Lua教程

极客学院出版

前言

Lua 语言是基于 C 语言之上的开源编程语言。它的价值在于其跨平台的特性，从大型服务器系统到小型移动应用都可以看到它的身影。

本教程不仅包括 Lua 编程语言的基本知识，还包括 Lua 语言在各类应用场景中的应用。

适用人群

本教程主要是为 Lua 语言的初学者准备的。不过，其中也包含了既适合初学者也适合高级用户的内容。

学习前提

建议您在开始学习这篇教程之前先掌握一些计算机编程的基本概念。但是教程本身是包含编程的基本概念的，即使您是完全的初始者，您也能够学习到大量的 Lua 语言的编程概念。您只需有一些基本的文本编辑与命令行的知识即可。

目录

1 Lua 基础

概述

Lua 是用 C 语言开发的可扩展的轻量级编程语言。它起源于 1993 年由 Roberto lerusalimschy,Luiz Henriqu e de Figueiredo 与 Waddemar Celes 领导的一个内部项目。
设计者的初衷是希望 Lua 可以成为一款整合 C 语言代码以及其它传统语言代码的软件。这种整合会带来很多好处，它让你不需要重复做 C 语言已经做的很好的工作，而专注于提供那些 C 语言不擅长的特性：提供更高的抽象（离硬件更远）、动态结构、无冗余、易于测试与调试。为了提供这些特性，Lua 提供了安全的环境、动态内存管理，以及擅长处理字符串和其它动态大小数据结构的工具。

特点

Lua 有着许多自身的特点使得它与其它编程语言不同。主要包括：

- 可扩展性
- 简单
- 高效
- 跨平台
- 免费与开源

示例代码

```
print("Hello World!")
```

Lua 是如何实现的

Lua 主要包括两个部分：Lua 解释器部分和运行软件系统。 该运行软件系统是一个实际的计算机应用程序，它可以解释用 Lua 编写的程序（译注：此处 Lua 翻译器部分用于将 Lua　代码编译成中间字节码，运行软件系统指 Lua 虚拟机，而一般我们所说 Lua 解释器包括这两部分）。 Lua 解释器是由 ANSI C 编写的，因此它有很好的可移植性，可以运行各种各运的设备上，无论是大型网络服务器还是小型移动设备。

无论 Lua 语言还是 Lua 解释器都已经是非常成熟的、同时还兼备体积小，运行速度非常快的特点。小体积的特性也使得 Lua 可以运行在很多只有少量内存的小型设备中。

学习 Lua

学习 Lua 语言最重要的一点是把注意力放在它的概念上，千万不要迷失在语言的技术细节中。

学习 Lua 的目的是成为一个更好的程序人员。也就是说，学习 Lua 可以帮助您在设计与实现新系统，或者维护旧系统的时候变得更加的高效。

Lua 的应用场景

- 游戏开发
- 开发单机应用
- 网站开发
- 扩展数据库或者为数据库开发插件，比如，MySQL 代理或 MySQL WorkBench
- 开发安全系统，如入侵检测系统（IDS）

运行环境

本地环境搭建

在本地搭建 Lua 编程语言的开发运行环境，你需要在你的计算机上安装如下三个软件：(1) 文本编辑器。(2) Lua 解释器。（3）Lua 编译器。

文本编辑器

文本编辑器用来编辑你的程序代码。有如下几款常用的文本编辑器软件：Windows notepad、Brief、Epsilon、EMACS、vim/vi。

在不同的操作系统中有各自不同的编辑器，而且编辑器的版本不一样。例如，Notepad 主要用在 Windows 系统中，vim/vi 不仅可以用于 Windows 系统也可以用于 Linux 和 UNIX 操作系统。

用文本编辑器编辑的文件被称为源文件。源文件中包含程序的源代码。Lua 程序的源文件经常以 .lua 作为其后缀名。

开始编写程序之前，请确保您已经安装好一个文本编辑软件，并且曾经有过写代码，将其存入文件，生成并执行的经验。

Lua 解释器

Lua 解释器是一个能让您输入 Lua 命令并立即执行的小程序。它在执行一个 Lua 文件过程中，一旦遇到错误就立即停止执行，而不像编译器会执行完整个文件。

Lua 编译器

如果将 Lua 扩展到其它语言或者应用中时，我们需要一个软件开发工具箱以及 Lua 应用程序接口兼容的编译器。

在 Windows 系统安装 Lua

在 Windows 系统环境可以安装一个叫 SciTE 的 Lua 开发 IDE (集成开发环境)。它可以在这儿下载：http://code.google.com/p/luaforwindows/。

运行下载的可执行程序就可安装 Lua 语言的 IDE 了。

在这个 IDE 上，你可以创建并生成 Lua 代码。

如果你希望在命令行模式下安装 Lua，你则需要安装 MinGW 或者 Cygwin，然后在 Windows 系统中编译安装 Lua。

在 Linux 系统安装 Lua

使用下面的命令下载并生成 Lua 程序：

```
$ wget http://www.lua.org/ftp/lua-5.2.3.tar.gz
$ tar zxf lua-5.2.3.tar.gz
$ cd lua-5.2.3
$ make linux test
```

在其它系统上安装 Lua 时，比如 aix，ansi，bsd，generic，linux，mingw，posix，solaris，你需要将 make linux test 命令中的 linux 替换为相应的系统平台名称。

假设我们已经有一个文件 helloWord.lua ，文件内容如下：

```
print("Hello World!")
```

我们先使用 cd 命令切换至 helloWord.lua　文件所在的目录，然后生成并运行该文件：

```
$ lua helloWorld
```

执行上面的命令，我们可以看到如下的输出：

```
hello world
```

在 Mac OS X 系统安装 Lua

使用下面的命令可以在 Mac OS X 系统生成并测试 Lua：

```
$ curl -R -O http://www.lua.org/ftp/lua-5.2.3.tar.gz
$ tar zxf lua-5.2.3.tar.gz
$ cd lua-5.2.3
$ make macosx test
```

如果你没有安装 Xcode 和命令行工具，那么你就不能使用 make 命令。你先需要从 mac 应用商店安装 Xcode，然后在 Xcode 首选项的下载选项中安装命令行工具组件。完成上面的步骤后，你就可以使用 make 命令了。

make macosx test 命令并不是非执行不可的。即使你没有执行这个命令，你仍可以在你的 Mac OS X 系统中使用 Lua。

假设我们已经有一个文件 helloWord.lua ，文件内容如下：

```
print("Hello World!")
```

我们先使用 cd 命令切换至 helloWord.lua　文件所在的目录，然后生成并运行该文件：

```
$ lua helloWorld
```

执行上面的命令，我们可以看到如下的输出：

```
hello world
```

Lua IDE

正如前面提到的那样，Windows 系统中 SciTE 是 Lua 创始团队提供的默认的 Lua 集成开发环境（IDE)。此外，还有一款名叫 ZeroBrane 的 IDE。它具有跨平台的特性，支持 Windows、Mac 与 Linux。
同时，许多 eclipse 插件使得 eclipse 能成为 Lua 的 IDE。IDE 中像代码自动补全等诸多特性使得开发变得简单了很多，因此建议你使用 IDE 开发 Lua 程序。同样，IDE 也能像 Lua 命令行版本那样提供交互式编程功能。

基本语法

Lua 学起来非常简单。现在，让我们开始创建我们的第一个 Lua 程序吧！

第一个 Lua 程序

Lua 提供交互式编程模式。在这个模式下，你可以一条一条地输入命令，然后立即就可以得到结果。你可以在 shell 中使用 lua -i 或者 lua 命令启动。输入命令后，按下回车键，就启动了交互模式，显示如下：

```
$ lua -i
$ Lua 5.1.4  Copyright (C) 1994-2008 Lua.org, PUC-Rio
quit to end; cd, dir and edit also available
```

你可以使用如下命令打印输出：

```
$> print("test")
```

按下回车键后，你会得到如下输出结果：

```
'test'
```

默认模式编辑

使用 Lua 文件做为解释器的参数启动解释器,然后开始执行文件直到文件结束。当脚本执行结束后，解释器就不在活跃了。

让我们写一个简单的 Lua 程序。所有的 Lua　文件都扩展名都是 `.lua` 。因此，将下面的源代码放到 test.lua 文件中。

```
print("test")
```

假如你已经设置好 Lua 程序的环境，用下面的命令运行程序：

```
$ lua test.lua
```

我们会得到如下的输出结果：

```
test
```

让我们尝试使用另外的方式运行 Lua 程序。下面是修改后的 test.lua 文件：

```
\#!/usr/local/bin/lua
print("test")
```

这里，我们假设你的 Lua 解释器程序在 /usr/local/bin/lua 目录下。test.lua 文件中第一行由于以 # 开始而被解释器忽略，运行这个程序可以得到如下的结果：

```
$ chmod a+rx test.lua
$./test.lua
```

我们会得到如下的的输出结果：

```
test
```

接下来让我们看一下 Lua 程序的基本结构。这样，你可以更容易理解 Lua 编程语言的基本结构单元。

Lua 中的符号

Lua 程序是由大量的符号组成的。这些符号可以分为关键字、标识符、常量、字符串常量几类。例如，下面的 Lua 语句中包含三个符号：

```
io.write("Hello world, from ",_VERSION,"!\n")
```

这三个符号分别是:

```
io.write
(
"Hello world, from ",_VERSION,"!\n"
)
```

注释

注释就是 Lua 程序中的帮助文档，Lua 解释器会自动忽略它们。所有注释都以 --[[开始，并以 --]]结束。如下所示：

```
--[[ my first program in Lua --]]
```

标识符

Lua 中标识符是识别变量、函数或者其它用户自定义项的名字。标符识总是以字母或者下划线开始，其后可以是零个或多个字母、下划线或数字。

Lua 标识符中不允许出现任何标点符号，比如，@，$ 或者 %。Lua 是大小写敏感的语言，因此 Manpower 和 manpower 是 Lua 中两个不同的标识符。下面所列的是一些合法标识符的例子。

```
mohd       zara    abc    move_name    a_123
myname50    _temp    j      a23b9       retVal
```

