绍。

Lua 简介

这一章我们简要地介绍 **Lua** 语言的基础知识，特别地，我们会有意将讨论放置于 **OpenResty** 的上下文中。同时，我们并不会回避 **LuaJIT** 独有的新特性；当然，在遇到这样的独有特性时，我们都会予以说明。我们会关注各个语言结构和标准库函数对性能的潜在影响。在讨论性能相关的问题时，我们只会关心 **LuaJIT** 实现。

Lua 是什么？

1993 年在巴西里约热内卢天主教大学(Pontifical Catholic University of Rio de Janeiro in Brazil)诞生了一门编程语言，发明者是该校的三位研究人员，他们给这门语言取了个浪漫的名字——**Lua**，在葡萄牙语里代表美丽的月亮。事实证明她没有糟蹋这个优美的单词，**Lua** 语言正如它名字所预示的那样成长为一门简洁、优雅且富有乐趣的语言。

Lua 从一开始就是作为一门方便嵌入(其它应用程序)并可扩展的轻量级脚本语言来设计的，因此她一直遵从着简单、小巧、可移植、快速的原则，官方实现完全采用 ANSI C 编写，能以 C 程序库的形式嵌入到宿主程序中。**LuaJIT 2** 和标准 **Lua 5.1** 解释器采用的是著名的 MIT 许可协议。正由于上述特点，所以 **Lua** 在游戏开发、机器人控制、分布式应用、图像处理、生物信息学等各种各样的领域中得到了越来越广泛的应用。其中尤以游戏开发为最，许多著名的游戏，比如 **Escape from Monkey Island**、**World of Warcraft**、大话西游，都采用了 **Lua** 来配合引擎完成数据描述、配置管理和逻辑控制等任务。即使像 **Redis** 这样中性的内存键值数据库也提供了内嵌用户 **Lua** 脚本的官方支持。

作为一门过程型动态语言，**Lua** 有着如下的特性：

1. 变量名没有类型，值才有类型，变量名在运行时可与任何类型的值绑定；
2. 语言只提供唯一一种数据结构，称为表(table)，它混合了数组、哈希，可以用任何类型的值作为 key 和 value。提供了一致且富有表达力的表构造语法，使得 **Lua** 很适合描述复杂的数据；
3. 函数是一等类型，支持匿名函数和正则尾递归(proper tail recursion)；
4. 支持词法定界(lexical scoping)和闭包(closure)；
5. 提供 **thread** 类型和结构化的协程(coroutine)机制，在此基础上可方便实现协作式多任务；
6. 运行期能编译字符串形式的程序文本并载入虚拟机执行；
7. 通过元表(metatable)和元方法(metamethod)提供动态元机制(dynamic meta-mechanism)，从而允许程序运行时根据需要改变或扩充语法设施的内定语义；
8. 能方便地利用表和动态元机制实现基于原型(prototype-based)的面向对象模型；
9. 从 5.1 版开始提供了完善的模块机制，从而更好地支持开发大型的应用程序；

Lua 的语法类似 PASCAL 和 Modula 但更加简洁，所有的语法产生式规则(EBNF)不过才 60 几个。熟悉 C 和 PASCAL 的程序员一般只需半个小时便可将其完全掌握。而在语义上 Lua 则与 Scheme 极为相似，她们完全共享上述的 1、3、4、6 点特性，Scheme 的 continuation 与协程也基本相同只是自由度更高。最引人注目的是，两种语言都只提供唯一一种数据结构：Lua 的表和 Scheme 的列表(list)。正因为如此，有人甚至称 Lua 为“只用表的 Scheme”。

Lua 和 LuaJIT 的区别

Lua 非常高效，它运行得比许多其它脚本(如 Perl、Python、Ruby)都快，这点在第三方的独立测评中得到了证实。尽管如此，仍然会有人不满足，他们总觉得“嗯，还不够快!”。LuaJIT 就是一个为了再榨出一些速度的尝试，它利用即时编译（Just-in Time）技术把 Lua 代码编译成本地机器码后交由 CPU 直接执行。LuaJIT 2 的测评报告表明，在数值运算、循环与函数调用、协程切换、字符串操作等许多方面它的加速效果都很显著。凭借着 FFI 特性，LuaJIT 2 在那些需要频繁地调用外部 C/C++ 代码的场景，也要比标准 Lua 解释器快很多。目前 LuaJIT 2 已经支持包括 i386、x86_64、ARM、PowerPC 以及 MIPS 等多种不同的体系结构。

LuaJIT 是采用 C 和汇编语言编写的 Lua 解释器与即时编译器。LuaJIT 被设计成全兼容标准的 Lua 5.1 语言，同时可选地支持 Lua 5.2 和 Lua 5.3 中的一些不破坏向后兼容性的有用特性。因此，标准 Lua 语言的代码可以不加修改地运行在 LuaJIT 之上。LuaJIT 和标准 Lua 解释器的一大区别是，LuaJIT 的执行速度，即使是其汇编编写的 Lua 解释器，也要比标准 Lua 5.1 解释器快很多，可以说是一个高效的 Lua 实现。另一个区别是，LuaJIT 支持比标准 Lua 5.1 语言更多的基本原语和特性，因此功能上也要更加强大。

若无特殊说明，我们接下来的章节都是基于 LuaJIT 进行介绍的。

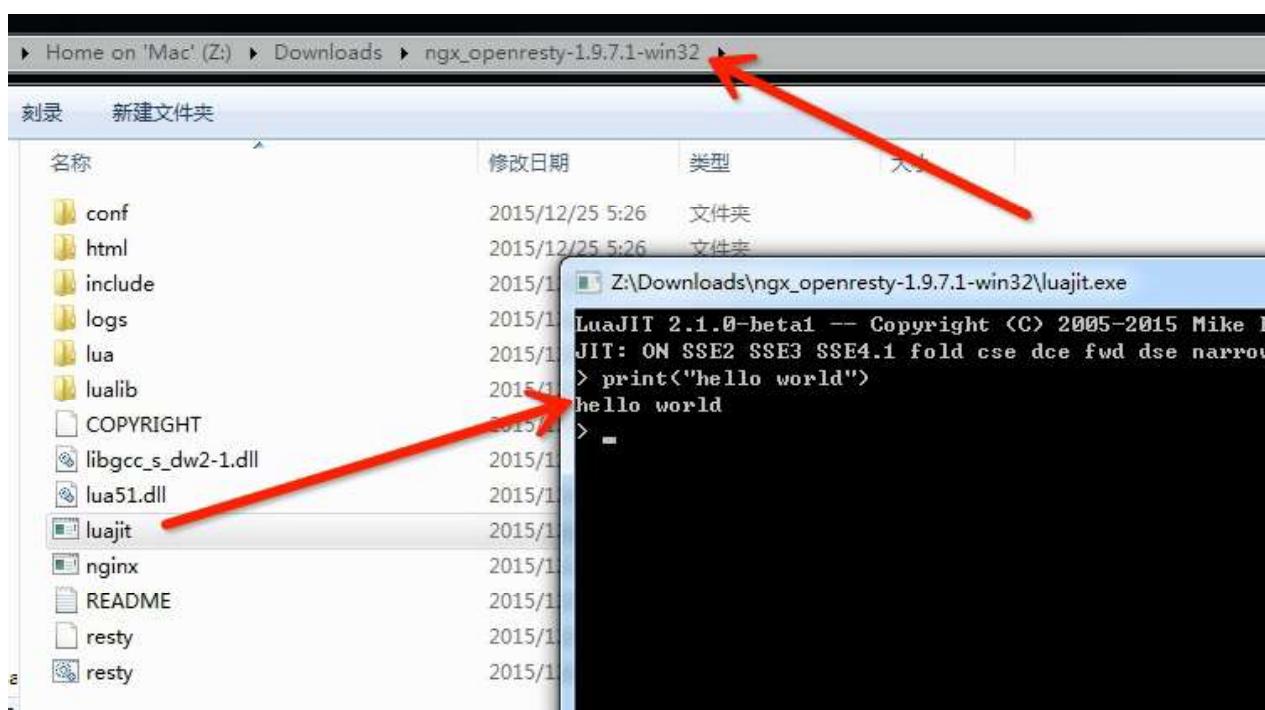
- Lua 官网链接：<http://www.lua.org>
- LuaJIT 官网链接：<http://luajit.org>

Lua 环境搭建

在 Windows 上搭建环境

从 1.9.3.2 版本开始，OpenResty 正式对外同时公布维护了 Windows 版本，其中直接包含了编译好的最新版本 LuaJIT。由于 Windows 操作系统自身相对良好的二进制兼容性，使用者只需要下载、解压两个步骤即可。

打开 <http://openresty.org>，选择左侧的 `Download` 连接，这时候我们就可以下载最新版本的 OpenResty 版本（例如笔者写书时的最新版本：[ngx_openresty-1.9.7.1-win32.zip](#)）。下载本地成功后，执行解压缩，就能看到下图所示目录结构：



双击图中的 `LuajIT.exe`，即可进入命令行模式，在这里我们就可以直接完成简单的 Lua 语法交互了。

在 Linux、Mac OS X 上搭建环境

到 `LuaJIT` 官网 <http://luajit.org/download.html>，查看当前最新开发版本，例如笔者写书时的最新版本：[http://luajit.org/download/LuaJIT-2.1.0-beta1.tar.gz](#)。

```
# wget http://luajit.org/download/LuaJIT-2.1.0-beta1.tar.gz
# tar -xvf LuaJIT-2.1.0-beta1.tar.gz
# cd LuaJIT-2.1.0-beta1
# make
# sudo make install
```

大家都知道，在不同平台，可能都有不同的安装工具来简化我们的安装。为什么我们这给大家推荐的是源码这么原始的方式？笔者为了偷懒么？答案：是的。当然还有另外一个原因，就是我们安装的是 **LuaJIT 2.1** 版本。

从实际应用性能表现来看，**LuaJIT 2.1** 虽然目前还是 **beta** 版本，但是生产运行稳定性已经很不错，并且在运行效率上要比 **LuaJIT 2.0** 好很多（大家可自行爬文了解一下），所以作为 **OpenResty** 的默认搭档，已经是 **LuaJIT 2.1** 很久了。但是针对不同系统的工具包安装工具，他们当前默认绑定推送的都还是 **LuaJIT 2.0**，所以这里就直接给出最符合我们最终方向的安装方法了。

验证 **LuaJIT** 是否安装成功

```
# luajit -v
LuaJIT 2.1.0-beta1 -- Copyright (C) 2005-2015 Mike Pall.
http://luajit.org/
```

如果想了解其他系统安装 **LuaJIT** 的步骤，或者安装过程中遇到问题，可以到 **LuaJIT** 官网查看：<http://luajit.org/install.html>

第一个“Hello World”

安装好 **LuaJIT** 后，开始我们的第一个 **hello world** 小程序。首先编写一个 **hello.lua** 文件，写入内容后，使用 **LuaJIT** 运行即可。

```
# cat hello.lua
print("hello world")
# luajit hello.lua
hello world
```

Lua 基础数据类型

函数 `type` 能够返回一个值或一个变量所属的类型。

```
print(type("hello world")) -->output:string
print(type(print))          -->output:function
print(type(true))           -->output:boolean
print(type(360.0))          -->output:number
print(type(nil))            -->output:nil
```

nil (空)

`nil` 是一种类型，Lua 将 `nil` 用于表示“无效值”。一个变量在第一次赋值前的默认值是 `nil`，将 `nil` 赋予给一个全局变量就等同于删除它。

```
local num
print(num)          -->output:nil

num = 100
print(num)          -->output:100
```

值得一提的是，OpenResty 的 Lua 接口还提供了一种特殊的空值，即 `ngx.null`，用来表示不同于 `nil` 的“空值”。后面在讨论 OpenResty 的 Redis 库的时候，我们还会遇到它。

boolean (布尔)

布尔类型，可选值 `true/false`；Lua 中 `nil` 和 `false` 为“假”，其它所有值均为“真”。比如 `0` 和空字符串就是“真”；C 或者 Perl 程序员或许会对此感到惊讶。

```

local a = true
local b = 0
local c = nil
if a then
    print("a")          -->output:a
else
    print("not a")     --这个没有执行
end

if b then
    print("b")          -->output:b
else
    print("not b")     --这个没有执行
end

if c then
    print("c")          --这个没有执行
else
    print("not c")      -->output:not c
end

```

number (数字)

Number 类型用于表示实数，和 C/C++ 里面的 double 类型很类似。可以使用数学函数 `math.floor` (向下取整) 和 `math.ceil` (向上取整) 进行取整操作。

```

local order = 3.99
local score = 98.01
print(math.floor(order))   -->output:3
print(math.ceil(score))    -->output:99

```

一般地，Lua 的 number 类型就是用双精度浮点数来实现的。值得一提的是，LuaJIT 支持所谓的“dual-number”(双数) 模式，即 LuaJIT 会根据上下文用整型来存储整数，而用双精度浮点数来存放浮点数。

另外，LuaJIT 还支持“长长整型”的大整数 (在 x86_64 体系结构上则是 64 位整数)。例如

```
print(9223372036854775807LL - 1) -->output:9223372036854775806LL
```

string (字符串)

Lua 中有三种方式表示字符串：

- 1、使用一对匹配的单引号。例：'hello'。
- 2、使用一对匹配的双引号。例："abclua"。

3、字符串还可以用一种长括号（即[[]]) 括起来的方式定义。我们把两个正的方括号（即[[]) 间插入 n 个等号定义为第 n 级正长括号。就是说，0 级正的长括号写作 [[]，一级正的长括号写作 [= [，如此等等。反的长括号也作类似定义；举个例子，4 级反的长括号写作]=====]。一个长字符串可以由任何一级的正的长括号开始，而由第一个碰到的同级反的长括号结束。整个词法分析过程将不受分行限制，不处理任何转义符，并且忽略掉任何不同级别的长括号。这种方式描述的字符串可以包含任何东西，当然本级别的反长括号除外。例：[[abc\nbc]]，里面的 "\n" 不会被转义。

另外，Lua 的字符串是不可改变的值，不能像在 c 语言中那样直接修改字符串的某个字符，而是根据修改要求来创建一个新的字符串。Lua 也不能通过下标来访问字符串的某个字符。想了解更多关于字符串的操作，请查看[String 库](#)章节。

```
local str1 = 'hello world'
local str2 = "hello lua"
local str3 = [[["add\name",'hello']]]
local str4 = [=string have a [[[]].]=]

print(str1)      -->output:hello world
print(str2)      -->output:hello lua
print(str3)      -->output:"add\name",'hello'
print(str4)      -->output:string have a [[[]].]
```

在 Lua 实现中，Lua 字符串一般都会经历一个“内化”(intern)的过程，即两个完全一样的 Lua 字符串在 Lua 虚拟机中只会存储一份。每一个 Lua 字符串在创建时都会插入到 Lua 虚拟机内部的一个全局的哈希表中。这意味着

1. 创建相同的 Lua 字符串并不会引入新的动态内存分配操作，所以相对便宜（但仍有全局哈希表查询的开销），
2. 内容相同的 Lua 字符串不会占用多份存储空间，
3. 已经创建好的 Lua 字符串之间进行相等性比较时是 $O(1)$ 时间度的开销，而不是通常见到的 $O(n)$.

table (表)

Table 类型实现了一种抽象的“关联数组”。“关联数组”是一种具有特殊索引方式的数组，索引通常是字符串 (string) 或者 number 类型，但也可以是除 nil 以外的任意类型的值。

```

local corp = {
    web = "www.google.com",      --索引为字符串，key = "web",
                                  --           value = "www.google.com"
    telephone = "12345678",     --索引为字符串
    staff = {"Jack", "Scott", "Gary"}, --索引为字符串，值也是一个表
    100876,                      --相当于 [1] = 100876，此时索引为数字
                                  --           key = 1, value = 100876
    100191,                      --相当于 [2] = 100191，此时索引为数字
    [10] = 360,                  --直接把数字索引给出
    ["city"] = "Beijing"         --索引为字符串
}

print(corp.web)                -->output:www.google.com
print(corp["telephone"])       -->output:12345678
print(corp[2])                 -->output:100191
print(corp["city"])            -->output:"Beijing"
print(corp.staff[1])           -->output:Jack
print(corp[10])                -->output:360

```

在内部实现上，table 通常实现为一个哈希表、一个数组、或者两者的混合。具体的实现为何种形式，动态依赖于具体的 table 的键分布特点。

想了解更多关于 table 的操作，请查看 [Table 库](#) 章节。

function (函数)

在 Lua 中，函数也是一种数据类型，函数可以存储在变量中，可以通过参数传递给其他函数，还可以作为其他函数的返回值。

示例

```

local function foo()
    print("in the function")
    --dosomething()
    local x = 10
    local y = 20
    return x + y
end

local a = foo      --把函数赋给变量

print(a())

--output:
in the function
30