关键字

下面列表中所示的是 Lua 中一小部分保留字。这些保留字不能用作常量、变量以及任何标识符的名字。

and	break	do	else
elseif	end	false	for
function	if	in	local
nil	not	or	repeat
return	then	true	until
while			

Lua 中的空白符

如果 Lua 程序中某一行只包含空格或者注释，那么这样的一行被称之为空行。 Lua 解释器将完全忽略这一行。

在 Lua 中，空白是用来描述空格、制表符、换行符和注释的术语。空白符用于将语句中的一部分与其它部分区分开，使得解释器可以语句中的一个元素，比如 int，何处结束，以及另一个元素从何处开始。因此，在下面的语句中：

```
local age
```

在 local 与 age 之间至少有一个空白符（通常是空格）,这个空白符使得解释器可以将 local 与 age 区分开。另一方面，在下面的语句中：

```
fruit = apples + oranges   --get the total fruit
```

fruit 与 = 之间以及 = 与 apples 之间的空白符都是可以没有的。但是为了程序的可读性目的，建议你在它们之间使用空白符。

变量

变量就是给一块内存区域赋予的一个名字。变量使得在程序中就可以修改或读取相应的内存区域中的内容。它可以代表各种不同类型的值，包括函数与表。

变量的名字由字母、数字与下划线组成。它必须是字母或下划线开头。由于 Lua 是字母大小写敏感的，所以大写字母与小写字母是不一样的。Lua 中有八种基本值类型：

在 Lua 语言中，虽然我们没有变量数据类型，但是依据变量的作用域我们可以将变量分为三类：

- 全局变量：除非显示的声明一个局部变量，否则所有的变量都被默认当作全局变量。
- 局部变量：如果我们将一个变量定义为局部变量，那么这么变量的作用域就被限制在函数内。
- 表字段：这种特殊的变量可以是除了 nil 以外的所有类型，包括函数。

Lua 变量定义

一个变量定义就意味着告诉解释器在什么地方创建多大的一块存储空间。一个变量定义包括一个可选的类型(type)以及该类型的一个或多个变量名的列表，如下所示：

```
type variable_list;
```

其中，type 是可以选择指定为 local 或者不指定使用默认值 global，variable_list 是包含一个或多个由逗号分隔的标识符名字。下面是合法变量定义的示例：

```
local   i, j
local   i
local   a,c
```

local i,j 声明定义了两个变量 i 与 j；它命令解释器创建两个名称分别为 i,j 的变量，并且将其作用域限制在局部。

在声明变量的时候可以同时初始化变量（为变量赋初值）。在变量名后跟上等号和一个常量表达式就可以初始化变量。如下所示：

```
type variable_list = value_list;
```

一些例子如下：

```
local d , f = 5 ,10 --声明局部变量 d，f。
d , f = 5, 10;     --声明全局变量 d，f。
d, f = 10          --[[声明全局变量 d，f，其中 f 的值是 nil--]]
```

如果只是定义没有初始化，则静态存储变量被隐式初始化为 nil。

Lua 变量声明

正如在上面例子看到的那样，为多个变量赋值就是在变量列表后跟上值列表。例子 local d，f = 5，10 中,变量列表是 d，f，值列表是 5，10。

Lua 赋值时会将第一个值赋给第一个变量，第二个值赋给第二个变量，依次类推。所以，d 的值是 5,f 的值是 10。

示例

下面的示例中，变量被声明在顶部，但是它们在主函数中定义和初始化:

```
-- 变量定义:
local a, b
-- 初始化
a = 10
b = 30
print("value of a:", a)
print("value of b:", b)
-- 交换变量的值
b, a = a, b
print("value of a:", a)
print("value of b:", b)
f = 70.0/3.0
print("value of f", f)
```

上面的代码被编译生成和执行后，会产生如下的结果:

```
value of a:    10
value of b:    30
value of a:    30
value of b:    10
value of f    23.333333333333
```

Lua 中的左值与右值

Lua 中有两种表达式:

- 左值：引用内存位置的表达式被称之为左值表达式。左值表达式既可以出现在赋值符号的左边也可以出现在赋值符号的右边。
- 右值：术语“右值”指存在内存某个位置的数据值。我们不能为右值表达式赋值，也就是说右值表达式只可能出现在赋值符号的右边，而不可能出现在赋值符号的左边。

变量属于左值表达式，所以它可以现在赋值符号的左边。数值常量属于右值表达式，它不能被赋值也不能出现在赋值符号的左边。下面是合法的语句：

```
g = 20
```

但是，下面的语句是非法的，它会产生生成时错误：

```
10 = 20
```

在 Lua 语言中，除了上面讲的这种赋值，还允许在一个赋值语句中存在多个左值表达式与多个右值表达式。如下所示：

```
g,l = 20,30
```

在这个语句中，g 被赋值为 20，l 被赋值为 30。

数据类型

Lua 是动态类型编程语言，变量没有类型，只有值才有类型。值可以存储在变量中，作为参数传递或者作为返回值。

尽管在 Lua 中没有变量数据类型，但是值是有类型的。下面的列表中列出了数据类型：

值类型	描述
nil	用于区分值是否有数据，nil 表示没有数据。
boolean	布尔值，有真假两个值，一般用于条件检查。
number	数值，表示实数(双精度浮点数)。
string	字符串。
function	函数，表示由 C 或者 Lua 写的方法。
userdata	表示任意 C 数据。
thread	线程，表示独立执行的线程，它被用来实现协程。
table	表，表示一般的数组，符号表，集合，记录，图，树等等，它还可以实现关联数组。它可以存储除了 nil 外的任何值。

type 函数

Lua 中有一个 type 函数，它可以让我们知道变量的类型。下面的代码中给出了一些例子：

```
print(type("What is my type"))   --> string
t=10
print(type(5.8*t))              --> number
print(type(true))               --> boolean
print(type(print))              --> function
print(type(type))               --> function
print(type(nil))               --> nil
print(type(type(ABC)))          --> string
```

在 Linux 系统中运行上面的代码可以得到如下的结果：

```
string
number
function
function
```

```
boolean
nil
string
```

默认情况下，在被初始化或赋值前，所有变量都指向 nil。 Lua 中空字符串和零在条件检查时，都被当作真。所以你在使用布尔运算的时候要特别注意。在下一章中，我们会了解到更多关于这些类型的知识。

操作符

操作符是用于告诉解释器执行特定的数学或逻辑运算的符号。Lua 语言有丰富的内置操作符，主要包括以下几类:

- 算术运算操作符
- 关系运算操作符
- 逻辑运算操作符
- 其它操作符

这篇教程将会依次介绍以上四类操作符。

算术去处操作符

下面的表中列出了所有 Lua 语言支持的算术运算操作符。假设 A 变量的值为 10，B 变量的值为 20，则:

操作符	描述	示例
==	判断两个操作数是否相等，若相等则条件为真，否则为假。	(A == B) 为假。
~=	判断两个操作数是否相等，若不相等则条件为真，否则为假。	(A ~= B) 为真。
>	如果左操作数大于右操作数则条件为真，否则条件为假。	(A > B) 为假。
<	如果左操作数小于右操作数则条件为真，否则条件为假。	(A < B) 为真。
>=	如果左操作数大于或等于右操作数则条件为真，否则条件为假。	(A >= B) 为假。
<=	如果左操作数小于或等于右操作数则条件为真，否则条件为假。	(A <= B) 为真。

逻辑运算符

下面的表列出了 Lua 支持的所有逻辑运算符。假设 A 的值为 真（非零），B 的值为 假（零），则:

操作符	描述	示例
and	逻辑与运算符。如果两个操作数都非零，则条件为真。	(A and B) 为假。
or	逻辑或运算符。如果两个操作数中其中有一个非零，则条件为真。	(A or B) 为真。
not	逻辑非运算符。翻转操作数的逻辑状态。如果条件是真，则逻辑非运算符会将其变成假。	!(A and B) 为真。

其它操作符

Lua 语言还支持另外两个操作符：

操作符	描述	示例
..	连接两个字符串。	若 a 为 "Hello"，b 为 "World",则 a..b 返回 "Hello World"。
#	一元运算符，返回字符串或者表的长度。	#"Hello" 返回 5。

操作符优先级

操作符的优先级将决定表达式中的项如何组合。这会影响到表达式的求值。一些操作符比另外一些操作符有更高的优先级。例如，乘法操作符优先级比加法操作符更高。

例如 x = 7 +3*2，这里 x 的值为 13，而不是 20。这是因为操作符 * 优级级比操作符 + 优先级更高，所以先得到 3*2 的乘积，然后再加上 7。

下面的表中，从上到下优先级递减。在每个表达式中，高优先级操作数先运算。

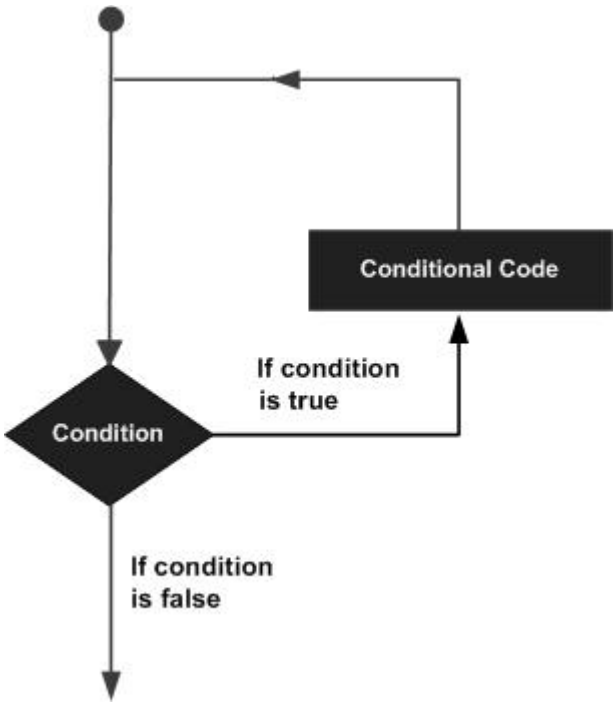
分类	操作数	结合性
一元运算符类	not # -	从右至左
连接运算符	..	从右至左
乘除运算符类	* / %	从左至右
加减运算符类	+ -	从左至右
关系运算符类	< > <= >= == ~=	从左至右
逻辑与运算符	and	从左至右
逻辑或运算符	or	从左至右