```

有名函数的定义本质上是匿名函数对变量的赋值。为说明这一点，考虑

```
function foo()
end
```

等价于

```
foo = function ()
end
```

类似地，

```
local function foo()
end
```

等价于

```
local foo = function ()
end
```

表达式

算术运算符

Lua 的算术运算符如下表所示：

算术运算符	说明
+	加法
-	减法
*	乘法
/	除法
^	指数
%	取模

示例代码：test1.lua

```

print(1 + 2)      -->打印 3
print(5 / 10)     -->打印 0.5。 这是Lua不同于c语言的
print(5.0 / 10)   -->打印 0.5。 浮点数相除的结果是浮点数
-- print(10 / 0)   -->注意除数不能为0，计算的结果会出错
print(2 ^ 10)     -->打印 1024。 求2的10次方

local num = 1357
print(num % 2)    -->打印 1
print((num % 2) == 1) -->打印 true。 判断num是否为奇数
print((num % 5) == 0) -->打印 false。判断num是否能被5整数

```

关系运算符

关系运算符	说明
<	小于
>	大于
<=	小于等于
>=	大于等于
==	等于
~=	不等于

示例代码：test2.lua

```

print(1 < 2)      -->打印 true
print(1 == 2)     -->打印 false
print(1 ~= 2)    -->打印 true
local a, b = true, false
print(a == b)    -->打印 false

```

注意：Lua 语言中“不等于”运算符的写法为：`~=`

在使用“`==`”做等于判断时，要注意对于 `table`, `userdata` 和函数，Lua 是作引用比较的。也就是说，只有当两个变量引用同一个对象时，才认为它们相等。可以看下面的例子：

```

local a = { x = 1, y = 0}
local b = { x = 1, y = 0}
if a == b then
    print("a==b")
else
    print("a~=b")
end

---output:
a~=b

```

由于 Lua 字符串总是会被“内化”，即相同内容的字符串只会被保存一份，因此 Lua 字符串之间的相等性比较可以简化为其内部存储地址的比较。这意味着 Lua 字符串的相等性比较总是为 $O(1)$ 。而在其他编程语言中，字符串的相等性比较则通常为 $O(n)$ ，即需要逐个字节（或按若干个连续字节）进行比较。

逻辑运算符

逻辑运算符	说明
<code>and</code>	逻辑与
<code>or</code>	逻辑或
<code>not</code>	逻辑非

Lua 中的 `and` 和 `or` 是不同于 C 语言的。在 C 语言中，`and` 和 `or` 只得到两个值 1 和 0，其中 1 表示真，0 表示假。而 Lua 中 `and` 的执行过程是这样的：

- `a and b` 如果 `a` 为 `nil`，则返回 `a`，否则返回 `b`;
- `a or b` 如果 `a` 为 `nil`，则返回 `b`，否则返回 `a`。

示例代码：test3.lua

```

local c = nil
local d = 0
local e = 100
print(c and d)    -->打印 nil
print(c and e)    -->打印 nil
print(d and e)    -->打印 100
print(c or d)     -->打印 0
print(c or e)     -->打印 100
print(not c)      -->打印 true
print(not d)      -->打印 false

```

注意：所有逻辑操作符将 **false** 和 **nil** 视作假，其他任何值视作真，对于 **and** 和 **or**，“短路求值”，对于 **not**，永远只返回 **true** 或者 **false**。

字符串连接

在 **Lua** 中连接两个字符串，可以使用操作符“**..**”（两个点）。如果其任意一个操作数是数字的话，**Lua** 会将这个数字转换成字符串。**注意，连接操作符只会创建一个新字符串，而不会改变原操作数。**也可以使用 **string** 库函数 **string.format** 连接字符串。

```

print("Hello " .. "World")      -->打印 Hello World
print(0 .. 1)                   -->打印 01

str1 = string.format("%s-%s", "hello", "world")
print(str1)                     -->打印 hello-world

str2 = string.format("%d-%s-%.2f", 123, "world", 1.21)
print(str2)                     -->打印 123-world-1.21

```

由于 **Lua 字符串本质上是只读的**，因此字符串连接运算符几乎总会创建一个新的（更大的）字符串。这意味着如果有很多这样的连接操作（比如在循环中使用 **..** 来拼接最终结果），则性能损耗会非常大。在这种情况下，推荐使用 **table** 和 **table.concat()** 来进行很多字符串的拼接，例如：

```

local pieces = {}
for i, elem in ipairs(my_list) do
    pieces[i] = my_process(elem)
end
local res = table.concat(pieces)

```

当然，上面的例子还可以使用 **LuaJIT** 独有的 **table.new** 来恰当地初始化 **pieces** 表的空间，以避免该表的动态生长。这个特性我们在后面还会详细讨论。

优先级

Lua 操作符的优先级如下表所示(从高到低)：

优先级					
\wedge					
not # -					
*	/	%			
+	-				
..					
<	>	\leq	\geq	$= =$	$\sim =$
and					
or					

示例：

```
local a, b = 1, 2
local x, y = 3, 4
local i = 10
local res = 0
res = a + i < b/2 + 1 -->等价于res = (a + i) < ((b/2) + 1)
res = 5 + x^2*8          -->等价于res = 5 + ((x^2) * 8)
res = a < y and y <= x -->等价于res = (a < y) and (y <= x)
```

若不确定某些操作符的优先级，就应显示地用括号来指定运算顺序。这样做还可以提高代码的可读性。

控制结构

流程控制语句对于程序设计来说特别重要，它可用于设定程序的逻辑结构。一般需要与条件判断语句结合使用。Lua 语言提供的控制结构有 `if`，`while`，`repeat`，`for`，并提供 `break` 关键字来满足更丰富的需求。本章主要介绍 Lua 语言的控制结构的使用。

控制结构 if-else

if-else 是我们熟知的一种控制结构。Lua 跟其他语言一样，提供了 **if-else** 的控制结构。因为是大家熟悉的语法，本节只简单介绍一下它的使用方法。

单个 if 分支型

```
x = 10
if x > 0 then
    print("x is a positive number")
end
```

运行输出：x is a positive number

两个分支 if-else 型

```
x = 10
if x > 0 then
    print("x is a positive number")
else
    print("x is a non-positive number")
end
```

运行输出：x is a positive number

多个分支 if-elseif-else 型

```
score = 90
if score == 100 then
    print("Very good!Your score is 100")
elseif score >= 60 then
    print("Congratulations, you have passed it,your score greater or equal to 60")
--此处可以添加多个elseif
else
    print("Sorry, you do not pass the exam! ")
end
```

运行输出：Congratulations, you have passed it,your score greater or equal to 60

与 C 语言的不同之处是 **else** 与 **if** 是连在一起的，若将 **else** 与 **if** 写成 "else if" 则相当于在 **else** 里嵌套另一个 **if** 语句，如下代码：

```
score = 0
if score == 100 then
    print("Very good!Your score is 100")
elseif score >= 60 then
    print("Congratulations, you have passed it,your score greater or equal to 60")
else
    if score > 0 then
        print("Your score is better than 0")
    else
        print("My God, your score turned out to be 0")
    end --与上一示例代码不同的是，此处要添加一个end
end
```

运行输出：My God, your score turned out to be 0

while 型控制结构

Lua 跟其他常见语言一样，提供了 `while` 控制结构，语法上也没有什么特别的。但是没有提供 `do-while` 型的控制结构，但是提供了功能相当的 `repeat`。

`while` 型控制结构语法如下，当表达式值为假（即 `false` 或 `nil`）时结束循环。也可以使用 `break` 语句提前跳出循环。

```
while 表达式 do
  --body
end
```

示例代码，求 $1 + 2 + 3 + 4 + 5$ 的结果

```
x = 1
sum = 0

while x <= 5 do
  sum = sum + x
  x = x + 1
end
print(sum) -->output 15
```

值得一提的是，Lua 并没有像许多其他语言那样提供类似 `continue` 这样的控制语句用来立即进入下一个循环迭代（如果有的话）。因此，我们需要仔细地安排循环体里的分支，以避免这样的需求。

没有提供 `continue`，却也提供了另外一个标准控制语句 `break`，可以跳出当前循环。例如我们遍历 `table`，查找值为 11 的数组下标索引：

```
local t = {1, 3, 5, 8, 11, 18, 21}

local i
for i, v in ipairs(t) do
  if 11 == v then
    print("index[" .. i .. "] have right value[11]")
    break
  end
end
```

repeat 控制结构

Lua 中的 **repeat** 控制结构类似于其他语言（如：C++ 语言）中的 **do-while**，但是控制方式是刚好相反的。简单点说，执行 **repeat** 循环体后，直到 **until** 的条件为真时才结束，而其他语言（如：C++ 语言）的 **do-while** 则是当条件为假时就结束循环。

以下代码将会形成死循环：

```
x = 10
repeat
    print(x)
until false
```

该代码将导致死循环，因为 **until** 的条件一直为假，循环不会结束

除此之外，**repeat** 与其他语言的 **do-while** 基本是一样的。同样，Lua 中的 **repeat** 也可以在使用 **break** 退出。

for 控制结构

Lua 提供了一组传统的、小巧的控制结构，包括用于条件判断的 `if` 用于迭代的 `while`、`repeat` 和 `for`，本章节主要介绍 `for` 的使用。

for 数字型

`for` 语句有两种形式：数字 `for`（numeric `for`）和范型 `for`（generic `for`）。

数字型 `for` 的语法如下：

```
for var = begin, finish, step do
    --body
end
```

关于数字 `for` 需要关注以下几点：1.`var` 从 `begin` 变化到 `finish`，每次变化都以 `step` 作为步长递增 2.`begin`、`finish`、`step` 三个表达式只会在循环开始时执行一次 3.第三个表达式 `step` 是可选的，默认为 1 4.控制变量 `var` 的作用域仅在 `for` 循环内，需要在外面控制，则需将值赋给一个新的变量 5.循环过程中不要改变控制变量的值，那样会带来不可预知的影响

示例

```
for i = 1, 5 do
    print(i)
end

-- output:
1
2
3
4
5
```

...

for

```
for i = 1, 10, 2 do
    print(i)
end

-- output:
1
3
5
7
9
```

以下是这种循环的一个典型示例：

```
for i = 10, 1, -1 do
    print(i)
end

-- output:
...
```

如果不想给循环设置上限的话，可以使用常量 `math.huge`：

```
for i = 1, math.huge do
    if (0.3*i^3 - 20*i^2 - 500 >=0) then
        print(i)
        break
    end
end
```

for 泛型

泛型 `for` 循环通过一个迭代器 (`iterator`) 函数来遍历所有值：

```
-- 打印数组a的所有值
local a = {"a", "b", "c", "d"}
for i, v in ipairs(a) do
    print("index:", i, " value:", v)
end

-- output:
index: 1 value: a
index: 2 value: b
index: 3 value: c
index: 4 value: d
```

for

Lua 的基础库提供了 `ipairs`，这是一个用于遍历数组的迭代器函数。在每次循环中，`i` 会被赋予一个索引值，同时 `v` 被赋予一个对应于该索引的数组元素值。

下面是另一个类似的示例，演示了如何遍历一个 `table` 中所有的 `key`

```
-- 打印table t中所有的key
for k in pairs(t) do
    print(k)
end
```

从外观上看泛型 `for` 比较简单，但其实它是非常强大的。**通过不同的迭代器，几乎可以遍历所有的东西**，而且写出的代码极具可读性。标准库提供了几种迭代器，包括用于迭代文件中每行的 (`io.lines`)、迭代 `table` 元素的 (`pairs`)、迭代数组元素的 (`ipairs`)、迭代字符串中单词的 (`string.gmatch`) 等。

泛型 `for` 循环与数字型 `for` 循环有两个相同点：(1) 循环变量是循环体的局部变量；(2) 决不应该对循环变量作任何赋值。

对于泛型 `for` 的使用，再来看一个更具体的示例。假设有这样一个 `table`，它的内容是一周中每天的名称：

```
local days = {
    "Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"
}
```

现在要将一个名称转换成它在一周中的位置。为此，需要根据给定的名称来搜索这个 `table`。然而在 Lua 中，通常更有效的方法是创建一个“逆向 `table`”。例如这个逆向 `table` 叫 `revDays`，它以一周中每天的名称作为索引，位置数字作为值：

```
local revDays = {
    ["Sunday"] = 1,
    ["Monday"] = 2,
    ["Tuesday"] = 3,
    ["Wednesday"] = 4,
    ["Thursday"] = 5,
    ["Friday"] = 6,
    ["Saturday"] = 7
}
```

接下来，要找出一个名称所对应的需要，只需用名字来索引这个 `reverse table` 即可：

```
local x = "Tuesday"
print(revDays[x]) -->3
```

当然，不必手动声明这个逆向 table，而是通过原来的 table 自动地构造出这个逆向 table：

```
local days = {  
    "Monday", "Tuesday", "Wednesday", "Thursday",  
    "Friday", "Saturday", "Sunday"  
}  
  
local revDays = {}  
for k, v in pairs(days) do  
    revDays[v] = k  
end  
  
-- print value  
for k,v in pairs(revDays) do  
    print("k:", k, " v:", v)  
end  
  
-- output:  
k: Tuesday v: 2  
k: Monday v: 1  
k: Sunday v: 7  
k: Thursday v: 4  
k: Friday v: 5  
k: Wednesday v: 3  
k: Saturday v: 6
```

这个循环会为每个元素进行赋值，其中变量 `k` 为 `key(1、2、...)`，变量 `v` 为 `value("Sunday"、"Monday"、...)`。

值得一提的是，在 LuaJIT 2.1 中，`ipairs()` 内建函数是可以被 JIT 编译的，而 `pairs()` 则只能被解释执行。因此在性能敏感的场景，应当合理安排数据结构，避免对哈希表进行遍历。事实上，即使未来 `pairs` 可以被 JIT 编译，哈希表的遍历本身也不会有数组遍历那么高效，毕竟哈希表就不是为遍历而设计的数据结构。

break，return 关键字

break

语句 `break` 用来终止 `while`、`repeat` 和 `for` 三种循环的执行，并跳出当前循环体，继续执行当前循环之后的语句。下面举一个 `while` 循环中的 `break` 的例子来说明：

```
-- 计算最小的x, 使从1到x的所有数相加和大于100
sum = 0
i = 1
while true do
    sum = sum + i
    if sum > 100 then
        break
    end
    i = i + 1
end
print("The result is " .. i) -->output:The result is 14
```

在实际应用中，`break` 经常用于嵌套循环中。

return

`return` 主要用于从函数中返回结果，或者用于简单的结束一个函数的执行。关于函数返回值的细节可以参考 [函数的返回值](#) 章节。`return` 只能写在语句块的最后，一旦执行了 `return` 语句，该语句之后的所有语句都不会再执行。若要写在函数中间，则只能写在一个显式的语句块内，参见示例代码：

break , return

```
local function add(x, y)
    return x + y
    --print("add: I will return the result " .. (x + y))
    --因为前面有个return，若不注释该语句，则会报错
end

local function is_positive(x)
    if x > 0 then
        return x .. " is positive"
    else
        return x .. " is non-positive"
    end

    --由于return只出现在前面显式的语句块，所以此语句不注释也不会报错
    --，但是不会被执行，此处不会产生输出
    print("function end!")
end

sum = add(10, 20)
print("The sum is " .. sum) -->output:The sum is 30
answer = is_positive(-10)
print(answer)           -->output:-10 is non-positive
```

有时候，为了调试方便，我们可以想在某个函数的中间提前 `return`，以进行控制流的短路。此时我们可以将 `return` 放在一个 `do ... end` 代码块中，例如：

```
local function foo()
    print("before")
    do return end
    print("after") -- 这一行语句永远不会执行到
end
```

Lua 函数

在 Lua 中，函数是一种对语句和表达式进行抽象的主要机制。函数既可以完成某项特定的任务，也可以做一些计算并返回结果。在第一种情况中，一句函数调用被视为一条语句；而在第二种情况中，则将其视为一句表达式。

示例代码：

```
print("hello world!")          -- 用 print() 函数输出 hello world!
local m = math.max(1, 5)        -- 调用数学库函数 max,
                                -- 用来求 1,5 中的最大值，并返回赋给变量 m
```

使用函数的好处：

1. 降低程序的复杂性：把函数作为一个独立的模块，写完函数后，只关心它的功能，而不再考虑函数里面的细节。
2. 增加程序的可读性：当我们调用 `math.max()` 函数时，很明显函数是用于求最大值的，实现细节就不关心了。
3. 避免重复代码：当程序中有相同的代码部分时，可以把这部分写成一个函数，通过调用函数来实现这部分代码的功能，节约空间，减少代码长度。
4. 隐含局部变量：在函数中使用局部变量，变量的作用范围不会超出函数，这样它就不会给外界带来干扰。

函数定义

Lua 使用关键字 `function` 定义函数，语法如下：

```
function function_name (arc) -- arc 表示参数列表，函数的参数列表可以为空
    -- body
end
```

上面的语法定义了一个全局函数，名为 `function_name`。全局函数本质上就是函数类型的值赋给了一个全局变量，即上面的语法等价于

```
function_name = function (arc)
    -- body
end
```

由于全局变量一般会污染全局名字空间，同时也有性能损耗（即查询全局环境表的开销），因此我们应当尽量使用“局部函数”，其记法是类似的，只是开头加上 `local` 修饰符：

```
local function function_name (arc)
    -- body
end
```

由于函数定义本质上就是变量赋值，而变量的定义总是应放置在变量使用之前，所以函数的定义也需要放置在函数调用之前。

示例代码：

```
local function max(a, b) -- 定义函数 max，用来求两个数的最大值，并返回
    local temp = nil -- 使用局部变量 temp，保存最大值
    if(a > b) then
        temp = a
    else
        temp = b
    end
    return temp -- 返回最大值
end

local m = max(-12, 20) -- 调用函数 max，找出 -12 和 20 中的最大值
print(m) --> output 20
```

如果参数列表为空，必须使用 `()` 表明是函数调用。

示例代码：

```
local function func() --形参为空
    print("no parameter")
end

func() -- 函数调用，圆扩号不能省

--> output:
no parameter
```

在定义函数要注意几点：

1. 利用名字来解释函数、变量的目的，使人通过名字就能看出来函数、变量的作用。
2. 每个函数的长度要尽量控制在一个屏幕内，一眼可以看明白。
3. 让代码自己说话，不需要注释最好。

由于函数定义等价于变量赋值，我们也可以把函数名替换为某个 Lua 表的某个字段，例如

```
function foo.bar(a, b, c)
    -- body ...
end
```

此时我们是把一个函数类型的值赋给了 `foo` 表的 `bar` 字段。换言之，上面的定义等价于

```
foo.bar = function (a, b, c)
    print(a, b, c)
end
```

对于此种形式的函数定义，不能再使用 `local` 修饰符了，因为不存在定义新的局部变量了。

函数的参数

按值传递

Lua 函数的参数大部分是按值传递的。值传递就是调用函数时，实参把它的值通过赋值运算传递给形参，然后形参的改变和实参就没有关系了。在这个过程中，实参是通过它在参数表中的位置与形参匹配起来的。

示例代码：

```
local function swap(a, b) --定义函数swap, 函数内部进行交换两个变量的值
    local temp = a
    a = b
    b = temp
    print(a, b)
end

local x = "hello"
local y = 20
print(x, y)
swap(x, y)      --调用swap函数
print(x, y)      --调用swap函数后, x和y的值并没有交换

-->output
hello 20
20 hello
hello 20
```

在调用函数的时候，若形参数个数和实参数个数不同时，Lua 会自动调整实参数个数。调整规则：若实参数个数大于形参数个数，从左向右，多余的实参被忽略；若实参数个数小于形参数个数，从左向右，没有被实参初始化的形参会被初始化为 nil。

示例代码：

```

local function fun1(a, b)          --两个形参，多余的实参被忽略掉
    print(a, b)
end

local function fun2(a, b, c, d) --四个形参，没有被实参初始化的形参，用nil初始化
    print(a, b, c, d)
end

local x = 1
local y = 2
local z = 3

fun1(x, y, z)          -- z被函数fun1忽略掉了，参数变成 x, y
fun2(x, y, z)          -- 后面自动加上一个nil，参数变成 x, y, z, nil

-->output
1 2
1 2 3 nil

```

变长参数

上面函数的参数都是固定的，其实 Lua 还支持变长参数。若形参为 `...`，表示该函数可以接收不同长度的参数。访问参数的时候也要使用 `...`。

示例代码：

```

local function func( ... )          -- 形参为 ..., 表示函数采用变长参数

    local temp = {...}            -- 访问的时候也要使用 ...
    local ans = table.concat(temp, " ") -- 使用 table.concat 库函数对数
                                         -- 组内容使用 " " 拼接成字符串。
    print(ans)
end

func(1, 2)           -- 传递了两个参数
func(1, 2, 3, 4)    -- 传递了四个参数

-->output
1 2
1 2 3 4

```

值得一提的是，**LuaJIT 2** 尚不能 **JIT** 编译这种变长参数的用法，只能解释执行。所以对性能敏感的代码，应当避免使用此种形式。

具名参数

Lua 还支持通过名称来指定实参，这时候要把所有的实参组织到一个 **table** 中，并将这个 **table** 作为唯一的实参传给函数。

示例代码：

```
local function change(arg) -- change 函数，改变长方形的长和宽，使其各增长一倍
    arg.width = arg.width * 2
    arg.height = arg.height * 2
    return arg
end

local rectangle = { width = 20, height = 15 }
print("before change:", "width =", rectangle.width,
                  "height =", rectangle.height)
rectangle = change(rectangle)
print("after change:", "width =", rectangle.width,
                  "height =", rectangle.height)

-->output
before change: width = 20 height = 15
after change: width = 40 height = 30
```

按引用传递

当函数参数是 **table** 类型时，传递进来的是实际参数的引用，此时在函数内部对该 **table** 所做的修改，会直接对调用者所传递的实际参数生效，而无需自己返回结果和让调用者进行赋值。我们把上面改变长方形长和宽的例子修改一下。

示例代码：

```
function change(arg) --change函数，改变长方形的长和宽，使其各增长一倍
    arg.width = arg.width * 2 --表arg不是表rectangle的拷贝，他们是同一个表
    arg.height = arg.height * 2
end -- 没有return语句了

local rectangle = { width = 20, height = 15 }
print("before change:", "width =", rectangle.width,
                  "height =", rectangle.height)
change(rectangle)
print("after change:", "width =", rectangle.width,
                  "height =", rectangle.height)

--> output
before change: width = 20 height = 15
after change: width = 40 height = 30
```

在常用基本类型中，除了 **table** 是按址传递类型外，其它的都是按值传递参数。用全局变量来代替函数参数的不好编程习惯应该被抵制，良好的编程习惯应该是减少全局变量的使用。

函数返回值

Lua 具有一项与众不同的特性，允许函数返回多个值。**Lua** 的库函数中，有一些就是返回多个值。

示例代码：使用库函数 `string.find`，在源字符串中查找目标字符串，若查找成功，则返回目标字符串在源字符串中的起始位置和结束位置的下标。

```
local s, e = string.find("hello world", "llo")
print(s, e) -->output 3 5
```

返回多个值时，值之间用“,”隔开。

示例代码：定义一个函数，实现两个变量交换值

```
local function swap(a, b) -- 定义函数 swap，实现两个变量交换值
    return b, a           -- 按相反顺序返回变量的值
end

local x = 1
local y = 20
x, y = swap(x, y)        -- 调用 swap 函数
print(x, y)               --> output 20 1
```

当函数返回值的个数和接收返回值的变量的个数不一致时，**Lua** 也会自动调整参数个数。

调整规则：若返回值个数大于接收变量的个数，多余的返回值会被忽略掉；若返回值个数小于参数个数，从左向右，没有被返回值初始化的变量会被初始化为 `nil`。

示例代码：

```
function init()           --init 函数 返回两个值 1 和 "lua"
    return 1, "lua"
end

x = init()
print(x)

x, y, z = init()
print(x, y, z)

--output
1
1 lua nil
```

当一个函数有一个以上返回值，且函数调用不是一个列表表达式的最后一个元素，那么函数调用只会产生一个返回值，也就是第一个返回值。

示例代码：

```
local function init()          -- init 函数 返回两个值 1 和 "lua"
    return 1, "lua"
end

local x, y, z = init(), 2    -- init 函数的位置不在最后，此时只返回 1
print(x, y, z)              -->output 1 2 nil

local a, b, c = 2, init()    -- init 函数的位置在最后，此时返回 1 和 "lua"
print(a, b, c)              -->output 2 1 lua
```

函数调用的实参列表也是一个列表表达式。考虑下面的例子：

```
local function init()
    return 1, "lua"
end

print(init(), 2)      -->output 1 2
print(2, init())     -->output 2 1 lua
```

如果你确保只取函数返回值的第一个值，可以使用括号运算符，例如

```
local function init()
    return 1, "lua"
end

print((init()), 2)    -->output 1 2
print(2, (init()))   -->output 2 1 lua
```

值得一提的是，如果实参列表中某个函数会返回多个值，同时调用者又没有显式地使用括号运算符来筛选和过滤，则这样的表达式是不能被 **LuaJIT 2** 所 JIT 编译的，而只能被解释执行。

全动态函数调用

调用回调函数，并把一个数组参数作为回调函数的参数。

```
local args = {...} or {}
method_name(unpack(args, 1, table.maxn(args)))
```

使用场景

如果你的实参 `table` 中确定没有 `nil` 空洞，则可以简化为

```
method_name(unpack(args))
```

1. 你要调用的函数参数是未知的；
2. 函数的实际参数的类型和数目也都是未知的。

伪代码

```
add_task(end_time, callback, params)

if os.time() >= endTime then
    callback(unpack(params, 1, table.maxn(params)))
end
```

值得一提的是，`unpack` 内建函数还不能为 `LuaJIT` 所 `JIT` 编译，因此这种用法总是会被解释执行。对性能敏感的代码路径应避免这种用法。

小试牛刀

```
local function run(x, y)
    print('run', x, y)
end

local function attack(targetId)
    print('targetId', targetId)
end

local function do_action(method, ...)
    local args = {...} or {}
    method(unpack(args, 1, table.maxn(args)))
end

do_action(run, 1, 2)          -- output: run 1 2
do_action(attack, 1111)       -- output: targetId 1111
```

模块

从 Lua 5.1 语言添加了对模块和包的支持。一个 Lua 模块的数据结构是用一个 Lua 值（通常是一个 Lua 表或者 Lua 函数）。一个 Lua 模块代码就是一个会返回这个 Lua 值的代码块。可以使用内建函数 `require()` 来加载和缓存模块。简单的说，一个代码模块就是一个程序库，可以通过 `require` 来加载。模块加载后的结果通过是一个 Lua `table`，这个表就像是一个命名空间，其内容就是模块中导出的所有东西，比如函数和变量。`require` 函数会返回 Lua 模块加载后的结果，即用于表示该 Lua 模块的 Lua 值。

require 函数

Lua 提供了一个名为 `require` 的函数用来加载模块。要加载一个模块，只需要简单地调用 `require "file"` 就可以了，`file` 指模块所在的文件名。这个调用会返回一个由模块函数组成的 `table`，并且还会定义一个包含该 `table` 的全局变量。

在 Lua 中创建一个模块最简单的方法是：创建一个 `table`，并将所有需要导出的函数放入其中，最后返回这个 `table` 就可以了。相当于将导出的函数作为 `table` 的一个字段，在 Lua 中函数是第一类值，提供了天然的优势。

把下面的代码保存在文件 `my.lua` 中

```
local foo={}

local function getname()
    return "Lucy"
end

function foo.greeting()
    print("hello " .. getname())
end

return foo
```

把下面代码保存在文件 `main.lua` 中，然后执行 `main.lua`，调用上述模块。

```
local fp = require("my")
fp.greeting()      -->output: hello Lucy
```

注：对于需要导出给外部使用的公共模块，出于安全考虑，是要避免全局变量的出现。我们可以使用 `lj-releng` 或 `luacheck` 工具完成全局变量的检测。具体参考本章的 [局部变量](#) 和“[测试](#)”一章的 [代码静态分析](#)。

String 库

Lua 字符串库包含很多强大的字符操作函数。字符串库中的所有函数都导出在模块 `string` 中。在 **Lua 5.1** 中，它还将这些函数导出作为 `string` 类型的方法。这样假设要返回一个字符串转的大写形式，可以写成 `ans = string.upper(s)`，也能写成 `ans = s:upper()`。为了避免与之前版本不兼容，此处使用前者。

Lua 字符串总是由字节构成的。**Lua** 核心并不尝试理解具体的字符集编码（比如 **GBK** 和 **UTF-8** 这样的多字节字符编码）。

需要特别注意的一点是，**Lua** 字符串内部用来标识各个组成字节的下标是从 1 开始的，这不同于像 **C** 和 **Perl** 这样的编程语言。这样数字符串位置的时候再也不用调整，对于非专业的开发者来说可能也是一个好事情，`string.sub(str, 3, 7)` 直接表示从第三个字符开始到第七个字符（含）为止的子串。

string.byte(s [, i [, j]])

返回字符 `s[i]`、`s[i + 1]`、`s[i + 2]`、……、`s[j]` 所对应的 ASCII 码。`i` 的默认值为 1，即第一个字节，`j` 的默认值为 `i`。

示例代码

```
print(string.byte("abc", 1, 3))
print(string.byte("abc", 3)) -- 缺少第三个参数，第三个参数默认与第二个相同，此时为 3
print(string.byte("abc"))    -- 缺少第二个和第三个参数，此时这两个参数都默认为 1

-->output
97    98    99
99
97
```

由于 `string.byte` 只返回整数，而并不像 `string.sub` 等函数那样（尝试）创建新的 **Lua** 字符串，因此使用 `string.byte` 来进行字符串相关的扫描和分析是最为高效的，尤其是在被 **LuajIT 2** 所 JIT 编译之后。

string.char (...)

接收 0 个或更多的整数（整数范围：0~255），返回这些整数所对应的 ASCII 码字符组成的字符串。当参数为空时，默认是一个 0。

示例代码

```

print(string.char(96, 97, 98))
print(string.char())           -- 参数为空，默认是一个0，
                               -- 你可以用string.byte(string.char())测试一下
print(string.char(65, 66))

--> output
`ab

AB

```

此函数特别适合从具体的字节构造出二进制字符串。这经常比使用 `table.concat` 函数和 `..` 连接运算符更加高效。

string.upper(s)

接收一个字符串 `s`，返回一个把所有小写字母变成大写字母的字符串。

示例代码

```
print(string.upper("Hello Lua")) -->output HELLO LUA
```

string.lower(s)

接收一个字符串 `s`，返回一个把所有大写字母变成小写字母的字符串。

示例代码

```
print(string.lower("Hello Lua")) -->output hello lua
```

string.len(s)

接收一个字符串，返回它的长度。

示例代码

```
print(string.len("hello lua")) -->output 9
```

使用此函数是不推荐的。应当总是使用 `#` 运算符来获取 Lua 字符串的长度。

由于 Lua 字符串的长度是专门存放的，并不需要像 C 字符串那样即时计算，因此获取字符串长度的操作总是 `O(1)` 的时间复杂度。

string.find(s, p [, init [, plain]])

在 `s` 字符串中第一次匹配 `p` 字符串。若匹配成功，则返回 `p` 字符串在 `s` 字符串中出现的开始位置和结束位置；若匹配失败，则返回 `nil`。第三个参数 `init` 默认为 `1`，并且可以为负整数，当 `init` 为负数时，表示从 `s` 字符串的 `string.len(s) + init` 索引处开始向后匹配字符串 `p`。第四个参数默认为 `false`，当其为 `true` 时，只会把 `p` 看成一个字符串对待。

示例代码

```
local find = string.find
print(find("abc cba", "ab"))
print(find("abc cba", "ab", 2))      -- 从索引为2的位置开始匹配字符串：ab
print(find("abc cba", "ba", -1))     -- 从索引为7的位置开始匹配字符串：ba
print(find("abc cba", "ba", -3))     -- 从索引为6的位置开始匹配字符串：ba
print(find("abc cba", "(%a+)", 1))   -- 从索引为1处匹配最长连续且只含字母的字符串
print(find("abc cba", "(%a+)", 1, true)) --从索引为1的位置开始匹配字符串：(%a+)

-->output
1 2
nil
nil
6 7
1 3 abc
nil
```

对于 LuaJIT 这里有个性能优化点，对于 `string.find` 方法，当只有字符串查找匹配时，是可以被 JIT 编译器优化的，有关 JIT 可以编译优化清单，大家可以参考 <http://wiki.luajit.org/NYI>，性能提升是非常明显的，通常是 100 倍量级。这里有个的例子，大家可以参考 <https://groups.google.com/forum/m/#!topic/openresty-en/rwS88FGRsUI>。

string.format(formatstring, ...)