循环

虽然一般情况下，语句都是顺序执行的：函数内的第一条语句先执行，然后是第二条，依次类推。 但是还是可能存在需要执行一段代码多次的情况。

为此编程语言提供各式各样的控制结构实现复杂的程序执行路径。

其中，循环语句可以让我们可以执行一条或一组语句多次。下图中所描述的是大多数语言中循环语句的形式：



Lua 语言提供了如下几种循环结构语句。点击链接可查看详细说明。

循环类型	描述
while 循环 (while.md)	先检测条件，条件为真时再执行循环体，直到条件为假时结束。
for 循环 (for.md)	执行一个语句序列多次，可以简化管理循环变量的代码。
repeat...until 循环 (repeat-until.md)	重复执行一组代码语句，直到 until 条件为真为止。
嵌套循环 (nested-loop.md)	可以在一个循环语句中再使用一个循环语句。

循环控制语句

循环控制语句改变循环正常的执行顺序。当离开一个作用域时，在该作用域内自动创建的对象都会被自动销毁。

Lua 支持如下所示的循环控制语句。点击下面的链接查看详细内容：

循环控制语句	描述
break (break.md)	break 语句结束循环，并立即跳转至循环或 switch 语句后的第一条语句处开始执行。

无限循环

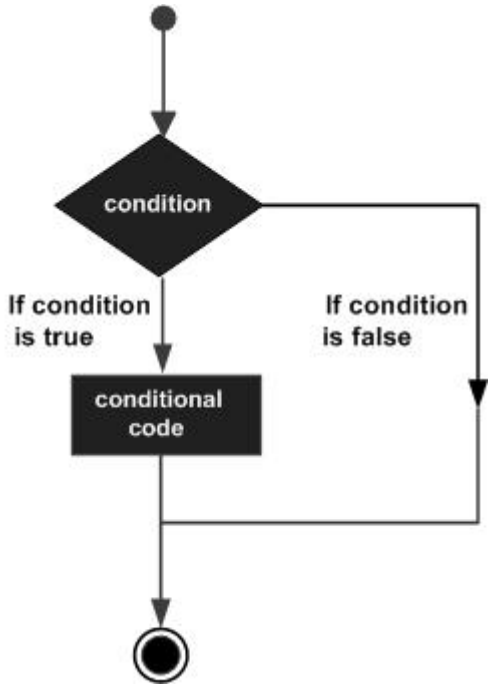
如果循环条件永远不可能为假，则此循环为无限循环。while 语句经常被当作无限循环语句使用。因为我们可以直接将其条件设置为真，这样 while 就会一直循环下去。在无限循环中，可以使用 break 跳出循环。

```
while( true )
do
   print("This loop will run forever.")
end
```

决策

决策结构要求程序开发人员设置一个或多个计算条件。如果条件计算结果为真，则执行一个或多个语句；如果条件为假，则执行另外的语句。

下面是大多数程序语言中的决策结构的一般形式：



Lua 语言中所有布尔真和非 nil 值都当作真；把所有的布尔假和 nil 作为假。请注意，Lua 中的零会被当作真，而其它大部分语言会将零当作假。

Lua 语言提供了如下几类决策语句。点击下面的链接查看详细内容。

语句	描述
if 语句 (./if-statement.md)	if 语句中包括一个布尔表达式和一个或多个语句。
if...else 语句 (./if-else-if-statement.md)	if 语句也可以选择和 esle 语句一起使用。当条件为假时，则执行 else 语句。
嵌套 if 语句 (./nested-if-statement.md)	在 if 语句或者 else if 语句内使用 if 或者 else if。

函数

函数用于将一组语句组合起来完成一个任务。你可以将你的代码分割到不同的函数中。如何将你的代码分到不同的函数中完全由你自己决定，不过一般会按照逻辑功能进行划分，每个函数都执行一个特定的任务。

在 Lua 中提供了大量的内置函数供我们使用。例如，print() 函数用于将输入的参数输出到终端。

函数往往也被称作方法，子例程或过程等等。

函数定义

Lua 中函数定义的语法如下所示：

```
optional_function_scope function function_name( argument1, argument2, argument3..., argumentn)
function_body
return result_params_comma_separated
end
```

Lua 中函数定义包括函数头和函数名两部分。如下列出函数的所有部分：

- 可选的函数作用域：你可以使用关键字 local 限制函数的作用域，你也可以忽略此部分而使用默认值。函数作用域默认是全局。
- 函数名：函数的真正名称。函数名与函数的参数列表一起被称为函数签名。
- 参数：一个参数就一个占位符一样。函数调用时，把值传递给参数。这个值被称之为实际参数或直参数。参数列表指参数的类型，顺序与数量。参数是可选的，一个函数可以没有参数。
- 函数体：函数体是代码语句集合，定义了函数的功能。
- 返回：在 Lua 中，可以使用 return 关键字同时返回多返回值，每个返回值之间使用逗号分隔。

示例

下面是函数 max() 源代码。此函数接受两个参数 num1 与 num2，返回两个输入参数的最大值。

```
--[[ function returning the max between two numbers --]]
function max(num1, num2)

  if (num1 > num2) then
    result = num1;
```

```
  else
    result = num2;
  end

  return result;
end
```

函数参数

如果函数需要用到参数，则它必须声明接受参数值的变量。这些被声明的变量被称为函数的形式参数或简称形参。

函数的形参与函数中其它局部变量一样，在函数的入口处被创建，函数结束时被销毁。

调用函数

创建函数的时候，我们已经定义了函数做什么。接下来，我们就可以调用函数来完成已定义的任务或功能。

当程序中调用一个函数时，程序的控制转移到被调用的函数中。被调用的函数执行定义的任务；当 return 语句被执行或者到达函数末尾时，程序的控制回到主程序中。

调用函数的方法很简单，你只需要将函数要求的参数传递给函数就可以实现函数的调用。如果函数有返回值，你也可以将函数的返回值存储下来。如下如示：

```
function max(num1, num2)
  if (num1 > num2) then
    result = num1;
  else
    result = num2;
  end

  return result;
end

-- 调用函数
print("The maximum of the two numbers is ",max(10,4))
print("The maximum of the two numbers is ",max(5,6))
```

执行上面的代码，可以得到如下的输出结果：

```
The maximum of the two numbers is    10
The maximum of the two numbers is    6
```

赋值与传递函数

在 Lua 语言中，我们可以将函数赋值给一个变量，也可以将函数作为参数传递给另外一个函数。下面是赋值传递函数的一个例子：

```
myprint = function(param)
  print("This is my print function -  ##",param,"##")
end

function add(num1,num2,functionPrint)
  result = num1 + num2
  functionPrint(result)
end
myprint(10)
add(2,5,myprint)
```

执行上面的代码，可以得到如下的输出结果：

```
This is my print function -  ##  10  ##
This is my print function -  ##  7  ##
```

变参函数

在 Lua 语言中，使用 ... 作为参数可以创建参数个数可变的函数，即变参函数。我们可以使用下面的这个例子来理解变参函数的概念。下面的这个例子中函数返回输入参数的平均值：

```
function average(...)
  result = 0
  local arg={...}
  for i,v in ipairs(arg) do
    result = result + v
  end
  return result/#arg
end

print("The average is",average(10,5,3,4,5,6))
```

执行上面的代码，可以得到如下的输出结果：

```
The average is  5.5
```

字符串

字符串就是一个由字符或控制字符组成的序列。字符串可以用以下三种方式任意一种进行初始化。

- 单引号字符串
- 双引号字符串
- [[和]]之间的字符串

上面三种初始化方式的示例如下：

```
string1 = "Lua"
print("\"String 1 is\"",string1)
string2 = 'Tutorial'
print("String 2 is",string2)

string3 = [["Lua Tutorial"]]
print("String 3 is",string3)
```

运行上面的程序，我们可以得到如下的输出结果：

```
"String 1" is    Lua
String 2 is    Tutorial
String 3 is    "Lua Tutorial"
```

字符串中转义字符用于改变字符的一般正常的解释。在上面的例子中，输出双引号（""）的时候，我们使用的是\"。下表列出了转义序列及相应的使用方法：

转义序列	用法
\a	响铃
\b	退格
\f	换页
\n	换行
\r	回车
\t	制表符
\v	垂直制表符
\\	反斜线
\"	双引号
\'	单引号
\[	左方括号
\]	右方括号

字符串操作

Lua 支持如下的字符串操作方法：

S.N.	函数及其功能
1	string.upper(argument):将输入参数全部字符转换为大写并返回。
2	string.lower(argument):将输入参数全部字符转换为小写并返回。
3	string.gsub(maingString,findString,replaceString):将 mainString 中的所有 findString 用 replaceString 替换并返回结果。
4	string.strfind(mainString,findString,optionalStartIndex,optionalEndIndex):在主字符串中查找 findString 并返回 findString 在主字符串中的开始和结束位置，若查找失败则返回 nil。
5	string.reverse(arg):将输入字符串颠倒并返回。
6	string.format(...):返回格式化后的字符串。
7	string.char(arg) 和 string.byte(arg):前者返回输出参数的所代表的字符，后者返回输入参数（字符）的数值。
8	string.len(arg):返回输入字符串的长度。
9	string.rep(string,n): 将输入字符串 string 重复 n 次后的新字符串返回。
10	..:连接两个字符串。

接下来我们用一些例子来讲解如何使用上面这些函数。

大小写操作函数

下面的代码将字符串中字符全部转换成大写或小写：

```
string1 = "Lua";
print(string.upper(string1))
print(string.lower(string1))
```

执行上面的代码可以得到如下的输出结果：

```
LUA
lua
```

替换子串

用一个字符串替换字符串的某子串的示例代码如下：

```
string = "Lua Tutorial"
-- 替换字符串
newstring = string.gsub(string,"Tutorial","Language")
print("The new string is",newstring)
```