按照格式化参数 `formatstring`，返回后面 `...` 内容的格式化版本。编写格式化字符串的规则与标准 C 语言中 `printf` 函数的规则基本相同：它由常规文本和指示组成，这些指示控制了每个参数应放到格式化结果的什么位置，及如何放入它们。一个指示由字符 `%` 加上一个字母组成，这些字母指定了如何格式化参数，例如 `a` 用于十进制数、`x` 用于十六进制数、`o` 用于八进制数、`f` 用于浮点数、`s` 用于字符串等。在字符 `%` 和字母之间可以再指定一些其他选项，用于控制格式的细节。

示例代码

```

print(string.format("%.4f", 3.1415926))      -- 保留4位小数
print(string.format("%d %x %o", 31, 31, 31))-- 十进制数31转换成不同进制
d = 29; m = 7; y = 2015                      -- 一行包含几个语句，用 ; 分开
print(string.format("%s %02d/%02d/%d", "today is:", d, m, y))

-->output
3.1416
31 1f 37
today is: 29/07/2015

```

string.match(s, p [, init])

在字符串 **s** 中匹配（模式）字符串 **p**，若匹配成功，则返回目标字符串中与模式匹配的子串；否则返回 **nil**。第三个参数 **init** 默认为 **1**，并且可以为负整数，当 **init** 为负数时，表示从 **s** 字符串的 **string.len(s) + init** 索引处开始向后匹配字符串 **p**。

示例代码

```

print(string.match("hello lua", "lua"))
print(string.match("lua lua", "lua", 2))  -- 匹配后面那个lua
print(string.match("lua lua", "hello"))
print(string.match("today is 27/7/2015", "%d+/%d+/%d+"))

-->output
lua
lua
nil
27/7/2015

```

`string.match` 目前并不能被 JIT 编译，应尽量使用 `ngx_lua` 模块提供的 `ngx.re.match` 等接口。

string.gmatch(s, p)

返回一个迭代器函数，通过这个迭代器函数可以遍历到在字符串 **s** 中出现模式串 **p** 的所有地方。

示例代码

```

s = "hello world from Lua"
for w in string.gmatch(s, "%a+") do -- 匹配最长连续且只含字母的字符串
    print(w)
end

-->output
hello
world
from
Lua

t = {}
s = "from=world, to=Lua"
for k, v in string.gmatch(s, "(%a+)=(%a+)") do -- 匹配两个最长连续且只含字母的
    t[k] = v                                -- 字符串，它们之间用等号连接
end
for k, v in pairs(t) do
    print (k,v)
end

-->output
to      Lua
from   world

```

此函数目前并不能被 LuaJIT 所 JIT 编译，而只能被解释执行。应尽量使用 `ngx_lua` 模块提供的 `ngx.re.gmatch` 等接口。

string.rep(s, n)

返回字符串 `s` 的 `n` 次拷贝。

示例代码

```

print(string.rep("abc", 3)) -- 拷贝3次"abc"

-->output abcabca

```

string.sub(s, i [, j])

返回字符串 `s` 中，索引 `i` 到索引 `j` 之间的子字符串。当 `j` 缺省时，默认为 `-1`，也就是字符串 `s` 的最后位置。`i` 可以为负数。当索引 `i` 在字符串 `s` 的位置在索引 `j` 的后面时，将返回一个空字符串。

示例代码

```

print(string.sub("Hello Lua", 4, 7))
print(string.sub("Hello Lua", 2))
print(string.sub("Hello Lua", 2, 1))    --看到返回什么了吗
print(string.sub("Hello Lua", -3, -1))

-->output
lo L
ello Lua

Lua

```

如果你只是想对字符串中的单个字节进行检查，使用 `string.char` 函数通常会更为高效。

string.gsub(s, p, r [, n])

将目标字符串 `s` 中所有的子串 `p` 替换成字符串 `r`。可选参数 `n`，表示限制替换次数。返回值有两个，第一个是被替换后的字符串，第二个是替换了多少次。

示例代码

```

print(string.gsub("Lua Lua Lua", "Lua", "hello"))
print(string.gsub("Lua Lua Lua", "Lua", "hello", 2)) --指明第四个参数

-->output
hello hello hello  3
hello hello Lua    2

```

此函数不能为 `LuaJIT` 所 `JIT` 编译，而只能被解释执行。一般我们推荐使用 `ngx_lua` 模块提供的 `ngx.re.gsub` 函数。

string.reverse (s)

接收一个字符串 `s`，返回这个字符串的反转。

示例代码

```
print(string.reverse("Hello Lua")) --> output: auL olleH
```

table 库

table 库是由一些辅助函数构成的，这些函数将 **table** 作为数组来操作。

下标从 1 开始

在 *Lua* 中，数组下标从 1 开始计数。

官方解释：*Lua lists have a base index of 1 because it was thought to be most friendly for non-programmers, as it makes indices correspond to ordinal element positions.*

确实，对于我们数数来说，总是从 1 开始数的，而从 0 开始对于描述偏移量这样的东西有利。而 *Lua* 最初设计是一种类似 XML 的数据描述语言，所以索引（**index**）反应的是数据在里面的位置，而不是偏移量。

在初始化一个数组的时候，若不显式地用键值对方式赋值，则会默认用数字作为下标，从 1 开始。由于在 *Lua* 内部实际采用哈希表和数组分别保存键值对、普通值，所以不推荐混合使用这两种赋值方式。

```
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
```

从其他语言过来的开发者会觉得比较坑的一点是，当我们把 **table** 当作栈或者队列使用的时候，容易犯错，追加到 **table** 的末尾用的是 `s[#s+1] = something`，而不是 `s[#s] = something`，而且如果这个 `something` 是一个 `nil` 的话，会导致这一次压栈（或者入队列）没有存入任何东西，`#s` 的值没有变。如果 `s = { 1, 2, 3, 4, 5, 6 }`，你令 `s[4] = nil`，`#s` 会令你“匪夷所思”地变成 3。

table.getn 获取长度

取长度操作符写作一元操作 `#`。字符串的长度是它的字节数（就是以一个字符一个字节计算的字符串长度）。

对于常规的数组，里面从 1 到 `n` 放着一些非空的值的时候，它的长度就精确的为 `n`，即最后一个值的下标。如果数组有一个“空洞”（就是说，`nil` 值被夹在非空值之间），那么 `#t` 可能是指向任何一个非 `nil` 值的前一个位置的下标（就是说，任何一个 `nil` 值都有可能被当成数组的结束）。这也就说明对于有“空洞”的情况，**table** 的长度存在一定的不可确定性。

```

local tblTest1 = { 1, a = 2, 3 }
print("Test1 " .. table.getn(tblTest1))

local tblTest2 = { 1, nil }
print("Test2 " .. table.getn(tblTest2))

local tblTest3 = { 1, nil, 2 }
print("Test3 " .. table.getn(tblTest3))

local tblTest4 = { 1, nil, 2, nil }
print("Test4 " .. table.getn(tblTest4))

local tblTest5 = { 1, nil, 2, nil, 3, nil }
print("Test5 " .. table.getn(tblTest5))

local tblTest6 = { 1, nil, 2, nil, 3, nil, 4, nil }
print("Test6 " .. table.getn(tblTest6))

```

我们使用 **Lua 5.1** 和 **LuaJIT 2.1** 分别执行这个用例，结果如下：

```

# lua test.lua
Test1 2
Test2 1
Test3 3
Test4 1
Test5 3
Test6 1
# luajit test.lua
Test1 2
Test2 1
Test3 1
Test4 1
Test5 1
Test6 1

```

这一段的输出结果，就是这么匪夷所思。请问，你以后还敢在 **Lua** 的 **table** 中用 **nil** 值吗？如果你继续往后面加 **nil**，你可能会发现点什么。你可能认为你发现的是个规律。但是，你千万不要认为这是个规律，因为这是错误的。

不要在 **Lua** 的 **table** 中使用 **nil** 值，如果一个元素要删除，直接 **remove**，不要用 **nil** 去代替。

table.concat (table [, sep [, i [, j]]])

对于元素是 **string** 或者 **number** 类型的表 **table**，返回 **table[i]..sep..table[i+1] ... sep..table[j]** 连接成的字符串。填充字符串 **sep** 默认为空白字符串。起始索引位置 **i** 默认为 1，结束索引位置 **j** 默认是 **table** 的长度。如果 **i** 大于 **j**，返回一个空字符串。

示例代码

```

local a = {1, 3, 5, "hello" }
print(table.concat(a))           -- output: 135hello
print(table.concat(a, "|"))      -- output: 1|3|5|hello
print(table.concat(a, " ", 4, 2)) -- output:
print(table.concat(a, " ", 2, 4)) -- output: 3 5 hello

```

table.insert (table, [pos ,] value)

在（数组型）表 **table** 的 **pos** 索引位置插入 **value**，其它元素向后移动到空的地方。**pos** 的默认值是表的长度加一，即默认是插在表的最后。

示例代码

```

local a = {1, 8}          --a[1] = 1,a[2] = 8
table.insert(a, 1, 3)    --在表索引为1处插入3
print(a[1], a[2], a[3])
table.insert(a, 10)       --在表的最后插入10
print(a[1], a[2], a[3], a[4])

-->output
3     1     8
3     1     8     10

```

table.maxn (table)

返回（数组型）表 **table** 的最大索引编号；如果此表没有正的索引编号，返回 0。

当长度省略时，此函数通常需要 $O(n)$ 的时间复杂度来计算 **table** 的末尾。因此用这个函数省略索引位置的调用形式来作 **table** 元素的末尾追加，是高代价操作。

示例代码

```

local a = {}
a[-1] = 10
print(table.maxn(a))
a[5] = 10
print(table.maxn(a))

-->output
0
5

```

此函数的行为不同于 `#` 运算符，因为 `#` 可以返回数组中任意一个 `nil` 空洞或最后一个 `nil` 之前的元素索引。当然，该函数的开销相比 `#` 运算符也会更大一些。

table.remove (table [, pos])

在表 `table` 中删除索引为 `pos` (`pos` 只能是 `number` 型) 的元素，并返回这个被删除的元素，它后面所有元素的索引值都会减一。`pos` 的默认值是表的长度，即默认是删除表的最后一个元素。

示例代码

```
local a = { 1, 2, 3, 4}
print(table.remove(a, 1)) --删除索引为1的元素
print(a[1], a[2], a[3], a[4])

print(table.remove(a)) --删除最后一个元素
print(a[1], a[2], a[3], a[4])

-->output
1
2     3     4     nil
4
2     3     nil     nil
```

table.sort (table [, comp])

按照给定的比较函数 `comp` 给表 `table` 排序，也就是从 `table[1]` 到 `table[n]`，这里 `n` 表示 `table` 的长度。比较函数有两个参数，如果希望第一个参数排在第二个的前面，就应该返回 `true`，否则返回 `false`。如果比较函数 `comp` 没有给出，默认从小到大排序。

示例代码

```
local function compare(x, y) --从大到小排序
    return x > y           --如果第一个参数大于第二个就返回true，否则返回false
end

local a = { 1, 7, 3, 4, 25}
table.sort(a)             --默认从小到大排序
print(a[1], a[2], a[3], a[4], a[5])
table.sort(a, compare)   --使用比较函数进行排序
print(a[1], a[2], a[3], a[4], a[5])

-->output
1     3     4     7     25
25     7     4     3     1
```

table 其他非常有用的函数

Luajit 2.1 新增加的 `table.new` 和 `table.clear` 函数是非常有用的。前者主要用来预分配 Lua `table` 空间，后者主要用来高效的释放 `table` 空间，并且它们都是可以被 JIT 编译的。具体可以参考一下 OpenResty 捆绑的 `lua-resty-*` 库，里面有些实例可以作为参考。

日期时间函数

在 Lua 中，函数 `time`、`date` 和 `difftime` 提供了所有的日期和时间功能。

在 OpenResty 的世界里，不推荐使用这里的标准时间函数，因为这些函数通常会引发不止一个昂贵的系统调用，同时无法为 `LuaJIT JIT` 编译，对性能造成较大影响。推荐使用 `ngx_lua` 模块提供的带缓存的时间接口，如 `ngx.today`，`ngx.time`，`ngx.utctime`，`ngx.localtime`，`ngx.now`，`ngx.http_time`，以及 `ngx.cookie_time` 等。

所以下面的部分函数，简单了解一下即可。

`os.time ([table])`

如果不使用参数 `table` 调用 `time` 函数，它会返回当前的时间和日期（它表示从某一时刻到现在的秒数）。如果用 `table` 参数，它会返回一个数字，表示该 `table` 中所描述的日期和时间（它表示从某一时刻到 `table` 中描述日期和时间的秒数）。`table` 的字段如下：

字段名称	取值范围
<code>year</code>	四位数字
<code>month</code>	1--12
<code>day</code>	1--31
<code>hour</code>	0--23
<code>min</code>	0--59
<code>sec</code>	0--61
<code>isdst</code>	<code>boolean</code> (<code>true</code> 表示夏令时)

对于 `time` 函数，如果参数为 `table`，那么 `table` 中必须含有 `year`、`month`、`day` 字段。其他字段省时段默认为中午（12:00:00）。

示例代码：（地点为北京）

```
print(os.time())    -->output 1438243393
a = { year = 1970, month = 1, day = 1, hour = 8, min = 1 }
print(os.time(a))  -->output 60
```

`os.difftime (t2, t1)`

返回 `t1` 到 `t2` 的时间差，单位为秒。

示例代码：

```
local day1 = { year = 2015, month = 7, day = 30 }
local t1 = os.time(day1)

local day2 = { year = 2015, month = 7, day = 31 }
local t2 = os.time(day2)
print(os.difftime(t2, t1)) -->output 86400
```

os.date ([format [, time]])

把一个表示日期和时间的数值，转换成更高级的表现形式。其第一个参数 **format** 是一个格式化字符串，描述了要返回的时间形式。第二个参数 **time** 就是日期和时间的数字表示，缺省时默认为当前的时间。使用格式字符 "*t"，创建一个时间表。

示例代码：

```
local tab1 = os.date("*t") --返回一个描述当前日期和时间的表
local ans1 = "{"
for k, v in pairs(tab1) do --把tab1转换成一个字符串
    ans1 = string.format("%s %s = %s,", ans1, k, tostring(v))
end

ans1 = ans1 .. "}"
print("tab1 = ", ans1)

local tab2 = os.date("*t", 360) --返回一个描述日期和时间数为360秒的表
local ans2 = "{"
for k, v in pairs(tab2) do --把tab2转换成一个字符串
    ans2 = string.format("%s %s = %s,", ans2, k, tostring(v))
end

ans2 = ans2 .. "}"
print("tab2 = ", ans2)

-->output
tab1 = { hour = 17, min = 28, wday = 5, day = 30, month = 7, year = 2015, sec = 10, yday = 211, isdst = false,}
tab2 = { hour = 8, min = 6, wday = 5, day = 1, month = 1, year = 1970, sec = 0, yday = 1, isdst = false,}
```

该表中除了使用到了 **time** 函数参数 **table** 的字段外，这还提供了星期 (**wday**，星期天为 1) 和一年中的第几天 (**yday**，一月一日为 1)。除了使用 "*t" 格式字符串外，如果使用带标记（见下表）的特殊字符串，**os.date** 函数会将相应的标记位以时间信息进行填充，得到一个包含时间的字符串。表如下：

格式字符	含义
%a	一星期中天数的简写（例如：Wed）
%A	一星期中天数的全称（例如：Wednesday）
%b	月份的简写（例如：Sep）
%B	月份的全称（例如：September）
%c	日期和时间（例如：07/30/15 16:57:24）
%d	一个月中的第几天[01 ~ 31]
%H	24小时制中的小时数[00 ~ 23]
%I	12小时制中的小时数[01 ~ 12]
%j	一年中的第几天[001 ~ 366]
%M	分钟数[00 ~ 59]
%m	月份数[01 ~ 12]
%p	“上午（am）”或“下午（pm）”
%S	秒数[00 ~ 59]
%w	一星期中的第几天[1 ~ 7 = 星期天 ~ 星期六]
%x	日期（例如：07/30/15）
%X	时间（例如：16:57:24）
%y	两位数的年份[00 ~ 99]
%Y	完整的年份（例如：2015）
%%	字符'%'

示例代码：

```
print(os.date("today is %A, in %B"))
print(os.date("now is %x %X"))

-->output
today is Thursday, in July
now is 07/30/15 17:39:22
```

数学库

Lua 数学库由一组标准的数学函数构成。数学库的引入丰富了 Lua 编程语言的功能，同时也方便了程序的编写。常用数学函数见下表：

函数名	函数功能
math.rad(x)	角度x转换成弧度
math.deg(x)	弧度x转换成角度
math.max(x, ...)	返回参数中值最大的那个数，参数必须是number型
math.min(x, ...)	返回参数中值最小的那个数，参数必须是number型
math.random ([m [, n]])	不传入参数时，返回一个在区间[0,1)内均匀分布的伪随机实数；只使用一个整数参数m时，返回一个在区间[1, m]内均匀分布的伪随机整数；使用两个整数参数时，返回一个在区间[m, n]内均匀分布的伪随机整数
math.randomseed (x)	为伪随机数生成器设置一个种子x，相同的种子将会生成相同的数字序列
math.abs(x)	返回x的绝对值
math.fmod(x, y)	返回 x对y取余数
math.pow(x, y)	返回x的y次方
math.sqrt(x)	返回x的算术平方根
math.exp(x)	返回自然数e的x次方
math.log(x)	返回x的自然对数
math.log10(x)	返回以10为底，x的对数
math.floor(x)	返回最大且不大于x的整数
math.ceil(x)	返回最小且不小于x的整数
math.pi	圆周率
math.sin(x)	求弧度x的正弦值
math.cos(x)	求弧度x的余弦值
math.tan(x)	求弧度x的正切值
math.asin(x)	求x的反正弦值
math.acos(x)	求x的反余弦值
math.atan(x)	求x的反正切值

示例代码：

```
print(math.pi)          -->output 3.1415926535898
print(math.rad(180))    -->output 3.1415926535898
print(math.deg(math.pi)) -->output 180

print(math.sin(1))      -->output 0.8414709848079
print(math.cos(math.pi)) -->output -1
print(math.tan(math.pi / 4)) -->output 1

print(math.atan(1))     -->output 0.78539816339745
print(math.asin(0))      -->output 0

print(math.max(-1, 2, 0, 3.6, 9.1))   -->output 9.1
print(math.min(-1, 2, 0, 3.6, 9.1))   -->output -1

print(math.fmod(10.1, 3))   -->output 1.1
print(math.sqrt(360))       -->output 18.97366596101

print(math.exp(1))          -->output 2.718281828459
print(math.log(10))         -->output 2.302585092994
print(math.log10(10))        -->output 1

print(math.floor(3.1415))   -->output 3
print(math.ceil(7.998))     -->output 8
```

另外使用 `math.random()` 函数获得伪随机数时，如果不使用 `math.randomseed()` 设置伪随机数生成种子或者设置相同的伪随机数生成种子，那么得得到的伪随机数序列是一样的。

示例代码：

```
math.randomseed (100) --把种子设置为100
print(math.random())      -->output 0.0012512588885159
print(math.random(100))    -->output 57
print(math.random(100, 360)) -->output 150
```

稍等片刻，再次运行上面的代码。

```
math.randomseed (100) --把种子设置为100
print(math.random())      -->output 0.0012512588885159
print(math.random(100))    -->output 57
print(math.random(100, 360)) -->output 150
```

两次运行的结果一样。为了避免每次程序启动时得到的都是相同的伪随机数序列，通常是使用当前时间作为种子。

修改上例中的代码：

```
math.randomseed (os.time())      --把100换成os.time()
print(math.random())            -->output 0.88369396038697
print(math.random(100))          -->output 66
print(math.random(100, 360))     -->output 228
```

稍等片刻，再次运行上面的代码。

```
math.randomseed (os.time())      --把100换成os.time()
print(math.random())            -->output 0.88946195867794
print(math.random(100))          -->output 68
print(math.random(100, 360))     -->output 129
```

文件操作

Lua I/O 库提供两种不同的方式处理文件：隐式文件描述，显式文件描述。

这些文件 I/O 操作，在 OpenResty 的上下文中对事件循环是会产生阻塞效应。OpenResty 比较擅长的是高并发网络处理，在这个环境中，任何文件的操作，都将阻塞其他并行执行的请求。实际中的应用，在 OpenResty 项目中应尽可能让网络处理部分、文件 I/O 操作部分相互独立，不要揉和在一起。

隐式文件描述

设置一个默认的输入或输出文件，然后在这个文件上进行所有的输入或输出操作。所有的操作函数由 `io` 表提供。

打开已经存在的 `test1.txt` 文件，并读取里面的内容

```
file = io.input("test1.txt")      -- 使用 io.input() 函数打开文件

repeat
    line = io.read()           -- 逐行读取内容，文件结束时返回nil
    if nil == line then
        break
    end
    print(line)
until (false)

io.close(file)                  -- 关闭文件

--> output
my test file
hello
lua
```

在 `test1.txt` 文件的最后添加一行 "hello world"

```
file = io.open("test1.txt", "a+")   -- 使用 io.open() 函数，以添加模式打开文件
io.output(file)                   -- 使用 io.output() 函数，设置默认输出文件
io.write("\nhello world")         -- 使用 io.write() 函数，把内容写到文件
io.close(file)
```

在相应目录下打开 `test1.txt` 文件，查看文件内容发生的变化。

显式文件描述

使用 `file:XXX()` 函数方式进行操作，其中 `file` 为 `io.open()` 返回的文件句柄。

打开已经存在的 `test2.txt` 文件，并读取里面的内容

```
file = io.open("test2.txt", "r")      -- 使用 io.open() 函数，以只读模式打开文件

for line in file:lines() do          -- 使用 file:lines() 函数逐行读取文件
    print(line)
end

file:close()

-->output
my test2
hello lua
```

在 `test2.txt` 文件的最后添加一行 "hello world"

```
file = io.open("test2.txt", "a")      -- 使用 io.open() 函数，以添加模式打开文件
file:write("\nhello world")         -- 使用 file:write() 函数，在文件末尾追加内容
file:close()
```

在相应目录下打开 `test2.txt` 文件，查看文件内容发生的变化。

文件操作函数

`io.open (filename [, mode])`

按指定的模式 `mode`，打开一个文件名为 `filename` 的文件，成功则返回文件句柄，失败则返回 `nil` 加错误信息。模式：

模式	含义	文件不存在时
"r"	读模式 (默认)	返回 <code>nil</code> 加错误信息
"w"	写模式	创建文件
"a"	添加模式	创建文件
"r+"	更新模式，保存之前的数据	返回 <code>nil</code> 加错误信息
"w+"	更新模式，清除之前的数据	创建文件
"a+"	添加更新模式，保存之前的数据,在文件尾进行添加	创建文件

模式字符串后面可以有一个 'b'，用于在某些系统中打开二进制文件。

注意 "w" 和 "wb" 的区别

- "w" 表示文本文件。某些文件系统(如 Linux 的文件系统)认为 0x0A 为文本文件的换行符，Windows 的文件系统认为 0x0D0A 为文本文件的换行符。为了兼容其他文件系统(如从 Linux 拷贝来的文件)，Windows 的文件系统在写文件时，会在文件中 0x0A 的前面加上 0x0D。使用 "w"，其属性要看所在的平台。
- "wb" 表示二进制文件。文件系统会按纯粹的二进制格式进行写操作，因此也就不存在格式转换的问题。(Linux 文件系统下 "w" 和 "wb" 没有区别)

file:close ()

关闭文件。注意：当文件句柄被垃圾收集后，文件将自动关闭。句柄将变为一个不可预知的值。

io.close ([file])

关闭文件，和 file:close() 的作用相同。没有参数 file 时，关闭默认输出文件。

file:flush ()

把写入缓冲区的所有数据写入到文件 file 中。

io.flush ()

相当于 file:flush()，把写入缓冲区的所有数据写入到默认输出文件。

io.input ([file])

当使用一个文件名调用时，打开这个文件（以文本模式），并设置文件句柄为默认输入文件；当使用一个文件句柄调用时，设置此文件句柄为默认输入文件；当不使用参数调用时，返回默认输入文件句柄。

file:lines ()

返回一个迭代函数，每次调用将获得文件中的一行内容，当到文件尾时，将返回 nil，但不关闭文件。

io.lines ([filename])

打开指定的文件 filename 为读模式并返回一个迭代函数，每次调用将获得文件中的一行内容，当到文件尾时，将返回 nil，并自动关闭文件。若不带参数时 io.lines() 等价于 io.input():lines() 读取默认输入设备的内容，结束时不关闭文件。

io.output ([file])

类似于 `io.input`，但操作在默认输出文件上。

file:read (...)

按指定的格式读取一个文件。按每个格式将返回一个字符串或数字，如果不能正确读取将返回 `nil`，若没有指定格式将指默认按行方式进行读取。格式：

格式	含义
"*n"	读取一个数字
"*a"	从当前位置读取整个文件。若当前位置为文件尾，则返回空字符串
"*l"	读取下一行的内容。若为文件尾，则返回 <code>nil</code> 。(默认)
number	读取指定字节数的字符。若为文件尾，则返回 <code>nil</code> 。如果 <code>number</code> 为 0，则返回空字符串，若为文件尾，则返回 <code>nil</code>

io.read (...)

相当于 `io.input():read`

io.type (obj)

检测 `obj` 是否一个可用的文件句柄。如果 `obj` 是一个打开的文件句柄，则返回 "file" 如果 `obj` 是一个已关闭的文件句柄，则返回 "closed file" 如果 `obj` 不是一个文件句柄，则返回 `nil`。

file:write (...)

把每一个参数的值写入文件。参数必须为字符串或数字，若要输出其它值，则需通过 `tostring` 或 `string.format` 进行转换。

io.write (...)

相当于 `io.output():write`。

file:seek ([whence] [, offset])

设置和获取当前文件位置，成功则返回最终的文件位置(按字节，相对于文件开头)，失败则返回 `nil` 加错误信息。缺省时，`whence` 默认为 "cur"，`offset` 默认为 0。参数 `whence`：

whence	含义
"set"	文件开始
"cur"	文件当前位置(默认)
"end"	文件结束

file:setvbuf (mode [, size])

设置输出文件的缓冲模式。模式：

模式	含义
"no"	没有缓冲，即直接输出
"full"	全缓冲，即当缓冲满后才进行输出操作(也可调用flush马上输出)
"line"	以行为单位，进行输出

最后两种模式，size 可以指定缓冲的大小（按字节），忽略 size 将自动调整为最佳的大小。

元表

在 Lua 5.1 语言中，元表 (*metatable*) 的表现行为类似于 C++ 语言中的操作符重载，例如我们可以重载 "__add" 元方法 (*metamethod*)，来计算两个 Lua 数组的并集；或者重载 "__index" 方法，来定义我们自己的 Hash 函数。Lua 提供了两个十分重要的用来处理元表的方法，如下：

- **setmetatable(table, metatable)**：此方法用于为一个表设置元表。
- **getmetatable(table)**：此方法用于获取表的元表对象。

设置元表的方法很简单，如下：

```
local mytable = {}
local mymetatable = {}
setmetatable(mytable, mymetatable)
```

上面的代码可以简写成如下的一行代码：

```
local mytable = setmetatable({}, {})
```

修改表的操作符行为

通过重载 "__add" 元方法来计算集合的并集实例：

```

local set1 = {10, 20, 30} -- 集合
local set2 = {20, 40, 50} -- 集合

-- 将用于重载__add的函数，注意第一个参数是self
local union = function (self, another)
    local set = {}
    local result = {}

    -- 利用数组来确保集合的互异性
    for i, j in pairs(self) do set[j] = true end
    for i, j in pairs(another) do set[j] = true end

    -- 加入结果集合
    for i, j in pairs(set) do table.insert(result, i) end
    return result
end

setmetatable(set1, {__add = union}) -- 重载 set1 表的 __add 元方法

local set3 = set1 + set2
for _, j in pairs(set3) do
    io.write(j.." ")           -->output : 30 50 20 40 10
end

```

除了加法可以被重载之外，Lua 提供的所有操作符都可以被重载：

元方法	含义
"__add"	+ 操作
"__sub"	- 操作 其行为类似于 "add" 操作
"__mul"	* 操作 其行为类似于 "add" 操作
"__div"	/ 操作 其行为类似于 "add" 操作
"__mod"	% 操作 其行为类似于 "add" 操作
"__pow"	^ (幂) 操作 其行为类似于 "add" 操作
"__unm"	一元 - 操作
"__concat"	.. (字符串连接) 操作
"__len"	# 操作
"__eq"	== 操作 函数 getcomphandler 定义了 Lua 怎样选择一个处理器来作比较操作 仅在两个对象类型相同且有对应操作相同的元方法时才起效
"__lt"	< 操作
"__le"	<= 操作

除了操作符之外，如下元方法也可以被重载，下面会依次解释使用方法：

元方法	含义
"__index"	取下标操作用于访问 <code>table[key]</code>
"__newindex"	赋值给指定下标 <code>table[key] = value</code>
"__tostring"	转换成字符串
"__call"	当 Lua 调用一个值时调用
"__mode"	用于弱表(<i>weak table</i>)
"__metatable"	用于保护 <code>metatable</code> 不被访问

__index 元方法

下面的例子中，我们实现了在表中查找键不存在时转而在元表中查找该键的功能：

```
mytable = setmetatable({key1 = "value1"}, --原始表
{__index = function(self, key)           --重载函数
  if key == "key2" then
    return "metatablevalue"
  end
end
})

print(mytable.key1, mytable.key2) --> output : value1 metatablevalue
```

关于 `__index` 元方法，有很多比较高阶的技巧，例如：`__index` 的元方法不需要非是一个函数，他也可以是一个表。

```
t = setmetatable({[1] = "hello"}, {__index = {[2] = "world"}})
print(t[1], t[2]) -->hello world
```

第一句代码有点绕，解释一下：先是把 `{__index = {}}` 作为元表，但 `__index` 接受一个表，而不是函数，这个表中包含 `[2] = "world"` 这个键值对。所以当 `t[2]` 去在自身的表中找不到时，在 `__index` 的表中去寻找，然后找到了 `[2] = "world"` 这个键值对。

`__index` 元方法还可以实现给表中每一个值赋上默认值；和 `__newindex` 元方法联合监控对表的读取、修改等比较高阶的功能，待读者自己去开发吧。

__tostring 元方法

与 Java 中的 `toString()` 函数类似，可以实现自定义的字符串转换。

```

arr = {1, 2, 3, 4}
arr = setmetatable(arr, {__tostring = function (self)
    local result = '{'
    local sep = ''
    for _, i in pairs(self) do
        result = result .. sep .. i
        sep = ', '
    end
    result = result .. '}'
    return result
end})
print(arr) --> {1, 2, 3, 4}

```

call 元方法

call 元方法的功能类似于 C++ 中的仿函数，使得普通的表也可以被调用。

```

functor = {}
function func1(self, arg)
    print ("called from", arg)
end

setmetatable(functor, {__call = func1})

functor("functor") --> called from functor
print(functor)      --> output: 0x00076fc8 (后面这串数字可能不一样)

```

metatable 元方法

假如我们想保护我们的对象使其使用者既看不到也不能修改 metatables。我们可以对 **metatable** 设置了 metatable 的值，**getmetatable** 将返回这个域的值，而调用 **setmetatable** 将会出错：

```

Object = setmetatable({}, {__metatable = "You cannot access here"})

print(getmetatable(Object)) --> You cannot access here
setmetatable(Object, {})    --> 引发编译器报错

```

Lua 面向对象编程

类

在 Lua 中，我们可以使用表和函数实现面向对象。将函数和相关的数据放置于同一个表中就形成了一个对象。

请看文件名为 `account.lua` 的源码：

```
local _M = {}

local mt = { __index = _M }

function _M.deposit (self, v)
    self.balance = self.balance + v
end

function _M.withdraw (self, v)
    if self.balance > v then
        self.balance = self.balance - v
    else
        error("insufficient funds")
    end
end

function _M.new (self, balance)
    balance = balance or 0
    return setmetatable({balance = balance}, mt)
end

return _M
```

引用代码示例：

```
local account = require("account")

local a = account:new()
a:deposit(100)

local b = account:new()
b:deposit(50)

print(a.balance) --> output: 100
print(b.balance) --> output: 50
```

上面这段代码 "setmetatable({balance = balance}, mt)"，其中 `mt` 代表 `{ __index = _M }`，这句话值得注意。根据我们在元表这一章学到的知识，我们明白，`setmetatable` 将 `_M` 作为新建表的原型，所以在自己的表内找不到 '`deposit`'、'`withdraw`' 这些方法和变量的时候，便会到 `__index` 所指定的 `_M` 类型中去寻找。

继承

继承可以用元表实现，它提供了在父类中查找存在的方法和变量的机制。在 `Lua` 中是不推荐使用继承方式完成构造的，这样做引入的问题可能比解决的问题要多，下面一个是字符串操作类库，给大家演示一下。

```
----- s_base.lua
local _M = {}

local mt = { __index = _M }

function _M.upper (s)
    return string.upper(s)
end

return _M

----- s_more.lua
local s_base = require("s_base")

local _M = {}
_M = setmetatable(_M, { __index = s_base })

function _M.lower (s)
    return string.lower(s)
end

return _M

----- test.lua
local s_more = require("s_more")

print(s_more.upper("Hello")) -- output: HELLO
print(s_more.lower("Hello")) -- output: hello
```

成员私有性

在动态语言中引入成员私有性并没有太大的必要，反而会显著增加运行时的开销，毕竟这种检查无法像许多静态语言那样在编译期完成。下面的技巧把对象作为各方法的 `upvalue`，本身是很巧妙的，但会让子类继承变得困难，同时构造函数动态创建了函数，会导致构造函数无法被 `JIT` 编译。

在 **Lua** 中，成员的私有性，使用类似于函数闭包的形式来实现。在我们之前的银行账户的例子中，我们使用一个工厂方法来创建新的账户实例，通过工厂方法对外提供的闭包来暴露对外接口。而不想暴露在外的例如 **balance** 成员变量，则被很好的隐藏起来。

```
function newAccount (initialBalance)
    local self = {balance = initialBalance}
    local withdraw = function (v)
        self.balance = self.balance - v
    end
    local deposit = function (v)
        self.balance = self.balance + v
    end
    local getBalance = function () return self.balance end
    return {
        withdraw = withdraw,
        deposit = deposit,
        getBalance = getBalance
    }
end

a = newAccount(100)
a.deposit(100)
print(a.getBalance()) --> 200
print(a.balance)      --> nil
```

局部变量

Lua 的设计有一点很奇怪，在一个 **block** 中的变量，如果之前没有定义过，那么认为它是一个全局变量，而不是这个 **block** 的局部变量。这一点和别的语言不同。容易造成不小心覆盖了全局同名变量的错误。

定义

Lua 中的局部变量要用 **local** 关键字来显式定义，不使用 **local** 显式定义的变量就是全局变量：

```
g_var = 1          -- global var
local l_var = 2    -- local var
```

作用域

局部变量的生命周期是有限的，它的作用域仅限于声明它的块（**block**）。一个块是一个控制结构的执行体、或者是一个函数的执行体再或者是一个程序块（**chunk**）。我们可以通过下面这个例子来理解一下局部变量作用域的问题：

示例代码 test.lua

```
x = 10
local i = 1          -- 程序块中的局部变量 i

while i <= x do
    local x = i * 2  -- while 循环体中的局部变量 x
    print(x)          -- output: 2, 4, 6, 8, ...
    i = i + 1
end

if i > 20 then
    local x          -- then 中的局部变量 x
    x = 20
    print(x + 2)    -- 如果 i > 20 将会打印 22，此处的 x 是局部变量
else
    print(x)          -- 打印 10，这里 x 是全局变量
end

print(x)            -- 打印 10
```

使用局部变量的好处

1. 局部变量可以避免因为命名问题污染了全局环境
2. `local` 变量的访问比全局变量更快
3. 由于局部变量出了作用域之后生命周期结束，这样可以被垃圾回收器及时释放