执行上面的代码可以得到如下的输出结果：

```
The new string is    Lua Language
```

查找与颠倒

查找一个子串的索引与颠倒一个字符串函数的示例代码如下所示：

```
string = "Lua Tutorial"
-- 替换字符串
print(string.find(string,"Tutorial"))
reversedString = string.reverse(string)
print("The new string is",reversedString)
```

执行上面的代码可以得到如下的输出结果：

```
5    12
The new string is    lairotuT auL
```

格式化字符串

在编程过程中，我们经常需要将字符串以某种格式输出。此时，你就可以使用 string.format 函数格式化你的输出内容。如下所示：

```
string1 = "Lua"
string2 = "Tutorial"
number1 = 10
number2 = 20
-- 基本字符串格式
print(string.format("Basic formatting %s %s",string1,string2))
-- 日期格式化
date = 2; month = 1; year = 2014
print(string.format("Date formatting %02d/%02d/%03d", date, month, year))
-- 符点数格式化
print(string.format("%.4f",1/3))
```

执行上面的代码可以得到如下的输出结果：

```
Basic formatting Lua Tutorial
Date formatting 02/01/2014
0.3333
```

字符与字节表示

字节表示函数用于将字符的内部表示转换为字符表示，而字符表示函数正好相反。 示例代码如下：

```
-- 字节转换
-- 第一个字符
print(string.byte("Lua"))
-- 第三个字符
print(string.byte("Lua",3))
-- 倒数第一个字符
print(string.byte("Lua",-1))
-- 第二个字符
print(string.byte("Lua",2))
-- 倒数第二个字符
print(string.byte("Lua",-2))

-- 内部 ASCII 字值转换为字符
print(string.char(97))
```

执行上面的代码可以得到如下的输出结果：

```
76
97
97
117
117
a
```

其它常用函数

其它常用的字符串处理函数包括字符串连接，字符串长度函数以及重复字符串多次的函数。它们的使用方法示例如下：

```
string1 = "Lua"
string2 = "Tutorial"
-- 用 .. 连接两个字符串
print("Concatenated string",string1..string2)

-- 字符串的长度
```

```
print("Length of string1 is ",string.len(string1))

-- 重复字符串
repeatedString = string.rep(string1,3)
print(repeatedString)
```

执行上面的代码可以得到如下的输出结果：

```
Concatenated string    LuaTutorial
Length of string1 is     3
LuaLuaLua
```

数组

数组是一组有序的对象排列，既可以是一维的也可以是多维的。

在 Lua 语言中，数组是用整数索引表实现的。数组的大小并不固定，随着数组元素的增加，它可以动态地增加内存空间大小。

一维数组

一维数组可以使用一个简单的表结构表示。可以通过一个简单循环初始化数组或者读取数组中数据。示例代码如下所示：

```
array = {"Lua", "Tutorial"}

for i= 0, 2 do
   print(array[i])
end
```

执行上面的代码可以得到如下的输出结果：

```
nil
Lua
Tutorial
```

从上面的例子中可以看出，当我们尝试着访问数组中一个不存在的索引时，会得到 nil 值。 Lua 语言与 C 语言不同，Lua 数组的索引是从 1 开始的，而 C 语言中索引是从 0 开始的。不过呢，你也可以在索引值为 0 或小于 0 的位置创建对象。下面的代码演示了如何使用负索引值创建并初始化数组：

```
array = {}

for i= -2, 2 do
   array[i] = i *2
end

for i = -2,2 do
   print(array[i])
end
```

执行上面的代码可以得到如下的输出结果：

```
-4
-2
0
2
4
```

多维数组

多维数组有以下两种实现方式：

1. 数组的数组（译注：数组的每一个元素是一个数组）。
2. 修改一维数组的索引值（译注：将多维数组映射到一维数组中）。

使用方法一创建 3x3 的二维数组：

```
-- 初始化数组
array = {}
for i=1,3 do
   array[i] = {}
      for j=1,3 do
         array[i][j] = i*j
      end
end

-- 访问数组元素
for i=1,3 do
   for j=1,3 do
      print(array[i][j])
   end
end
```

执行上面的代码可以得到如下的输出结果：

```
1
2
3
2
4
6
3
6
9
```

通过修改数组的的索引值实现 3x3 的二维数组，示例代码如下：

```
-- 初始化数组
array = {}
maxRows = 3
maxColumns = 3
for row=1,maxRows do
   for col=1,maxColumns do
      array[row*maxColumns +col] = row*col
   end
end

-- 访问数组元素
for row=1,maxRows do
   for col=1,maxColumns do
      print(array[row*maxColumns +col])
   end
end
```

执行上面的代码可以得到如下的输出结果：

```
1
2
3
2
4
6
3
6
9
```

正如从上面例子中所看到的那样，数组中数据是基于索引存储的。这使得数组可以以稀疏的方式存储，这也是 Lua 矩阵的存储方式。正是因为 Lua 中不会存储 nil 值，所以 Lua 不需要使用任何特殊的技术就可以节约大量的空间，这一点在其它语言中是做不到的。

迭代器

迭代器是用于遍历集合或容器中元素的一种结构。在 Lua 语言中，集合往往指的是可以用来创建各种数据结构的表。比如，数组就是用表来创建的。

通用迭代器

通用迭代器可以访问集合中的键值对。下面是通用迭代器的一个简单例子：

```
array = {"Lua", "Tutorial"}

for key,value in ipairs(array)
do
   print(key, value)
end
```

执行的上面的代码，我们可以得到如下的输出结果：

```
1  Lua
2  Tutorial
```

上面的例子中使用了 Lua 提供的默认迭代器函数 ipairs。

在 Lua 语言中，我们使用函数表示迭代器。根据是否在迭代器函数中是否维护状态信息，我们将迭代器分为以下两类：

- 无状态迭代器
- 有状态迭代器

无状态迭代器

由此迭代器的名称就可以看出来，这一类的迭代器函数中不会保存任何中间状态。

让我们一起来看一下下面这个例子。在这个例子中，我们用一个简单的函数创建了一个自己的迭代器。这个迭代器用以输出 n 个数的平方值。

```
function square(iteratorMaxCount,currentNumber)
   if currentNumber<iteratorMaxCount
   then
```

```
    currentNumber = currentNumber+1
  return currentNumber, currentNumber*currentNumber
  end
end

for i,n in square,3,0
do
  print(i,n)
end
```

执行上面的代码，我们可以得到如下的输出结果：

```
1   1
2   4
3   9
```

我们可以稍微的修改一下上面的代码，使得此迭代器可以像 ipairs 那样工作。如下所示：

```
function square(iteratorMaxCount,currentNumber)
  if currentNumber<iteratorMaxCount
  then
    currentNumber = currentNumber+1
  return currentNumber, currentNumber*currentNumber
  end
end

function squares(iteratorMaxCount)
  return square,iteratorMaxCount,0
end

for i,n in squares(3)
do
   print(i,n)
end
```

执行上面的代码，我们可以得到如下的输出结果：

```
1   1
2   4
3   9
```

有状态迭代器

前面的例子使用的迭代器函数是不保存状态的。每次调用迭代器函数时，函数基于传入函数的第二个变量访问集合的下一个元素。在 Lua 中可以使用闭包来存储当前元素的状态。闭包通过函数调用得到变量的值。为了创建一个新的闭包，我们需创建两个函数，包括闭包函数本身和一个工厂函数，其中工厂函数用于创建闭包。

下面的示例中，我们将使用闭包来创建我们的迭代器。

```
array = {"Lua", "Tutorial"}

function elementIterator (collection)
  local index = 0
  local count = #collection
  -- 返回闭包函数
  return function ()
    index = index + 1
    if index <= count
    then
      -- 返回迭代器的当前元素
      return collection[index]
    end
  end
end

for element in elementIterator(array)
do
  print(element)
end
```

执行上面的代码，我们可以得到如下的输出结果：

```
Lua
Tutorial
```

上面的例子中我们可以看到，在 elementIterator 函数内定义了另外一个匿名函数。此匿名函数中使用了一个外部变量 index (译注：此变量在匿名函数之外，elementIterator 函数内)。每次内部的匿名函数被调用时，都会将 index 的值增加 1，并统计数返回的每个元素。

我们可以参照上面的方法使用闭包创建一个迭代器函数。每次我们使用迭代器遍历集合时，它都可以返回多个元素。

表

在 Lua 语言中，表是唯一可以用来创建不同数据类型的数据结构，比如常见的数组和字典都是用表来创建的。 Lua 语言中经常到关联数组这种数据类型，它不仅可以用数值作为索引值，除了 nil 以外的字符串同样可以作为其索引。表没有固定的大小，当数据量增加时表会自动增大。

Lua 语言中的各种结构表示都用到了表，包括包（package）的表示。当我们使用方法 string.format 时，我们用到的其实是包 string 中的方法 format。

使用表

表被称之为对象，它既不是值也不是变量。Lua 用构造表达式 {} 创建一个空表。需要注意的是，在存储表的变量和表本身之间没有什么固定的对应关系（译注：一个表可以被不同的变量引用，一个变量也可以随时改变其所引用的表对象）。

```
--简单的表初始化
mytable = {}

--简单的表赋值
mytable[1]= "Lua"

--移除引用
mytable = nil
-- lua 的垃圾回收机制负责回收内存空间
```

当我们有一个拥有一系列元素的表时，如果我们将其赋值给 b。那么 a 和 b 都会引用同一个表对象(a 先引用该表)，指向相同的内存空间。而不会再单独为 b 分配内存空间。即使给变量 a 赋值 nil，我们仍然可以用变量 b 访问表本身。如果已经没有变量引用表时，Lua　语言垃圾回收机制负责回收不再使用的内存以被重复使用。

下面的示例代码使用到了上面提到的表的特性。

```
-- 声明空表
mytable = {}
print("Type of mytable is ",type(mytable))

mytable[1]= "Lua"
mytable["wow"] = "Tutorial"
print("mytable Element at index 1 is ", mytable[1])
print("mytable Element at index wow is ", mytable["wow"])
```

```
-- alternatetable 与 mytable 引用相同的表
alternatetable = mytable

print("alternatetable Element at index 1 is ", alternatetable[1])
print("mytable Element at index wow is ", alternatetable["wow"])

alternatetable["wow"] = "I changed it"

print("mytable Element at index wow is ", mytable["wow"])

-- 只是变量被释放，表本身没有被释放
alternatetable = nil
print("alternatetable is ", alternatetable)

-- mytable 仍然可以访问
print("mytable Element at index wow is ", mytable["wow"])

mytable = nil
print("mytable is ", mytable)
```

执行上面的代码，我们可以得到如下的输出结果：

```
Type of mytable is    table
mytable Element at index 1 is    Lua
mytable Element at index wow is    Tutorial
alternatetable Element at index 1 is    Lua
mytable Element at index wow is    Tutorial
mytable Element at index wow is    I changed it
alternatetable is    nil
mytable Element at index wow is    I changed it
mytable is    nil
```

表的操作函数

下面的表中列出了 Lua 语言内置的表操作函数，具体内容如下所示：

S.N.	方法与作用
1	table.concat(table[, sep [, i[,j]]]): 根据指定的参数合并表中的字符串。具体用法参考下面的示例。
2	table.insert(table,[pos,]value):在表中指定位置插入一个值。
3	table.maxn(table)：返回表中最大的数值索引。
4	table.remove(table[,pos]):从表中移出指定的值。
5	table.sort(table[,comp]):根据指定的（可选）比较方法对表进行排序操作。

让我们一起看一些上述函数使用的例子。

表连接操作

我们可以使用表连接操作连接表中的元素，如下所示。

```
fruits = {"banana","orange","apple"}
-- 返回表中字符串连接后的结果
print("Concatenated string ",table.concat(fruits))

--用字符串连接
print("Concatenated string ",table.concat(fruits,", "))

--基于索引连接 fruits
print("Concatenated string ",table.concat(fruits,", ", 2,3))
```

执行上面的代码，我们可以得到如下的输出结果：

```
Concatenated string    bananaorangeapple
Concatenated string    banana, orange, apple
Concatenated string    orange, apple
```

插入与移出操作

插入和移除表中元素是对表最常见的操作。使用方法如下所示：

```
fruits = {"banana","orange","apple"}

-- 在 fruits 的末尾插入一种水果
table.insert(fruits,"mango")
print("Fruit at index 4 is ",fruits[4])

-- 在索引 2 的位置插入一种水果
table.insert(fruits,2,"grapes")
print("Fruit at index 2 is ",fruits[2])

print("The maximum elements in table is",table.maxn(fruits))

print("The last element is",fruits[5])
table.remove(fruits)
print("The previous last element is",fruits[5])
```

执行上面的代码，我们可以得到如下的输出结果：

```
Fruit at index 4 is    mango
Fruit at index 2 is    grapes
The maximum elements in table is   5
The last element is    mango
The previous last element is    nil
```

表排序操作

在程序开发过程中，常常有对表排序的需求。 sort 函数默认使用字母表对表中的元素进行排序（可以通过提供比较函数改变排序策略）。示例代码如下：

```
fruits = {"banana","orange","apple","grapes"}
for k,v in ipairs(fruits) do
print(k,v)
end
table.sort(fruits)
print("sorted table")
for k,v in ipairs(fruits) do
print(k,v)
end
```

执行上面的代码，我们可以得到如下的输出结果：

```
1   banana
2   orange
3   apple
4   grapes
sorted table
1   apple
2   banana
3   grapes
4   orange
```

模块

什么是模块?

Lua 中的模块与库的概念相似，每个模块都有一个全局唯一名字，并且每个模块都包含一个表。使用一个模块时，可以使用 require 加载模块。模块中可以包括函数和变量，所有这些函数和变量被表存储于模块的表中。模块中的表的功能类似于命名空间，用于隔离不同模块中的相同的变量名。在使用模块的时候，我们应该遵守模块定义的规范，在 require 加载模块时返回模块中的包含函数和变量的表对象。

Lua 模块的特别之处

模块中表的使用使得我们可在绝大多数情况下可以像操作其它表一样操作模块。由于 Lua 语言允许对模块本身进行操作，所以 Lua 也就具备了许多其它语言需要特殊机制才能实现的特殊性质。例如，这种自由的表操作机制使得编程人员可以用多种方法调用模块中的函数。下面的例子演示了其中的一些方法：

```
-- 假设我们有一个板块 printFormatter
-- 该模块有一个函数 simpleFormat(arg)
-- 方法 1
require "printFormatter"
printFormatter.simpleFormat("test")

-- 方法 2
local formatter = require "printFormatter"
formatter.simpleFormat("test")

-- 方法 3
require "printFormatter"
local formatterFunction = printFormatter.simpleFormat
formatterFunction("test")
```

从上面的例子中可以看出，Lua 不需要任何额外的代码就可以实现非常灵活的编程技巧。

require 函数

Lua 提供了一个高层次抽象的函数 require，使用这个函数可以加载所有需要的模块。在设计之初，这个函数就被设计的尽可能的简单，以避免加载模块时需要太多的模块信息。require 函数加载模块时把所有模块都只当作一段定义了变量的代码（事实上是一些函数或者包含函数的表）而已，完全不需要更多的模块信息。

示例

让我们看一下面这个例子。在这个例子中，我们定义了模块 mymath,在这个模块中定义一些数学函数，并将该模块存储于 mymath.lua 文件中。具体内容如下：

```
local mymath = {}
function mymath.add(a,b)
  print(a+b)
end

function mymath.sub(a,b)
  print(a-b)
end

function mymath.mul(a,b)
  print(a*b)
end

function mymath.div(a,b)
  print(a/b)
end

return mymath
```

接下来，我们在另一个文件 moduletutorial.lua 文件中访问这个模块。具体代码如下所示：

```
mymathmodule = require("mymath")
mymathmodule.add(10,20)
mymathmodule.sub(30,20)
mymathmodule.mul(10,20)
mymathmodule.div(30,20)
```

运行这段代码这前，我们需要将两个 lua 源代码文件放在同一目录下，或者把模块代码文件放在包路径下（这种情况需要额外的配置）。运行上面的代码，可以得到如下的输出结果：

```
30
10
200
1.5
```

注意事项

- 把模块和待运行的文件放在相同的目录下。
- 模块的名称与文件名称相同。
- 为 require 函数返回模块（在模块中使用 return 命令返回存储了函数和变量的表）。尽管有其它的模块实现的方式，但是建议您使用上面的实现方法。

更老的实现模块的方法

下面，我们将用 package.seall 这种比较老的方法重新实现上面的例子。这种实现方法主要用于 Lua 5.1 或 5.2 版本。使用这种方式实现模块的代码如下所示：

```
module("mymath", package.seeall)

function mymath.add(a,b)
  print(a+b)
end

function mymath.sub(a,b)
  print(a-b)
end

function mymath.mul(a,b)
  print(a*b)
end

function mymath.div(a,b)
  print(a/b)
end
```

使用此模块的代码如下所示：

```
require("mymath")
mymath.add(10,20)
mymath.sub(30,20)
mymath.mul(10,20)
mymath.div(30,20)
```

当我们运行这段代码，我们会得到与前面相同的输出结果。但是建议你不要使用这种方式，因为普遍认为这种方式不及新的方法安全。许多用到 Lua 语言的 SDK 都已经不再使用这种方式定义模块，例如, Corna SDK。

元表

正如其名，元表也是表。不过，将元表与表相关联后，我们就可以通过设置元表的键和相关方法来改变表的行为。元方法的功能十分强大，使用元方法可以实现很多的功能，比如：

- 修改表的操作符功能或为操作符添加新功能（译注：如果您学过 C++ 之类的面向对象的语言，应该比较好理解，其实它实现的是操作的重载）。
- 使用元表中的 __index 方法，我们可以实现在表中查找键不存在时转而在元表中查找键值的功能。

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。

首先，让我们看一下如何将一个表设置为另一个表的元表。示例如下：

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

上面的代码可以简写成如下的一行代码：

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

__index

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"}, {
  __index = function(mytable, key)
    if key == "key2" then
      return "metatablevalue"
    else
      return mytable[key]
    end
  end
})

print(mytable.key1,mytable.key2)
```

运行上面的程序，我们可以得到如下的输出结果：

```
value1    metatablevalue
```

接下来逐步解释上面例子运行的过程：

- 表 mytable 为 {key = "values1"}
- 为 mytable 设置了一个元表，该元表的键 __index 存储了一个函数，我们称这个函数为元方法。
- 这个元方法的工作也十分简单。它仅查找索引 “key2”,如果找到该索引值，则返回 "metatablevalue",否则返回 mytable 中索引对应的值。

上面的程序同样可以简化成如下的形式：

```
mytable = setmetatable({key1 = "value1"}, { __index = { key2 = "metatablevalue" } })
print(mytable.key1,mytable.key2)
```

__newindex

为元表添加 __newindex 后，当访问的键在表中不存在时，此时添加新键值对的行为将由此元方法（ __newindex ）定义。下面的例子中，如果访问的索引在表中不存在则在元表中新加该索引值（注意，是添加在另外一个表 mymetatable 中而非在原表 mytable 中。），具体代码如下(译注：请注意此处 __newindex 的值并非一个方法而是一个表。)：

```
mymetatable = {}
mytable = setmetatable({key1 = "value1"}, { __newindex = mymetatable })

print(mytable.key1)

mytable.newkey = "new value 2"
print(mytable.newkey,mymetatable.newkey)

mytable.key1 = "new  value 1"
print(mytable.key1,mymetatable.newkey1)
```

执行上面的程序，我们可以得到如下的输出结果：

```
value1
nil    new value 2
new  value 1    nil
```

可以看出，在上面的程序中，如果键存在于主表中，只会简单更新相应的键值。而如果键不在表中时，会在另外的表 mymetatable 中添加该键值对。

在接下来这个例子中，我们用 rawset 函数在相同的表（主表）中更新键值，而不再是将新的键添加到另外的表中。代码如下所示：