常见实现如：`local print = print`

在 `Lua` 中，应该尽量让定义变量的语句靠近使用变量的语句，这也可以被看做是一种良好的编程风格。在 `C` 这样的语言中，强制程序员在一个块（或一个过程）的起始处声明所有的局部变量，所以有些程序员认为在一个块的中间使用声明语句是一种不良好的习惯。实际上，在需要时才声明变量并且赋予有意义的初值，这样可以提高代码的可读性。对于程序员而言，相比在块中的任意位置顺手声明自己需要的变量，和必须跳到块的起始处声明，大家应该能掂量哪种做法比较方便了吧？

“尽量使用局部变量”是一种良好的编程风格。然而，初学者在使用 `Lua` 时，很容易忘记加上 `local` 来定义局部变量，这时变量就会自动变成全局变量，很可能导致程序出现意想不到的问题。那么我们怎么检测哪些变量是全局变量呢？我们如何防止全局变量导致的影响呢？下面给出一段代码，利用元表的方式来自动检查全局变量，并打印必要的调试信息：

检查模块的函数使用全局变量

把下面代码保存在 `foo.lua` 文件中。

```
local _M = { _VERSION = '0.01' }

function _M.add(a, b)      -- 两个number型变量相加
    return a + b
end

function _M.update_A()     -- 更新变量值
    A = 365
end

return _M
```

把下面代码保存在 `use_foo.lua` 文件中。该文件和 `foo.lua` 在相同目录。

```
A = 360      -- 定义全局变量
local foo = require("foo")

local b = foo.add(A, A)
print("b = ", b)

foo.update_A()
print("A = ", A)
```

输出结果：

```
# luajit use_foo.lua
b = 720
A = 365
```

无论是做基础模块或是上层应用，肯定都不愿意存在这类灰色情况存在，因为它对我们系统的存在，带来很多不确定性（注意 OpenResty 会限制请求过程中全局变量的使用）。生产中我们是要尽力避免这种情况的出现。

Lua 上下文中应当严格避免使用自己定义的全局变量。可以使用一个 **lj-releng** 工具来扫描 **Lua** 代码，定位使用 **Lua** 全局变量的地方。**lj-releng** 的相关链接：<https://github.com/openresty/openresty-devel-utils/blob/master/lj-releng>

如果使用 **macOS** 或者 **Linux**，可以使用下面命令安装 **lj-releng**：

```
curl -L https://github.com/openresty/openresty-devel-utils/blob/master/lj-releng > /usr/local/bin/lj-releng
chmod +x /usr/local/bin/lj-releng
```

Windows 用户把 **lj-releng** 文件所在的目录的绝对路径添加进 **PATH** 环境变量。然后进入你自己的 **Lua** 文件所在的工作目录，得到如下结果：

```
# lj-releng
foo.lua: 0.01 (0.01)
Checking use of Lua global variables in file foo.lua...
op no.  line  instruction args ; code
2 [8] SETGLOBAL 0 -1 ; A
Checking line length exceeding 80...
WARNING: No "_VERSION" or "version" field found in `use_foo.lua`.
Checking use of Lua global variables in file use_foo.lua...
op no.  line  instruction args ; code
2 [1] SETGLOBAL 0 -1 ; A
7 [4] GETGLOBAL 2 -1 ; A
8 [4] GETGLOBAL 3 -1 ; A
18 [8] GETGLOBAL 4 -1 ; A
Checking line length exceeding 80...
```

结果显示：在 **foo.lua** 文件中，第 8 行设置了一个全局变量 **A**；在 **use_foo.lua** 文件中，没有版本信息，并且第 1 行设置了一个全局变量 **A**，第 4、8 行使用了全局变量 **A**。

当然，更推荐采用 **luacheck** 来检查项目中全局变量，见 [代码静态分析](#) 一节的内容。

判断数组大小

`table.getn(t)` 等价于 `#t` 但计算的是数组元素，不包括 `hash` 键值。而且数组是以第一个 `nil` 元素来判断数组结束。`#` 只计算 `array` 的元素个数，它实际上调用了对象的 `metatable` 的 `_len` 函数。对于有 `_len` 方法的函数返回函数返回值，不然就返回数组成员数目。

`Lua` 中，数组的实现方式其实类似于 `C++` 中的 `map`，对于数组中所有的值，都是以键值对的形式来存储（无论是显式还是隐式），`Lua` 内部实际采用哈希表和数组分别保存键值对、普通值，所以不推荐混合使用这两种赋值方式。尤其需要注意的一点是：`Lua` 数组中允许 `nil` 值的存在，但是数组默认结束标志却是 `nil`。这类似与 `C` 语言中的字符串，字符串中允许 '`\0`' 存在，但当读到 '`\0`' 时，就认为字符串已经结束了。

初始化是例外，在 `Lua` 相关源码中，初始化数组时首先判断数组的长度，若长度大于 0，并且最后一个值不为 `nil`，返回包括 `nil` 的长度；若最后一个值为 `nil`，则返回截至第一个非 `nil` 值的长度。

注意：一定不要使用 `#` 操作符或 `table.getn` 来计算包含 `nil` 的数组长度，这是一个未定义的操作，不一定报错，但不能保证结果如你所想。如果你要删除一个数组中的元素，请使用 `remove` 函数，而不是用 `nil` 赋值。

```
-- test.lua
local tblTest1 = { 1, a = 2, 3 }
print("Test1 " .. #(tblTest1))

local tblTest2 = { 1, nil }
print("Test2 " .. #(tblTest2))

local tblTest3 = { 1, nil, 2 }
print("Test3 " .. #(tblTest3))

local tblTest4 = { 1, nil, 2, nil }
print("Test4 " .. #(tblTest4))

local tblTest5 = { 1, nil, 2, nil, 3, nil }
print("Test5 " .. #(tblTest5))

local tblTest6 = { 1, nil, 2, nil, 3, nil, 4, nil }
print("Test6 " .. #(tblTest6))
```

我们分别使用 `Lua` 和 `LuaJIT` 来执行一下：

```
→ luajit test.lua
Test1 2
Test2 1
Test3 1
Test4 1
Test5 1
Test6 1

→ lua test.lua
Test1 2
Test2 1
Test3 3
Test4 1
Test5 3
Test6 1
```

这一段的输出结果，就是这么匪夷所思。不要在 Lua 的 `table` 中使用 `nil` 值，如果一个元素要删除，直接 `remove`，不要用 `nil` 去代替。

非空判断

大家在使用 **Lua** 的时候，一定会遇到不少和 **nil** 有关的坑吧。有时候不小心引用了一个没有赋值的变量，这时它的值默认为 **nil**。如果对一个 **nil** 进行索引的话，会导致异常。

如下：

```
local person = {name = "Bob", sex = "M"}

-- do something
person = nil
-- do something

print(person.name)
```

上面这个例子把 **nil** 的错误用法显而易见地展示出来，执行后，会提示下面的错误：

```
stdin:1:attempt to index global 'person' (a nil value)
stack traceback:
  stdin:1: in main chunk
[C]: ?
```

然而，在实际的工程代码中，我们很难这么轻易地发现我们引用了 **nil** 变量。因此，在很多情况下我们在访问一些 **table** 型变量时，需要先判断该变量是否为 **nil**，例如将上面的代码改成：

```
local person = {name = "Bob", sex = "M"}

-- do something
person = nil
-- do something
if person ~= nil and person.name ~= nil then
  print(person.name)
else
  -- do something
end
```

对于简单类型的变量，我们可以用 ***if (var == nil) then*** 这样的简单句子来判断。但是对于 **table** 型的 **Lua** 对象，就不能这么简单判断它是否为空了。一个 **table** 型变量的值可能是 **{}**，这时它不等于 **nil**。我们来看下面这段代码：

```

local next = next
local a = {}
local b = {name = "Bob", sex = "Male"}
local c = {"Male", "Female"}
local d = nil

print(#a)
print(#b)
print(#c)
--print(#d)    -- error

if a == nil then
    print("a == nil")
end

if b == nil then
    print("b == nil")
end

if c == nil then
    print("c == nil")
end

if d == nil then
    print("d == nil")
end

if next(a) == nil then
    print("next(a) == nil")
end

if next(b) == nil then
    print("next(b) == nil")
end

if next(c) == nil then
    print("next(c) == nil")
end

```

返回的结果如下：

```

0
0
2
d == nil
next(a) == nil

```

因此，我们要判断一个 **table** 是否为 `{}`，不能采用 `#table == 0` 的方式来判断。可以用下面这样的方法来判断：

```
function isEmpty(t)
    return t == nil or next(t) == nil
end
```

注意：`next` 指令是不能被 LuaJIT 的 JIT 编译优化，并且 LuaJIT 貌似没有明确计划支持这个指令优化，在不是必须的情况下，尽量少用。

正则表达式

在 OpenResty 中，同时存在两套正则表达式规范：*Lua* 语言的规范和 `ngx.re.*` 的规范，即使您对 *Lua* 语言中的规范非常熟悉，我们仍不建议使用 *Lua* 中的正则表达式。一是因为 *Lua* 中正则表达式的性能并不如 `ngx.re.*` 中的正则表达式优秀；二是 *Lua* 中的正则表达式并不符合 *POSIX* 规范，而 `ngx.re.*` 中实现的是标准的 *POSIX* 规范，后者明显更具备通用性。

Lua 中的正则表达式与 Nginx 中的正则表达式相比，有 5% - 15% 的性能损失，而且 *Lua* 将表达式编译成 *Pattern* 之后，并不会将 *Pattern* 缓存，而是每此使用都重新编译一遍，潜在地降低了性能。`ngx.re.*` 中的正则表达式可以通过参数缓存编译过后的 *Pattern*，不会有类似的性能损失。

`ngx.re.*` 中的 `o` 选项，指明该参数，被编译的 *Pattern* 将会在工作进程中缓存，并且被当前工作进程的每次请求所共享。*Pattern* 缓存的上限值通过 `lua_regex_cache_max_entries` 来修改。

`ngx.re.*` 中的 `j` 选项，指明该参数，如果使用的 PCRE 库支持 JIT，OpenResty 会在编译 *Pattern* 时启用 JIT。启用 JIT 后正则匹配会有明显的性能提升。较新的平台，自带的 PCRE 库均支持 JIT。如果系统自带的 PCRE 库不支持 JIT，出于性能考虑，最好自己编译一份 `libpcre.so`，然后在编译 OpenResty 时链接过去。要想验证当前 PCRE 库是否支持 JIT，可以这么做

1. 编译 OpenResty 时在 `./configure` 中指定 `--with-debug` 选项
2. 在 `error_log` 指令中指定日志级别为 `debug`
3. 运行正则匹配代码，查看日志中是否有 `pcre JIT compiling result: 1`

即使运行在不支持 JIT 的 OpenResty 上，加上 `j` 选项也不会带来坏的影响。在 OpenResty 官方的 *Lua* 库中，正则匹配至少都会带上 `jo` 这两个选项。

```
location /test {
    content_by_lua_block {
        local regex = [[\d+]]

        -- 参数 "j" 启用 JIT 编译，参数 "o" 是开启缓存必须的
        local m = ngx.re.match("hello, 1234", regex, "jo")
        if m then
            ngx.say(m[0])
        else
            ngx.say("not matched!")
        end
    }
}
```

测试结果如下：

```
→ ~ curl 127.0.0.1/test
1234
```

另外还可以试试引入 `lua-resty-core` 中的正则表达式 API。这么做需要在代码里加入 `require 'resty.core.regex'`。`lua-resty-core` 版本的 `ngx.re.*`，是通过 FFI 而非 Lua/C API 来跟 OpenResty C 代码交互的。某些情况下，会带来明显的性能提升。

Lua 正则简单汇总

Lua 中正则表达式语法上最大的区别，*Lua* 使用 '%' 来进行转义，而其他语言的正则表达式使用 '\' 符号来进行转义。其次，*Lua* 中并不使用 '?' 来表示非贪婪匹配，而是定义了不同的字符来表示是否是贪婪匹配。定义如下：

符号	匹配次数	匹配模式
+	匹配前一字符 1 次或多次	非贪婪
*	匹配前一字符 0 次或多次	贪婪
-	匹配前一字符 0 次或多次	非贪婪
?	匹配前一字符 0 次或 1 次	仅用于此，不用于标识是否贪婪

符号	匹配模式
.	任意字符
%a	字母
%c	控制字符
%d	数字
%l	小写字母
%p	标点字符
%s	空白符
%u	大写字母
%w	字母和数字
%x	十六进制数字
%z	代表 0 的字符

- `string.find` 的基本应用是在目标串内搜索匹配指定的模式的串。函数如果找到匹配的串，就返回它的开始索引和结束索引，否则返回 `nil`。`find` 函数第三个参数是可选的：标示目标串中搜索的起始位置，例如当我们想实现一个迭代器时，可以传进上一次调用时的结

束索引，如果返回了一个 *nil* 值的话，说明查找结束了。

```
local s = "hello world"
local i, j = string.find(s, "hello")
print(i, j) --> 1 5
```

- *string.gmatch* 我们也可以使用返回迭代器的方式。

```
local s = "hello world from Lua"
for w in string.gmatch(s, "%a+") do
    print(w)
end

-- output :
--     hello
--     world
--     from
--     Lua
```

- *string.gsub* 用来查找匹配模式的串，并将使用替换串其替换掉，但并不修改原字符串，而是返回一个修改后的字符串的副本，函数有目标串，模式串，替换串三个参数，使用范例如下：

```
local a = "Lua is cute"
local b = string.gsub(a, "cute", "great")
print(a) --> Lua is cute
print(b) --> Lua is great
```

- 还有一点值得注意的是，'*%b*' 用来匹配对称的字符，而不是一般正则表达式中的单词的开始、结束。'*%b*' 用来匹配对称的字符，而且采用贪婪匹配。常写为 '*%bxy*'，*x* 和 *y* 是任意两个不同的字符；*x* 作为匹配的开始，*y* 作为匹配的结束。比如，'*%b()*' 匹配以 '(' 开始，以 ')' 结束的字符串：

```
print(string.gsub("a (enclosed (in) parentheses) line", "%b()", ""))
-- output: a line 1
```

不用标准库

虚变量

当一个方法返回多个值时，有些返回值有时候用不到，要是声明很多变量来一一接收，显然不太合适（不是不能）。Lua 提供了一个虚变量(dummy variable)，以单个下划线（“_”）来命名，用它来丢弃不需要的数值，仅仅起到占位的作用。

看一段示例代码：

```
-- string.find (s,p) 从string 变量s的开头向后匹配 string
-- p，若匹配不成功，返回nil，若匹配成功，返回第一次匹配成功
-- 的起止下标。

local start, finish = string.find("hello", "he") --start 值为起始下标, finish
                                                --值为结束下标
print ( start, finish )                      --输出 1  2

local start = string.find("hello", "he")      -- start值为起始下标
print ( start )                            -- 输出 1

local _,finish = string.find("hello", "he")   --采用虚变量（即下划线），接收起
                                                --始下标值，然后丢弃，finish接收
                                                --结束下标值
print ( finish )                          --输出 2
```

代码倒数第二行，定义了一个用 `local` 修饰的 虚变量（即 单个下划线）。使用这个虚变量接收 `string.find()` 第一个返回值，静默丢掉，这样就直接得到第二个返回值了。

虚变量不仅仅可以被用在返回值，还可以用在迭代等。

在for循环中的使用：

```
-- test.lua 文件
local t = {1, 3, 5}

print("all  data:")
for i,v in ipairs(t) do
    print(i,v)
end

print("")
print("part data:")
for _,v in ipairs(t) do
    print(v)
end
```

执行结果：

```
# luajit test.lua
all  data:
1    1
2    3
3    5

part data:
1
3
5
```

抵制使用 `module()` 定义模块

旧式的模块定义方式是通过 `module("filename"[, package.seeall])*` 来显式声明一个包，现在官方不推荐再使用这种方式。这种方式将会返回一个由 `filename` 模块函数组成的 `table`，并且还会定义一个包含该 `table` 的全局变量。

`module("filename", package.seeall)` 这种写法是不提倡的，官方给出了两点原因：

1. `package.seeall` 这种方式破坏了模块的高内聚，原本引入 "filename" 模块只想调用它的 `foobar()` 函数，但是它却可以读写全局属性，例如 `"filename.os"`。
2. `module` 函数压栈操作引发的副作用，污染了全局环境变量。例如 `module("filename")` 会创建一个 `filename` 的 `table`，并将这个 `table` 注入全局环境变量中，这样使得没有引用它的文件也能调用 `filename` 模块的方法。

比较推荐的模块定义方法是：

```
-- square.lua 长方形模块
local _M = {}           -- 局部的变量
_M._VERSION = '1.0'      -- 模块版本

local mt = { __index = _M }

function _M.new(self, width, height)
    return setmetatable({ width=width, height=height }, mt)
end

function _M.get_square(self)
    return self.width * self.height
end

function _M.get_circumference(self)
    return (self.width + self.height) * 2
end

return _M
```

引用示例代码：

```
local square = require "square"

local s1 = square:new(1, 2)
print(s1:get_square())          --output: 2
print(s1:get_circumference())   --output: 6
```

另一个跟 Lua 的 `module` 模块相关需要注意的点是，当 `lua_code_cache on` 开启时，`require` 加载的模块是会被缓存下来的，这样我们的模块就会以最高效的方式运行，直到被显式地调用如下语句（这里有点像模块卸载）：

```
package.loaded["square"] = nil
```