```
mytable = setmetatable({key1 = "value1"}, {
  __newindex = function(mytable, key, value)
        rawset(mytable, key, "\""..value.."\"")

  end
})

mytable.key1 = "new value"
mytable.key2 = 4

print(mytable.key1,mytable.key2)
```

执行上面的程序，我们可以得到如下的输出结果：

```
new value    "4"
```

rawset 函数设置值时不会使用元表中的 __newindex 元方法。同样的，Lua 中也存的一个 rawget 方法，该方法访问表中键值时也不会调用 __index 的元方法。

为表添加操作符行为

使用 + 操作符完成两个表组合的方法如下所示（译注：可以看出重载的意思了）：

```
mytable = setmetatable({ 1, 2, 3 }, {
  __add = function(mytable, newtable)
    for i = 1, table.maxn(newtable) do
      table.insert(mytable, table.maxn(mytable)+1,newtable[i])
    end
    return mytable
  end
})

secondtable = {4,5,6}

mytable = mytable + secondtable
for k,v in ipairs(mytable) do
print(k,v)
end
```

执行上面的的程序，我们可以得到如下的输出结果：

```
1   1
2   2
3   3
4   4
5   5
6   6
```

元表中 __add 键用于修改加法操作符的行为。其它操作对应的元表中的键值如下表所示。

键	描述
__add	改变加法操作符的行为。
__sub	改变减法操作符的行为。
__mul	改变乘法操作符的行为。
__div	改变除法操作符的行为。
__mod	改变模除操作符的行为。
__unm	改变一元减操作符的行为。
__concat	改变连接操作符的行为。
__eq	改变等于操作符的行为。
__lt	改变小于操作符的行为。
__le	改变小于等于操作符的行为。

__call

使用 __call 可以使表具有像函数一样可调用的特性。下面的例子中涉及两个表，主表 mytable 和 传入的实参表结构 newtable，程序完成两个表中值的求和。

```
mytable = setmetatable({10}, {
  __call = function(mytable, newtable)
    sum = 0
    for i = 1, table.maxn(mytable) do
        sum = sum + mytable[i]
    end
    for i = 1, table.maxn(newtable) do
        sum = sum + newtable[i]
    end
    return sum
  end
})
newtable = {10,20,30}
print(mytable(newtable))
```

运行上面的代码，我们可以得到如下的输出结果：

```
70
```

__tostring

要改变 print 语句的行为，我们需要用到 __tostring 元方法。下面是一个简单的例子：

```
mytable = setmetatable({ 10, 20, 30 }, {
  __tostring = function(mytable)
    sum = 0
    for k, v in pairs(mytable) do
        sum = sum + v
    end
    return "The sum of values in the table is " .. sum
  end
})
print(mytable)
```

运行上面的代码，我们可以得到如下的输出结果：

```
The sum of values in the table is 60
```

如果你完全掌握了元表的用法，你就可以实现很多看上面很复杂的操作。如果不使用元表，就不仅仅是看上去很复杂了，而是真的非常复杂。所以，多做一些使用元表的练习，并熟练掌握所有元表的可选项，这会让你受益匪浅。

协程

概述

协程具有协同的性质，它允许两个或多个方法以某种可控的方式协同工作。在任何一个时刻，都只有一个协程在运行，只有当正在运行的协程主动挂起时它的执行才会被挂起（暂停）。

上面的定义可能看上去比较模糊。接下来让我讲得很清楚一点，假设我们有两个方法，一个是主程序方法，另一个是一个协程。当我们使用 resume 函数调用一个协程时，协程才开始执行。当在协程调用 yield 函数时，协程挂起执行。再次调用 resume 函数时，协程再从上次挂起的地方继续执行。这个过程一直持续到协程执行结束为止。

协程中可用的函数

下面的表中列出 Lua 语言为支持协程而提供的所有函数以及它们的用法。

S.N.	方法和功能
1	coroutine.create(f):用函数 f 创建一个协程，返回 thread 类型对象。
2	coroutine.resume(co[,val1,...]): 传入参数（可选），重新执行协程 co。此函数返回执行状态，也可以返回其它值。
3	coroutine.running():返回正在运行的协程，如果在主线程中调用此函数则返回 nil。
4	coroutine.status(co):返回指定协程的状态，状态值允许为：正在运行(running)，正常(normal)，挂起(suspended)，结束(dead)。
5	coroutine.wrap(f):与前面 coroutine.create 一样，coroutine.wrap 函数也创建一个协程，与前者返回协程本身不同，后者返回一个函数。当调用该函数时，重新执行协程。
6	coroutine.yield(...):挂起正在执行的协程。为此函数传入的参数值作为执行协程函数 resume 的额外返回值（默认会返回协程执行状态）。

示例

让我们通过下面的例子来理解一下协程这个概念。

```
co = coroutine.create(function (value1,value2)
   local tempvar3 =10
   print("coroutine section 1", value1, value2, tempvar3)
   local tempvar1 = coroutine.yield(value1+1,value2+1)
   tempvar3 = tempvar3 + value1
```

```
  print("coroutine section 2",tempvar1 ,tempvar2, tempvar3)
  local tempvar1, tempvar2= coroutine.yield(value1+value2, value1-value2)
  tempvar3 = tempvar3 + value1
  print("coroutine section 3",tempvar1,tempvar2, tempvar3)
  return value2, "end"
end)

print("main", coroutine.resume(co, 3, 2))
print("main", coroutine.resume(co, 12,14))
print("main", coroutine.resume(co, 5, 6))
print("main", coroutine.resume(co, 10, 20))
```

执行上面的程序，我们可以得到如下的输出结果：

```
coroutine section 1    3    2    10
main    true    4    3
coroutine section 2    12    nil    13
main    true    5    1
coroutine section 3    5    6    16
main    true    2    end
main    false    cannot resume dead coroutine
```

上面的例子到底做了些什么呢?

和前面说到的一样，在例子中我们使用 resume 函数继续执行协程，用 yield 函数挂起协程。同样，从例子中也可以看出如何为执行协程的 resueme 函数返回多个值。下面我将逐步解释上面的代码。

- 首先，我们创建了一个协程并将其赋给变量 co。此协程允许传入两个参数。
- 第一次调用函数 resume 时，协程内局部变量 value1 和 value2 的值分别为 3 和 2。
- 为了便于理解，我们使用了局部变量 tempvar3 该变量被初始化为 10。由于变量 value1 的值为3，所以 tempvar3 在随后的协程调用过程中被先后更新为 13 和 16。
- 第一次调用 coroutine.yield 时，为 resume 函数返回了值 4 和 3，这两个值是由传入的参数 3，2 分别加 1 后的结果，这一点可以从 yield 语句中得到证实。除了显示指定的返回值外，resume 还收到隐式的返回值 true，该值表示协程执行的状态，有 true 和 false 两个可能取值。
- 上面的例子中，我们还应该关注在下一次调用 resume 时如何为协程传入参数。从例子中可以看到，coroutine.yield 函数返回后为两个变量赋值，该值即是第二次调用 resume 时传入的参数。这种参数传递的机制让可以结合前面传入的参数完成很多新的操作。
- 最后，协程中所有语句执行完后，后面的调用就会返回 false 状态，同时返回 "cannot resume dead coroutine"消息。

另一个协程的示例

下面这例子中的协程使用 yield 函数和 resume 函数依次返回数字 1 到 5。示例中，如果没有协程对象或对象已结束（dead），则重新创建一个新的协程对象；若协程已经存在，则执行已经存在的协程。

```
function getNumber()
  local function getNumberHelper()
    co = coroutine.create(function ()
    coroutine.yield(1)
    coroutine.yield(2)
    coroutine.yield(3)
    coroutine.yield(4)
    coroutine.yield(5)
    end)
    return co
  end
  if(numberHelper) then
    status, number = coroutine.resume(numberHelper);
    if coroutine.status(numberHelper) == "dead" then
      numberHelper = getNumberHelper()
      status, number = coroutine.resume(numberHelper);
    end
    return number
  else
    numberHelper = getNumberHelper()
    status, number = coroutine.resume(numberHelper);
    return number
  end
end
for index = 1, 10 do
  print(index, getNumber())
end
```

执行上述的程序，我们可以得到如下的输出结果：

```
1   1
2   2
3   3
4   4
5   5
6   1
7   2
8   3
```

```
9    4
10    5
```

大家经常会把协程和多线程编程语言中的线程进行对比，但我们要明白，协程有着与线程类似的特性，但是协程与线程的区别在于协程不能并发，任意时刻只会有一个协程执行，而线程允许并发的存在。（译注：译者认为本质上协程其是就是线程，不过是用户态的线罢了，它将调度问题交由程序开发人员手动完成。）

我们通过控制程序执行顺序以满足获取某些临时信息的需求。配合全局变量的使用，协和会变得更加的灵活方便。

文件 I/O

Lua 的 IO 库用于读取或操作文件。Lua IO 库提供两类文件操作，它们分别是隐式文件描述符(implict file descriptors)和显式文件描述符(explicit file descriptors)。

在接下来的例子的，我们会用到一个示例文件 test.lua，文件内容如下：

```
-- sample test.lua
-- sample2 test.lua
```

简单的打开文件操作可以用如下的语句完成。

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

可选的打开文件的模式如下表所示。

模式	描述
"r"	只读模式，这也是对已存在的文件的默认打开模式。
"w"	可写模式，允许修改已经存在的文件和创建新文件。
"a"	追加模式，对于已存的文件允许追加新内容，但不允许修改原有内容，同时也可以创建新文件。
"r+"	读写模式打开已存的在文件。
"w+"	如果文件已存在则删除文件中数据；若文件不存在则新建文件。读写模式打开。
"a+"	以可读的追加模式打开已存在文件，若文件不存在则新建文件。

隐式文件描述符

隐式文件描述符使用标准输入输出模式或者使用单个输入文件和输出文件。使用隐匿文件描述符的示例代码如下：

```
-- 只读模式打开文件
file = io.open("test.lua", "r")

-- 将 test.lua 设置为默认输入文件
io.input(file)

--打印输出文件的第一行
print(io.read())

-- 关闭打开的文件
io.close(file)
```

```
-- 以追加模式打开文件
file = io.open("test.lua", "a")

-- 将 test.lua 设置为默认的输出文件
io.output(file)

-- 将内容追加到文件最后一行
io.write("-- End of the test.lua file")

-- 关闭打开的文件
io.close(file)
```

执行上面的程序，我们可以看到输出了 test.lua 文件的第一行。在本例中，输出的结果为：

```
- Sample test.lua
```

输出的内容是 test.lua 文件中的第一行。“-- End of the test.lua file” 他会被追加到 test.lua 文件的最后一行。

从上面的例子中，你可以看到隐式的描述述如何使用 io."x" 方法与文件系统交互。上面的例子使用 io.read() 函数时没有使用可选参数。此函数的可选参数包括：

模式	描述
"*n"	从文件当前位置读入一个数字，如果该位置不为数字则返回 nil。
"*a"	读入从当前文件指针位置开始的整个文件内容。
"*l"	读入当前行。
number	读入指定字节数的内容。

另外一些常用的方法：

- io.tmpfile():返回一个可读写的临时文件，程序结束时该文件被自动删除。
- io.type(file):检测输入参数是否为可用的文件句柄。返回 "file" 表示一个打开的句柄；“closed file” 表示已关闭的句柄；nil 表示不是文件句柄。
- io.flush():清空输出缓冲区。
- io.lines(optional file name): 返回一个通用循环迭代器以遍历文件，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。若显示提供了文件句柄，则结束时自动关闭文件；使用默认文件时，结束时不会自动关闭文件。

显示文件描述符

我们也会经常用到显示文件描述符，因为它允许我们同时操作多个文件。这些函数与隐式文件描述符非常相似，只不过我们在这儿使用 file:function_name 而不是使用 io.function_name 而已。下面的例子使用显示文件描述符实现了与前面例子中完全相同的功能。

```
-- 只读模式打开文件
file = io.open("test.lua", "r")

-- 输出文件的第一行
print(file:read())

-- 关闭打开的文件
file:close()

-- 以追加模式打开文件
file = io.open("test.lua", "a")

-- 添加内容到文件的尾行
file:write("--test")

-- 关闭打开的文件
file:close()
```

执行上面的程序，我们可以得到与前面使用隐式文件描述符类似的输出结果：

```
-- Sample test.lua
```

在显式文件描述符中，打开文件的描述与读文件时的参数与隐式文件描述中的完全相同。另外的常用方法包括：

- file:seek(option whence,option offset)：此函数用于移动文件指针至新的位置。参数 whence 可以设置为 “set”，"cur","end"，offset 为一个偏移量值，描述相对位置。如果第一个参数为 "set"，则相对位置从文件开始处开始计算；如果第一个参数为 "cur"，则相对位置从文件当前位置处开始计算； 如果第一个参数为 "end"，则相对位置从文件末尾处开始计算。函数的参数默认值分别为 "cur" 和 0，因此不传递参数调用此函数可以获得文件的当前位置。
- file:flush()：清空输出缓冲区。
- io.lines(optional file name)：提供一个循环迭代器以遍历文件，如果指定了文件名则当遍历结束后将自动关闭该文件；若使用默认文件，则遍历结束后不会自动关闭文件。

下面的例子演示 seek 函数的使用方法。它将文件指针从文件末尾向前移动 25。并使用 read 函数从该位置出输出剩余的文件内容。

```
-- Opens a file in read
file = io.open("test.lua", "r")

file:seek("end",-25)
print(file:read("*a"))

-- closes the opened file
file:close()
```

执行上的面的程序，你可以得到类似下面的输出结果：

```
 sample2 test.lua
--test
```

你还可以尝试不同的参数了解更多的 Lua 文件操作方法。

错误处理

为什么需要错误处理机制

在真实的系统中程序往往非常复杂，它们经常涉及到文件操作、数据库事务操作或网络服务调用等，这个时候错误处理就显得非常重要。不关注错误处理可能在处理诸如涉密或金融交易这些业务时造成重大的损失。
无论什么时候，程序开发都要求小心地做好错误处理工作。在 Lua 中错误可以被分为两类：

- 语法错误
- 运行时错误

语法错误

语法错误是由于不正确的使用各种程序语法造成的，比如错误的使用操作符或表达式。下面即是一个语法错误的例子：

```
a == 2
```

正如你知道的那样，单个等号与双等号是完全不一样的。二者之间随意的替换就导致语法错误。一个等号表示的是赋值，而双等号表示比较。类似地，下面这一小段代码中也存在语法错误：

```
for a= 1,10
  print(a)
end
```

执行上面的这段程序，我们会得到如下的输出结果：

```
lua: test2.lua:2: 'do' expected near 'print'
```

语法错误相比于运行时错误更容易处理，因为 Lua 解释器可以更精确的定位到语法错误的位置。由上面的错误，我们可以容易就知道，在 print 语句前添加 do 语句就可以了，这是 Lua 语法结构所要求的。

运行时错误

对于运行时错误，虽然程序也能成功运行，但是程序运行过程中可能因为错误的输入或者错误的使用函数而导致运行过程中产生错误。下面的例子显示了运行时错误如何产生的：

```
function add(a,b)
  return a+b
end

add(10)
```

当我们尝试生成(build)上面的程序，程序可以正常的生成和运行。但是一旦运行后，立马出现下面的运行时错误。

```
lua: test2.lua:2: attempt to perform arithmetic on local 'b' (a nil value)
stack traceback:
   test2.lua:2: in function 'add'
   test2.lua:5: in main chunk
   [C]: ?
```

这个运行时错误是由于没有正确的为 add 函数传入参数导致的，由于没有为 b 传入值，所有 b 的值为 nil 从而导致在进行加法运算时出错。

Assert and Error 函数

我们经常用到 assert 和 error 两个函数处理错误。下面是一个简单的例子。

```
local function add(a,b)
  assert(type(a) == "number", "a is not a number")
  assert(type(b) == "number", "b is not a number")
  return a+b
end
add(10)
```

执行上面的程序，我们会得到如下的输出结果：

```
lua: test2.lua:3: b is not a number
stack traceback:
   [C]: in function 'assert'
   test2.lua:3: in function 'add'
   test2.lua:6: in main chunk
   [C]: ?
```

error(message [,level]) 函数会结束调用自己的函数，并将 message 作为错误信息返回调用者(译注:保护模式下才会返回调用者，一般情况会结束程序运行并在控制终端输出错误信息)。error 函数本身从不返回。一般地，error 函数会在消息前附上错误位置信息。级别(level) 参数指定错误发生的位置。若其值为 1(默认值)，返回的错误的位置是 error 函数被调用的位置。若为 2, 返回的错误位置为调用 error 函数的函数被调用的位置，依次类推。将 level 参数的值设为 0 就不再需要在消息前增加额外的位置信息了。

pcall 与 xpcall

在 Lua 中，为了避免使用抛出错误和处理错误，我们需要用到 pcall 和 xpcall 函数来处理异常。
使用 pcall(f,arg1,...) 函数可以使用保护模式调用一个函数。如果函数 f 中发生了错误， 它并不会抛出一个错误，而是返回错误的状态。使用的 pcall 函数的方法如下所示：

```
function myfunction ()
   n = n/nil
end

if pcall(myfunction) then
   print("Success")
else
    print("Failure")
end
```

执行上面的程序，我们可以得到如下的输出结果：

```
Failure
```

xpcall(f,err) 函数调用函数 f 同时为其设置了错误处理方法 err，并返回调用函数的状态。任何发生在函数 f 中的错误都不会传播，而是由 xpcall 函数捕获错误并调用错误处理函数 err，传入的参数即是错误对象本身。xpcall 的使用示例如下：

```
function myfunction ()
   n = n/nil
end

function myerrorhandler( err )
   print( "ERROR:", err )
end

status = xpcall( myfunction, myerrorhandler )
print( status)
```

执行上面的程序，我们可以得到如下的输出结果：

```
ERROR:    test2.lua:2: attempt to perform arithmetic on global 'n' (a nil value)
false
```

作为程序开发人员，在程序中正确合理地处理错误是非常重要的。正确地处理错误可以保证发生意外情况不会影响到程序用户的使用。

2

Lua 进阶

调试

Lua 提供一个调试库，这个库中提供了创建自己的调试器所需的所有原语函数。虽然，Lua 没有内置调试器，但是开发者们为 Lua 开发了许多的开源调试器。

Lua 调试库包括的函数如下表所示。

S.N.	方法和描述
1	debug(): 进入交互式调试模式，在此模式下用户可以用其它函数查看变量的值。
2	getfenv(object): 返回对象的环境。
3	gethook(optional thread): 返回线程当前的钩子设置，总共三个值：当前钩子函数、当前的钩子掩码与当前的钩子计数。
4	getinfo(optional thread,function or stack leve,optional flag): 返回保存函数信息的一个表。你可以直接指定函数，或者你也可以通过一个值指定函数，该值为函数在当前线程的函数调用栈的层次。其中，0 表示当前函数（getinfo 本身）；层次 1 表示调用 getinfo 的函数，依次类推。如果数值大于活跃函数的总量，getinfo 则返回 nil。
5	getlocal(optional thread,stack level,local index): 此函数返回在 level 层次的函数中指定索引位置处的局部变量和对应的值。如果指定的索引处不存在局部变量，则返回 nil。当 level 超出范围时，则抛出错误。
6	getmetatable(value): 返回指定对象的元表，如果不存在则返回 nil。
7	getregistry(): 返回寄存器表。寄存器表是一个预定义的用于 C 代码存储 Lua 值的表。
8	getupvalue(func function,upvalue index): 根据指定索引返回函数 func 的 upvalue 值（译注：upvalue 值与函数局部变量的区别在于，即使函数并非活跃状态也可能有 upvalue 值，而非活跃函数则不存在局部变量，所以其第一个参数不是栈的层次而是函数）。如果不存在，则返回 nil。
9	setfenv(function or thread or userdata,environment table): 将指定的对象的环境设置为 table,即改变对象的作用域。
10	sethook(optional thread,hook function,hook mask string with "c" and/or "r" and/or "l",optional instruction count): 把指定函数设置为钩子。字符串掩码和计数值表示钩子被调用的时机。这里，c 表示每次调用函数时都会执行钩子；r 表示每次从函数中返回时都调用钩子；l 表示每进入新的一行调用钩子。
11	setlocal(optional thread,stack level,local index,value): 在指定的栈深度的函数中，为 index 指定的局部变量赋予值。如果局部变量不存在，则返回 nil。若 level 超出范围则抛出错误；否则返回局部变量的名称。
12	setmetatable(value,metatable): 为指定的对象设置元表，元表可以为 nil。

13	setupvalue(function,upvalue index,value): 为指定函数中索引指定的 upvalue 变量赋值。如果 upvalue 不存在，则返回 nil。否则返回此 upvalue 的名称。
14	traceback(optional thread,optional meesage string,opitona level argument): 用 traceback 构建扩展错误消息。

上面的表中列出了 Lua 的全部调试函数，我们经常用到的调试库都会用到上面的函数，它让调试变得非常容易。虽然提供了便捷的接口，但是想要用上面的函数创建一个自己的调试器并不是件容易的事。无论怎样，我们可以看一下下面这个例子中怎么使用这些调试函数的。