我们可以利用这个特性代码来做一些高阶玩法，比如代码热更新等。

调用代码前先定义函数

`Lua` 里面的函数必须放在调用的代码之前，下面的代码是一个常见的错误：

```
-- test.lua 文件
local i = 100
i = add_one(i)

function add_one(i)
    return i + 1
end
```

我们将得到如下错误：

```
# luajit test.lua
luajit: test.lua:2: attempt to call global 'add_one' (a nil value)
stack traceback:
  test.lua:2: in main chunk
[C]: at 0x0100002150
```

为什么放在调用后面就找不到呢？原因是 `Lua` 里的 `function` 定义本质上是变量赋值，即

```
function foo() ... end
```

等价于

```
foo = function () ... end
```

因此在函数定义之前使用函数相当于在变量赋值之前使用变量，`Lua` 世界对于没有赋值的变量，默认都是 `nil`，所以这里也就产生了一个 `nil` 的错误。

一般地，由于全局变量是每个请求的生命期，因此以此种方式定义的函数的生命期也是每个请求的。为了避免每个请求创建和销毁 `Lua closure` 的开销，建议将函数的定义都放置在自己的 `Lua module` 中，例如：

```
-- my_module.lua
local _M = {_VERSION = "0.1"}

function _M.foo()
    -- your code
    print("i'm foo")
end

return _M
```

然后，再在 `content_by_lua_file` 指向的 `.lua` 文件中调用它：

```
local my_module = require "my_module"
my_module.foo()
```

因为 **Lua module** 只会在第一次请求时加载一次（除非显式禁用了 `lua_code_cache` 配置指令），后续请求便可直接复用。

点号与冒号操作符的区别

看下面示例代码：

```
local str = "abcde"
print("case 1:", str:sub(1, 2))
print("case 2:", str.sub(str, 1, 2))
```

执行结果：

```
case 1: ab
case 2: ab
```

冒号操作会带入一个 `self` 参数，用来代表 `自己`。而点号操作，只是 `内容` 的展开。

在函数定义时，使用冒号将默认接收一个 `self` 参数，而使用点号则需要显式传入 `self` 参数。

示例代码：

```
obj = { x = 20 }

function obj:fun1()
    print(self.x)
end
```

等价于

```
obj = { x = 20 }

function obj.fun1(self)
    print(self.x)
end
```

参见 [官方文档](#) 中的以下片段：

The colon syntax is used for defining methods, that is, functions that have an implicit extra parameter `self`. Thus, the statement

```
function t.a.b.c:f (params) body end
```

is syntactic sugar for

```
t.a.b.c.f = function (self, params) body end
```

冒号的操作，只有当变量是类对象时才需要。有关如何使用 **Lua** 构造类，大家可参考相关章节。

module 是邪恶的

Lua 是所有脚本语言中最快、最简洁的，我们爱她的快、她的简洁，但是我们也不得不忍受因为这些快、简洁最后带来的一些弊端，我们来挨个数数 module 有多少“邪恶”的吧。

由于 `lua_code_cache off` 情况下，缓存的代码会伴随请求完结而释放。module 的最大好处缓存这时候是无法发挥的，所以本章的内容都是基于 `lua_code_cache on` 的情况下。

先看看下面代码：

```

local ngx_socket_tcp = ngx.socket.tcp          -- ①

local _M = { _VERSION = '0.06' }              -- ②
local mt = { __index = _M }                   -- ③

function _M.new(self)
    local sock, err = ngx_socket_tcp()         -- ④
    if not sock then
        return nil, err
    end
    return setmetatable({ sock = sock }, mt)   -- ⑤
end

function _M.set_timeout(self, timeout)
    local sock = self.sock
    if not sock then
        return nil, "not initialized"
    end

    return sock:settimeout(timeout)
end

-- ... 其他功能代码，这里简略

return _M

```

① 对于比较底层的模块，内部使用到的非本地函数，都需要 `local` 本地化，这样做的好处：

- 避免命名冲突：防止外部是 `require(...)` 的方法调用造成全局变量污染
- 访问局部变量的速度比全局变量更快、更快、更快（重要事情说三遍）

② 每个基础模块最好有自己 `_VERSION` 标识，方便后期利用 `_VERSION` 完成热代码部署等高级特性，也便于使用者对版本有整体意识。

③ 其实 `_M` 和 `mt` 对于不同的请求实例（`require` 方法得到的对象）是相同的，因为 `module` 会被缓存到全局环境中。所以在这个位置千万不要放单请求内个性信息，例如 `ngx.ctx` 等变量。

④ 这里需要实现的是给每个实例绑定不同的 `tcp` 对象，后面 `setmetatable` 确保了每个实例拥有自己的 `socket` 对象，所以必须放在 `new` 函数中。如果放在 ③ 的下面，那么这时候所有的不同实例内部将绑定了同一个 `socket` 对象。

```
?local mt = { __index = _M }          -- ③
?local sock = ngx_socket_tcp()        -- ④ 错误的
?
?function _M.new(self)
?    return setmetatable({ sock = sock }, mt) -- ⑤
?end
```

⑤ Lua 的 `module` 有两种类型：支持面向对象痕迹可以保留私有属性；静态方法提供者，没有任何私有属性。真正起到区别作用的就是 `setmetatable` 函数，是否有自己的个性元表，最终导致两种不同的形态。

笔者写这章的时候，想起一个场景，我觉得两者之间重叠度很大。
不幸的婚姻有千万种，可幸福的婚姻只有一种。糟糕的 `module` 有千万个错误，可好的 `module` 都一个样。我们真没必要尝试了解所有错误格式的不好，但是正确的格式就摆在那里，不懂就照搬，搬多了就有感觉了。起点的不同，可以让我们从一开始有正确的认知形态，少走弯路，多一些时间学习有价值的东西。

也许你要问，哪里有正确的 `module` 所有格式？先从 OpenResty 默认绑定的各种 `lua-resty-*` 代码开始熟悉吧，她就是我说的正确格式（注意：这里我用了一个女字旁的 她，看的出来我有多爱她了）。

FFI

`FFI` 库，是 `LuaJIT` 中最重要的一个扩展库。它允许从纯 `Lua` 代码调用外部 C 函数，使用 C 数据结构。有了它，就不用再像 `Lua` 标准 `math` 库一样，编写 `Lua` 扩展库。把开发者从开发 `Lua` 扩展 C 库（语言/功能绑定库）的繁重工作中释放出来。学习完本小节对开发纯 `ffi` 的库是有帮助的，像 [Iru-resty-irucache](#) 中的 `pureffi.lua`，这个纯 `ffi` 库非常高效地完成了 `Iru` 缓存策略。

简单解释一下 `Lua` 扩展 C 库，对于那些能够被 `Lua` 调用的 C 函数来说，它的接口必须遵循 `Lua` 要求的形式，就是 `typedef int (*lua_CFunction)(lua_State* L)`，这个函数包含的参数是 `lua_State` 类型的指针 `L`。可以通过这个指针进一步获取通过 `Lua` 代码传入的参数。这个函数的返回值类型是一个整型，表示返回值的数量。需要注意的是，用 C 编写的函数无法把返回值返回给 `Lua` 代码，而是通过虚拟栈来传递 `Lua` 和 C 之间的调用参数和返回值。不仅在编程上开发效率变低，而且性能上比不上 `FFI` 库调用 C 函数。

`FFI` 库最大限度的省去了使用 C 手工编写繁重的 `Lua/C` 绑定的需要。不需要学习一门独立/额外的绑定语言——它解析普通 C 声明。这样可以从 C 头文件或参考手册中，直接剪切，粘贴。它的任务就是绑定很大的库，但不需要捣鼓脆弱的绑定生成器。

`FFI` 紧紧的整合进了 `LuaJIT`（几乎不可能作为一个独立的模块）。`JIT` 编译器在 C 数据结构上所产生的代码，等同于一个 C 编译器应该生产的代码。在 `JIT` 编译过的代码中，调用 C 函数，可以被内连处理，不同于基于 `Lua/C API` 函数调用。

ffi 库 词汇

<i>noun</i>	<i>Explanation</i>
<code>cdecl</code>	A definition of an abstract C type(actually, is a <code>lua string</code>)
<code>ctype</code>	C type object
<code>cdata</code>	C data object
<code>ct</code>	C type format, is a template object, may be <code>cdecl</code> , <code>cdata</code> , <code>ctype</code>
<code>cb</code>	callback object
<code>VLA</code>	An array of variable length
<code>VLS</code>	A structure of variable length

ffi.* API

功能：Lua ffi 库的 API，与 LuaJIT 不可分割。

毫无疑问，在 `lua` 文件中使用 `ffi` 库的时候，必须要有下面的一行。

```
local ffi = require "ffi"
```

ffi.cdef

语法：`ffi.cdef(def)`

功能：声明 C 函数或者 C 的数据结构，数据结构可以是结构体、枚举或者是联合体，函数可以是 C 标准函数，或者第三方库函数，也可以是自定义的函数，注意这里只是函数的声明，并不是函数的定义。声明的函数应该要和原来的函数保持一致。

```
ffi.cdef[[
typedef struct foo { int a, b; } foo_t; /* Declare a struct and typedef. */
int printf(const char *fmt, ...); /* Declare a typical printf function. */
]]
```

注意：所有使用的库函数都要对其进行声明，这和我们写 C 语言时候引入 `.h` 头文件是一样的。

顺带一提的是，并不是所有的 C 标准函数都能满足我们的需求，那么如何使用第三方库函数或自定义的函数呢，这会稍微麻烦一点，不用担心，你可以很快学会。:) 首先创建一个 `myffi.c`，其内容是：

```
int add(int x, int y)
{
    return x + y;
}
```

接下来在 Linux 下生成动态链接库：

```
gcc -g -o libmyffi.so -fpic -shared myffi.c
```

为了方便我们测试，我们在 `LD_LIBRARY_PATH` 这个环境变量中加入了刚刚库所在的路径，因为编译器在查找动态库所在的路径的时候其中一个环节就是在 `LD_LIBRARY_PATH` 这个环境变量中的所有路径进行查找。命令如下所示。

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:your_lib_path
```

在 Lua 代码中要增加如下的行：

```
ffi.load(name [,global])
```

`ffi.load` 会通过给定的 `name` 加载动态库，返回一个绑定到这个库符号的新的 C 库命名空间，在 `POSIX` 系统中，如果 `global` 被设置为 `true`，这个库符号被加载到一个全局命名空间。另外这个 `name` 可以是一个动态库的路径，那么会根据路径来查找，否则的话会在默认的搜索路径中去找动态库。在 `POSIX` 系统中，如果在 `name` 这个字段中没有写上点符号 `.`，那么 `.so` 将会被自动添加进去，例如 `ffi.load("z")` 会在默认的共享库搜寻路径中去查找 `libz.so`，在 `windows` 系统，如果没有包含点号，那么 `.dll` 会被自动加上。

下面看一个完整例子：

```
local ffi = require "ffi"
local myffi = ffi.load('myffi')

ffi.cdef[[
int add(int x, int y); /* don't forget to declare */
]]

local res = myffi.add(1, 2)
print(res) -- output: 3 Note: please use luajit to run this script.
```

除此之外，还能使用 `ffi.c` (调用 `ffi.cdef` 中声明的系统函数) 来直接调用 `add` 函数，记得要在 `ffi.load` 的时候加上参数 `true`，例如 `ffi.load('myffi', true)`。

完整的代码如下所示：

```
local ffi = require "ffi"
ffi.load('myffi',true)

ffi.cdef[[
int add(int x, int y); /* don't forget to declare */
]]

local res = ffi.C.add(1, 2)
print(res) -- output: 3 Note: please use luajit to run this script.
```

ffi.typeof

语法：`ctype = ffi.typeof(ct)`

功能：创建一个 `ctype` 对象，会解析一个抽象的 C 类型定义。

```

local uintptr_t = ffi.typeof("uintptr_t")
local c_str_t = ffi.typeof("const char*")
local int_t = ffi.typeof("int")
local int_array_t = ffi.typeof("int[?]")

```

ffi.new

语法： `cdata = ffi.new(ct [,nelem] [,init...])`

功能： 开辟空间，第一个参数为 `ctype` 对象，`ctype` 对象最好通过 `ctype = ffi.typeof(ct)` 构建。

顺便一提，可能很多人会有疑问，到底 `ffi.new` 和 `ffi.C.malloc` 有什么区别呢？

如果使用 `ffi.new` 分配的 `cdata` 对象指向的内存块是由垃圾回收器 `Luajit GC` 自动管理的，所以不需要用户去释放内存。

如果使用 `ffi.C.malloc` 分配的空间便不再使用 `Luajit` 自己的分配器了，所以不是由 `Luajit GC` 来管理的，但是，要注意的是 `ffi.C.malloc` 返回的指针本身所对应的 `cdata` 对象还是由 `Luajit GC` 来管理的，也就是这个指针的 `cdata` 对象指向的是用 `ffi.C.malloc` 分配的内存空间。这个时候，你应该通过 `ffi.gc()` 函数在这个 C 指针的 `cdata` 对象上面注册自己的析构函数，这个析构函数里面你可以再调用 `ffi.C.free`，这样的话当 C 指针所对应的 `cdata` 对象被 `Luajit GC` 管理器垃圾回收时候，也会自动调用你注册的那个析构函数来执行 C 级别的内存释放。

请尽可能使用最新版本的 `Luajit`，`x86_64` 上由 `Luajit GC` 管理的内存已经由 `1G->2G`，虽然管理的内存变大了，但是如果要使用很大的内存，还是用 `ffi.C.malloc` 来分配会比较好，避免耗尽了 `Luajit GC` 管理内存的上限，不过还是建议不要一下子分配很大的内存。

```

local int_array_t = ffi.typeof("int[?]")
local bucket_v = ffi.new(int_array_t, bucket_sz)

local queue_arr_type = ffi.typeof("lruqueue_pureffi_queue_t[?]")
local q = ffi.new(queue_arr_type, size + 1)

```

ffi.fill

语法： `ffi.fill(dst, len [,c])`

功能： 填充数据，此函数和 `memset(dst, c, len)` 类似，注意参数的顺序。

```
ffi.fill(self.bucket_v, ffi_sizeof(int_t, bucket_sz), 0)
ffi.fill(q, ffi_sizeof(queue_type, size + 1), 0)
```

ffi.cast

语法： `cdata = ffi.cast(ct, init)`

功能： 创建一个 `scalar cdata` 对象。

```
local c_str_t = ffi.typeof("const char*")
local c_str = ffi.cast(c_str_t, str)           -- 转换为指针地址

local uintptr_t = ffi.typeof("uintptr_t")
tonumber(ffi.cast(uintptr_t, c_str))          -- 转换为数字
```

cdata 对象的垃圾回收

所有由显式的 `ffi.new()`, `ffi.cast()` etc. 或者隐式的 `accessors` 所创建的 `cdata` 对象都是能被垃圾回收的，当他们被使用的时候，你需要确保有在 `Lua stack`, `upvalue`，或者 `Lua table` 上保留有对 `cdata` 对象的有效引用，一旦最后一个 `cdata` 对象的有效引用失效了，那么垃圾回收器将自动释放内存（在下一个 `GC` 周期结束时候）。另外如果你要分配一个 `cdata` 数组给一个指针的话，你必须保持这个持有这个数据的 `cdata` 对象活跃，下面给出一个官方的示例：

```
ffi.cdef[[
typedef struct { int *a; } foo_t;
]]

local s = ffi.new("foo_t", ffi.new("int[10]")) -- WRONG!

local a = ffi.new("int[10]") -- OK
local s = ffi.new("foo_t", a)
-- Now do something with 's', but keep 'a' alive until you're done.
```

相信看完上面的 `API` 你已经很累了，再坚持一下吧！休息几分钟后，让我们来看看下面对官方文档中的示例做剖析，希望能再加深你对 `ffi` 的理解。

调用 C 函数

真的很用容易去调用一个外部 C 库函数，示例代码：

```
local ffi = require("ffi")
ffi.cdef[[
int printf(const char *fmt, ...);
]]
ffi.C.printf("Hello %s!", "world")
```

以上操作步骤，如下：

1. 加载 FFI 库。
2. 为函数增加一个函数声明。这个包含在 中括号 对之间的部分，是标准 C 语法。
3. 调用命名的 C 函数——非常简单。

事实上，背后的实现远非如此简单：③ 使用标准 C 库的命名空间 `ffi.c`。通过符号名 `printf` 索引这个命名空间，自动绑定标准 C 库。索引结果是一个特殊类型的对象，当被调用时，执行 `printf` 函数。传递给这个函数的参数，从 `Lua` 对象自动转换为相应的 C 类型。

再来一个源自官方的示例代码：

```

local ffi = require("ffi")
ffi.cdef[[
unsigned long compressBound(unsigned long sourceLen);
int compress2(uint8_t *dest, unsigned long *destLen,
             const uint8_t *source, unsigned long sourceLen, int level);
int uncompress(uint8_t *dest, unsigned long *destLen,
               const uint8_t *source, unsigned long sourceLen);
]]
local zlib = ffi.load(ffi.os == "Windows" and "zlib1" or "z")

local function compress(txt)
    local n = zlib.compressBound(#txt)
    local buf = ffi.new("uint8_t[?]", n)
    local buflen = ffi.new("unsigned long[1]", n)
    local res = zlib.compress2(buf, buflen, txt, #txt, 9)
    assert(res == 0)
    return ffi.string(buf, buflen[0])
end

local function uncompress(comp, n)
    local buf = ffi.new("uint8_t[?]", n)
    local buflen = ffi.new("unsigned long[1]", n)
    local res = zlib.uncompress(buf, buflen, comp, #comp)
    assert(res == 0)
    return ffi.string(buf, buflen[0])
end

-- Simple test code.
local txt = string.rep("abcd", 1000)
print("Uncompressed size: ", #txt)
local c = compress(txt)
print("Compressed size: ", #c)
local txt2 = uncompress(c, #txt)
assert(txt2 == txt)

```

解释一下这段代码。我们首先使用 `ffi.cdef` 声明了一些被 `zlib` 库提供的 C 函数。然后加载 `zlib` 共享库，在 Windows 系统上，则需要我们手动从网上下载 `zlib1.dll` 文件，而在 POSIX 系统上 `libz` 库一般都会被预安装。因为 `ffi.load` 函数会自动填补前缀和后缀，所以我们简单地使用 `z` 这个字母就可以加载了。我们检查 `ffi.os`，以确保我们传递给 `ffi.load` 函数正确的名字。

一开始，压缩缓冲区的最大值被传递给 `compressBound` 函数，下一行代码分配了一个要压缩字符串长度的字节缓冲区。`[?]` 意味着他是一个变长数组。它的实际长度由 `ffi.new` 函数的第二个参数指定。

我们仔细审视一下 `compress2` 函数的声明就会发现，目标长度是用指针传递的！这是因为我们要传递进去缓冲区的最大值，并且得到缓冲区实际被使用的大小。

在 C 语言中，我们可以传递变量地址。但因为在 Lua 中并没有地址相关的操作符，所以我们使用只有一个元素的数组来代替。我们先用最大缓冲区大小初始化这唯一一个元素，接下来就是很直观地调用 `zlib.compress2` 函数了。使用 `ffi.string` 函数得到一个存储着压缩数据的 Lua 字符串，这个函数需要一个指向数据起始区的指针和实际长度。实际长度将会在 `buflen` 这个数组中返回。因为压缩数据并不包括原始字符串的长度，所以我们要显式地传递进去。

使用 C 数据结构

`cdata` 类型用来将任意 C 数据保存在 Lua 变量中。这个类型相当于一块原生的内存，除了赋值和相同性判断，Lua 没有为之预定义任何操作。然而，通过使用 `metatable`（元表），程序员可以为 `cdata` 自定义一组操作。`cdata` 不能在 Lua 中创建出来，也不能在 Lua 中修改。这样的操作只能通过 C API。这一点保证了宿主程序完全掌管其中的数据。

我们将 C 语言类型与 `metamethod`（元方法）关联起来，这个操作只用做一次。`ffi.metatype` 会返回一个该类型的构造函数。原始 C 类型也可以被用来创建数组，元方法会被自动地应用到每个元素。

尤其需要指出的是，`metatable` 与 C 类型的关联是永久的，而且不允许被修改，`__index` 元方法也是。

下面是一个使用 C 数据结构的实例

```
local ffi = require("ffi")
ffi.cdef[[
typedef struct { double x, y; } point_t;
]]

local point
local mt = {
    __add = function(a, b) return point(a.x+b.x, a.y+b.y) end,
    __len = function(a) return math.sqrt(a.x*a.x + a.y*a.y) end,
    __index = {
        area = function(a) return a.x*a.x + a.y*a.y end,
    },
}
point = ffi.metatype("point_t", mt)

local a = point(3, 4)
print(a.x, a.y) --> 3 4
print(#a) --> 5
print(a:area()) --> 25
local b = a + point(0.5, 8)
print(#b) --> 12.5
```

附表：Lua 与 C 语言语法对应关系

<i>Idiom</i>	<i>C code</i>	<i>Lua code</i>
Pointer dereference	<code>x = *p</code>	<code>x = p[0]</code>
<code>int *p</code>	<code>*p = y</code>	<code>p[0] = y</code>
Pointer indexing	<code>x = p[i]</code>	<code>x = p[i]</code>
<code>int i, *p</code>	<code>p[i+1] = y</code>	<code>p[i+1] = y</code>
Array indexing	<code>x = a[i]</code>	<code>x = a[i]</code>
<code>int i, a[]</code>	<code>a[i+1] = y</code>	<code>a[i+1] = y</code>
struct/union dereference	<code>x = s.field</code>	<code>x = s.field</code>
<code>struct foo s</code>	<code>s.field = y</code>	<code>s.field = y</code>
struct/union pointer deref	<code>x = sp->field</code>	<code>x = sp.field</code>
<code>struct foo *sp</code>	<code>sp->field = y</code>	<code>s.field = y</code>
<code>int i, *p</code>	<code>y = p - i</code>	<code>y = p - i</code>
Pointer dereference	<code>x = p1 - p2</code>	<code>x = p1 - p2</code>
Array element pointer	<code>x = &a[i]</code>	<code>x = a + i</code>

内存问题

todo：介绍 FFI 就必须从必要的 C 基础，包括内存管理的细节，说起，同时也须介绍包括 Valgrind 在内的内存问题调试工具的细节（by agentzh），后面重点补充。

什么是 JIT？

自从 OpenResty 1.5.8.1 版本之后，默认捆绑的 **Lua** 解释器就被替换成 **LuaJIT**，而不再是标准 **Lua**。单从名字上，我们就可以直接看到这个新的解释器多了一个 **JIT**，接下来我们就一起来聊聊 **JIT**。

先看一下 **LuaJIT** 官方的解释：**LuaJIT is a Just-In-Time Compiler for the Lua programming language**。

LuaJIT 的运行时环境包括一个用手写汇编实现的 **Lua** 解释器和一个可以直接生成机器代码的 **JIT** 编译器。

Lua 代码在被执行之前总是会先被 **lfn** 成 **LuaJIT** 自己定义的字节码（**Byte Code**）。关于 **LuaJIT** 字节码的文档，可以参见：<http://wiki.luajit.org/Bytecode-2.0>（这个文档描述的是 **LuaJIT 2.0** 的字节码，不过 2.1 里面的变化并不算太大）。

一开始的时候，**Lua** 字节码总是被 **LuaJIT** 的解释器解释执行。**LuaJIT** 的解释器会在执行字节码时同时记录一些运行时的统计信息，比如每个 **Lua** 函数调用入口的实际运行次数，还有每个 **Lua** 循环的实际执行次数。当这些次数超过某个预设的阈值时，便认为对应的 **Lua** 函数入口或者对应的 **Lua** 循环足够的“热”，这时便会触发 **JIT** 编译器开始工作。

JIT 编译器会从热函数的入口或者热循环的某个位置开始尝试编译对应的 **Lua** 代码路径。编译的过程是把 **LuaJIT** 字节码先转换成 **LuaJIT** 自己定义的中间码（**IR**），然后再生成针对目标体系结构的机器码（比如 **x86_64** 指令组成的机器码）。

如果当前 **Lua** 代码路径上的所有的操作都可以被 **JIT** 编译器顺利编译，则这条编译过的代码路径便被称为一个“**trace**”，在物理上对应一个 **trace** 类型的 **GC** 对象（即参与 **Lua GC** 的对象）。

你可以通过 **ngx-lj-gc-objs** 工具看到指定的 **Nginx worker** 进程里所有 **trace** 对象的一些基本的统计信息，见 <https://github.com/openresty/stapxx#ngx-lj-gc-objs>

比如下面这一行 **ngx-lj-gc-objs** 工具的输出

```
102 trace objects: max=928, avg=337, min=160, sum=34468 (in bytes)
```

则表明当前进程内的 **LuaJIT VM** 里一共有 102 个 **trace** 类型的 **GC** 对象，其中最小的 **trace** 占用 160 个字节，最大的占用 928 个字节，平均大小是 337 字节，而所有 **trace** 的总大小是 34468 个字节。

LuaJIT 的 JIT 编译器的实现目前还不完整，有一些基本原语它还无法编译，比如 `pairs()` 函数、`unpack()` 函数、`string.match()` 函数、基于 `lua_CFunction` 实现的 Lua C 模块、`FNEW` 字节码，等等。所以当 JIT 编译器在当前代码路径上遇到了它不支持的操作，便会立即终止当前的 `trace` 编译过程（这被称为 `trace abort`），而重新退回到解释器模式。

JIT 编译器不支持的原语被称为 NYI (Not Yet Implemented) 原语。比较完整的 NYI 列表在这篇文档里面：

```
http://wiki.luajit.org/NYI
```

所谓“让更多的 Lua 代码被 JIT 编译”，其实就是帮助更多的 Lua 代码路径能为 JIT 编译器所接受。这一般通过两种途径来实现：

1. 调整对应的 Lua 代码，避免使用 NYI 原语。
2. 增强 JIT 编译器，让越来越多的 NYI 原语能够被编译。

对于第 2 种方式，春哥一直在推动公司 (CloudFlare) 赞助 Mike Pall 的开发工作。不过有些原语因为本身的代价过高，而永远不会被编译，比如基于经典的 `lua_CFunction` 方式实现的 Lua C 模块（所以需要尽量通过 LuaJIT 的 FFI 来调用 C）。

而对于第 1 种方法，我们如何才能知道具体是哪一行 Lua 代码上的哪一个 NYI 原语终止了 `trace` 编译呢？答案很简单。就是使用 LuaJIT 安装自带的 `jit.v` 和 `jit.dump` 这两个 Lua 模块。这两个 Lua 模块会打印出 JIT 编译器工作的细节过程。

在 Nginx 的上下文中，我们可以在 `nginx.conf` 文件中的 `http {}` 配置块中添加下面这一段：

```
init_by_lua_block {
    local verbose = false
    if verbose then
        local dump = require "jit.dump"
        dump.on(nil, "/tmp/jit.log")
    else
        local v = require "jit.v"
        v.on("/tmp/jit.log")
    end

    require "resty.core"
}
```

那一行 `require "resty.core"` 倒并不是必需的，放在那里的主要目的是为了尽量避免使用 `ngx_lua` 模块自己的基于 `lua_CFunction` 的 Lua API，减少 NYI 原语。

在上面这段 Lua 代码中，当 `verbose` 变量为 `false` 时（默认就为 `false` 哈），我们使用 `jit.v` 模块打印出比较简略的流水信息到 `/tmp/jit.log` 文件中；而当 `verbose` 变量为 `true` 时，我们则使用 `jit.dump` 模块打印所有的细节信息，包括每个 `trace` 内部的字节码、IR 码和最终生成的机

器指令。

这里我们主要以 `jit.v` 模块为例。在启动 Nginx 之后，应当使用 `ab` 和 `weighttp` 这样的工具对相应的服务接口进行预热，以触发 `LuaJIT` 的 JIT 编译器开始工作（还记得刚才我们说的“热函数”和“热循环”吗？）。预热过程一般不用太久，跑个二三百个请求足矣。当然，压更多的请求也没关系。完事后，我们就可以检查 `/tmp/jit.log` 文件里面的输出了。

`jit.v` 模块的输出里如果有类似下面这种带编号的 `TRACE` 行，则指示成功编译了的 `trace` 对象，例如

```
[TRACE 6 shdict.lua:126 return]
```

这个 `trace` 对象编号为 6，对应的 Lua 代码路径是从 `shdict.lua` 文件的第 126 行开始的。

下面这样的也是成功编译了的 `trace`:

```
[TRACE 16 (15/1) waf-core.lua:419 -> 15]
```

这个 `trace` 编号为 16，是从 `waf-core.lua` 文件的第 419 行开始的，同时它和编号为 15 的 `trace` 联接了起来。

而下面这个例子则是被中断的 `trace`:

```
[TRACE --- waf-core.lua:455 -- NYI: FastFunc pairs at waf-core.lua:458]
```

上面这一行是说，这个 `trace` 是从 `waf-core.lua` 文件的第 455 行开始编译的，但当编译到 `waf-core.lua` 文件的第 458 行时，遇到了一个 `NYI` 原语编译不了，即 `pairs()` 这个内建函数，于是当前的 `trace` 编译过程被迫终止了。

类似的例子还有下面这些：

```
[TRACE --- exit.lua:27 -- NYI: FastFunc coroutine.yield at waf-core.lua:439]
[TRACE --- waf.lua:321 -- NYI: bytecode 51 at raven.lua:107]
```

上面第二行是因为操作码 51 的 `LuaJIT` 字节码也是 `NYI` 原语，编译不了。

那么我们如何知道 51 字节码究竟是啥呢？我们可以用 `nginx-devel-utils` 项目中的 `lcbc.lua` 脚本来取得 51 号字节码的名字：

```
$ /usr/local/openresty/luajit/bin/luajit-2.1.0-alpha lcbc.lua 51
opcode 51:
FNEW
```

我们看到原来是用来（动态）创建 `Lua` 函数的 `FNEW` 字节码。`lcbc.lua` 脚本的位置是

<https://github.com/agentzh/nginx-devel-utils/blob/master/ljbc.lua>

非常简单的一个脚本，就几行 Lua 代码。

这里需要提醒的是，不同版本的 LuaJIT 的字节码可能是不相同的，所以一定要使用和你的 Nginx 链接的同一个 LuaJIT 来运行这个 `ljbc.lua` 工具，否则有可能会得到错误的结果。

我们实际做个对比实验，看看 JIT 带来的好处：

本例子可以看到效率相差大约 $9.2/5.19 \approx 1.77$ 倍，换句话说标准 Lua 需要 177% 的时间才能完成同样的工作。估计大家觉得这个还不过瘾，再看下面示例代码：

文件 test.lua :

```

local loop_count = tonumber(arg[1])
local fun_pair = "ipairs" == arg[2] and ipairs or pairs

local t = {}
for i=1,100 do
    t[i] = i
end

for i=1,loop_count do
    for j=1,1000 do
        for k,v in fun_pair(t) do
            --
        end
    end
end

```

执行参数	执行结果
time lua test.lua 1000 ipairs	3.96s user 0.02s system 98% cpu 4.039 total
time lua test.lua 1000 pairs	3.97s user 0.01s system 99% cpu 3.992 total
time luajit test.lua 1000 ipairs	0.10s user 0.00s system 95% cpu 0.113 total
time luajit test.lua 10000 ipairs	0.98s user 0.00s system 99% cpu 0.991 total
time luajit test.lua 1000 pairs	1.54s user 0.01s system 99% cpu 1.559 total

从这个执行结果中，大致可以总结出下面几个观点：

- 在标准 Lua 解释器中，使用 ipairs 或 pairs 没有区别；
- 对于 pairs 方式，LuaJIT 的性能大约是标准 Lua 的 4 倍；
- 对于 ipairs 方式，LuaJIT 的性能大约是标准 Lua 的 40 倍。

可以被 JIT 编译的元操作

下面给大家列一下截止到目前已经可以被 JIT 编译的元操作。其他还有 IO、Bit、FFI、Coroutine、OS、Package、Debug、JIT 等分类，使用频率相对较低，这里就不罗列了，可以参考官网：<http://wiki.luajit.org/NYI>。

基础库的支持情况

函数	编译?	备注
assert	yes	
collectgarbage	no	
dofile	never	
error	never	
getfenv	2.1 partial	只有 getfenv(0) 能编译
getmetatable	yes	
ipairs	yes	
load	never	
loadfile	never	
loadstring	never	
next	no	
pairs	no	
pcall	yes	
print	no	
rawequal	yes	
rawget	yes	
rawlen (5.2)	yes	
rawset	yes	
select	partial	第一个参数是静态变量的时候可以编译
setfenv	no	
setmetatable	yes	
tonumber	partial	不能编译非10进制，非预期的异常输入
tostring	partial	只能编译：字符串、数字、布尔、nil 以及支持 __tostring 元方法的类型
type	yes	
unpack	no	
xpcall	yes	

字符串库

函数	编译？	备注
string.byte	yes	
string.char	2.1	
string.dump	never	
string.find	2.1 partial	只有字符串样式查找（没有样式）
string.format	2.1 partial	不支持 %p 或 非字符串参数的 %s
string.gmatch	no	
string.gsub	no	
string.len	yes	
string.lower	2.1	
string.match	no	
string.rep	2.1	
string.reverse	2.1	
string.sub	yes	
string.upper	2.1	

表

函数	编译？	备注
table.concat	2.1	
table.foreach	no	2.1: 内部编译，但还没有外放
table.foreachi	2.1	
table.getn	yes	
table.insert	partial	只有 push 操作
table.maxn	no	
table.pack (5.2)	no	
table.remove	2.1	部分，只有 pop 操作
table.sort	no	
table.unpack (5.2)	no	

math 库

函数	编译?	备注
math.abs	yes	
math.acos	yes	
math.asin	yes	
math.atan	yes	
math.atan2	yes	
math.ceil	yes	
math.cos	yes	
math.cosh	yes	
math.deg	yes	
math.exp	yes	
math.floor	yes	
math.fmod	no	
math.frexp	no	
math.ldexp	yes	
math.log	yes	
math.log10	yes	
math.max	yes	
math.min	yes	
math.modf	yes	
math.pow	yes	
math.rad	yes	
math.random	yes	
math.randomseed	no	
math.sin	yes	
math.sinh	yes	
math.sqrt	yes	
math.tan	yes	
math.tanh	yes	