```
function myfunction ()
print(debug.traceback("Stack trace"))
print(debug.getinfo(1))
print("Stack trace end")
   return 10
end
myfunction ()
print(debug.getinfo(1))
```

执行上面的程序，我们可以得到如下的栈轨迹信息：

```
Stack trace
stack traceback:
   test2.lua:2: in function 'myfunction'
   test2.lua:8: in main chunk
   [C]: ?
table: 0054C6C8
Stack trace end
```

上面的例子中，我们使用 debug.trace 函数输出了栈轨迹。 debug.getinfo 函数获得函数的当前表。

示例二

在调试过程中，我们常常需要查看或修改函数局部变量的值。因此，我们可以用 getupvalue 获得变量的值，用 setupvalue 修改变量的值。示例如下：

```
function newCounter ()
  local n = 0
  local k = 0
  return function ()
    k = n
    n = n + 1
    return n
    end
```

```
end

counter = newCounter ()
print(counter())
print(counter())

local i = 1

repeat
  name, val = debug.getupvalue(counter, i)
  if name then
    print ("index", i, name, "=", val)
    if(name == "n") then
       debug.setupvalue (counter,2,10)
    end
    i = i + 1
  end -- if
until not name

print(counter())
```

运行上面的程序，我们可以得到如下面的输出结果：

```
1
2
index   1   k   =   1
index   2   n   =   2
11
```

在这个例子中，每次调用 counter 都会更新该闭包函数。我们可以通过 getupvalue 查看其当前的局部变量值。随后，我们更新局部变量的值。在为 n 设置新值之前，其值为 2。调用 setupvalue 后，n 被设置为 10。再调用 counter 时，它就会返回值 11 而不再是 3。

调试类型

- 命令行调试
- 图形界面调试

命令行调试工具

命令行调试就是使用命令行命令和 print 语句来调试程序。已经有许多现成的 Lua 命令行调试工具，下面列出了其中的一部分：

- RemDebug：RemDebug 是一个远程的调试器，它支持 Lua 5.0 和 5.1 版本。允许远程调试 Lua 程序，设置断点以及查看程序的当前状态。同时，它还能调试 CGILua 脚本。
- clidebugger：此调试器是用纯 Lua 脚本开发的命令行调试工具，支持 Lua 5.1。除了 Lua 5.1 标准库以外，它不依赖于任何其它的 Lua 库。虽然它受到了 RemDebug 影响而产生的，但是它没有远程调试的功能。
- ctrace：跟踪 Lua API 调用的小工具。
- xdbLua：windows 平台下的 Lua 命令行调试工具。
- LuaInterface - Debuger：这个项目是 LuaInterface 的扩展，它对 Lua 调试接口进行进一步的抽象，允许通过事件和方法调用的方式调试程序。
- Rldb：使用套接字的远程 Lua 调试器，支持 Linux 和 Windows 平台。它的特性比任何其它调试器都丰富。
- ModDebug：允许远程控制另外一个 Lua　程序的执行、设置断点以及查看程序的当前状态。

图形界面调试工具

图形界面的调试工具往往和集成开发环境（IDE）打包在一起。它允许在可视环境下进行调试，比如查看变量值，栈跟踪等。通过 IDE 的图形界面，你可以设置断点单步执行程序。

下面列出了几种图形界面的调试工具。

- SciTE：Windows 系统上默认的 Lua 集成开发环境，它提供了丰富的调试功能，比如，断点、单步、跳过、查看变量等等。
- Decoda：一个允许远程调试的图形界面调试工具。
- ZeroBrane Studio：一个 Lua 的集成开发环境，它集成了远程调试器、栈视图、远程控制终端、静态分析等诸多功能。它兼容各类 Lua 引擎，例如 LuaJIT,Love2d,Moai等。支持 Windows, OSX, Linux；开源。
- akdebugger：eclipse 的 Lua 调试器和编辑器插件。
- luaedit：支持运程调试、本地调试、语法高亮、自动补完、高级断点管理（包括有条件地触发断和断点计数）、函数列表、全局和本地变量列表、面对方案的管理等。

垃圾回收机制

Lua 通过特定算法的垃圾回收机制实现自动内存管理。由于自动内存管理机制的存在，作为程序开发人员：

- 不需要关心对象的内存分配问题。
- 不再使用对象时，除了将引用它的变量设为 nil，不需要主动释放对象。

Lua 的垃圾回收器会不断运行去收集不再被 Lua 程序访问的对象。
所有的对象，包括表、userdata、函数、线程、字符串等都由自动内存管理机制管理它们空间的分配和释放。Lua 实现了一个增量式标记清除垃圾收集器。它用两个数值控制垃圾回收周期，垃圾收集器暂停时间（garbage-collector pause）和垃圾收集器步长倍增器（garbage-collector step multiplier）。其数值是以百分制计数的，即数值 100 内部表示 1。

垃圾收集器暂停时间

该数值被用于控制垃圾收集器被 Lua 自动内存管理再次运行之前需要的等待时长。当其小于 100 时意味着收集器在新周期开始前不再等待。其值越大垃圾回收器被运行的频率越低，越不主动。当其值 200 时，收集器在总使用内存数量达到上次垃圾收集时的两倍时再开启新的收集周期。因此，根据程序不同的特征，可以通过修改该值使得程序达到最佳的性能。

垃圾收集器步长倍增器

步长倍增器用于控制了垃圾收集器相对内存分配的速度。数值越大收集器工作越主动，但同时也增加了垃圾收集每次迭代步长的大小。值小于 100 可能会导致垃圾器一个周期永远不能结束，建议不要这么设置。默认值为 200，表示垃圾收集器运行的速率是内存分配的两倍。

垃圾回收器相关函数

作为开发人员，我们可能需要控制 Lua 的自动内存管理机制，可以使用下面的这些方法：

- collectgarbage("collect")：运行一个完整的垃圾回收周期。
- collectgarbage("count")：返回当前程序使用的内存总量，以 KB 为单位。
- collectgarbage("restart")：如果垃圾回收器停止，则重新运行它。

- collectgarbage("setpause")：设置垃圾收集暂停时间变量的值，值由第二个参数指出（第二参数的值除以 100 后赋予变量）。稍后，我们将详细讨论它的用法。
- collectgarbage("setsetmul")：设置垃圾收集器步长倍增器的值，第二个参数的含义与上同。
- collectgarbage("step")：进行一次垃圾回收迭代。第二个参数值越大，一次迭代的时间越长；如果本次迭代是垃圾回收的最后一次迭代则此函数返回 true。
- collectgarbage("stop")：停止垃圾收集器运行。

下面的示例代码中使用了垃圾收集器相关函数，如下所示：

```
mytable = {"apple", "orange", "banana"}

print(collectgarbage("count"))

mytable = nil

print(collectgarbage("count"))

print(collectgarbage("collect"))

print(collectgarbage("count"))
```

运行上面的程序，我们可以得到如下的输出结果。请注意，输出结果与操作系统类型与 Lua 自动内存管理都有关，所以可能实际运行的结果与下面不相同。

```
20.9560546875
20.9853515625
0
19.4111328125
```

从上面的程序，我们可以看出，一旦垃圾回收运行后，使用的内存量立即就减少了。但是，我们并不需要主动去调用它。因为，即使我们不调用此函数，Lua 也会按配置的周期自动的调用垃圾回收器。
显然，如果需要，我们可以用上面的这些函数调整垃圾回收器的行为。这些函数帮且程序开发人员处理更加复杂的场景。根据开发的不同程序的内存需求，我们可以使用到这些方法来提高程序的性能。虽然大部分情况下，我们都不会用到这些函数，但是了解这些方法可以帮助我们调试程序，以免应用上线后带来的损失。

译注：更多垃圾回收器的内容请参考官网或者此博客 (http://www.xuebuyuan.com/1636688.html) 。

面向对象

面向对象概述

面向对象编程技术是目前最常用的编程技术之一。目前大量的编程语言都支持面向对象的特性：

- C++
- Java
- Objective-C
- Smalltalk
- C#
- Ruby

面向对象的特征

-类（class）：类是可以创建对象，并为状态（成员变量）提供初值及行为实现的可扩展模板。 -对象（objects）：对象是类的实例，每个对象都有独立的内存区域。 -继承（inheritance）：继承用于描述一个类的变量和函数被另一个类继承的行为。 -封装（encapsulation）：封装是指将数据和函数组织在一个类中。外部可以通过类的方法访问内中的数据。封装也被称之为数据抽象。

Lua 中的面向对象

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。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。
Lua 中的表拥有对象的特征，比如状态和独立于其值的标识。两个有相同值的对象（表）是两个不同的对象，但是一个对象在不同的时间可以拥有不同的值。与对象一样，表拥有独立于其创建者和创建位置的生命周期。

一个真实世界的例子

面向对象已经是一个广泛使用的概念，但是你需要正确清楚地理解它。
让我们看一个数学方面的例子。我们经常需要处理各种形状，比如圆、矩形、正方形。
这些形状有一个共同的特征——面积。所以，所有其它的形状都可以从有一个公共特征——面积的基类扩展而

来。每个对象都可以有它自己的特征和函数，比如矩阵有属性长、宽和面积，printArea 和 calculateArea 方法。

创建一个简单的类

下面例子实现了矩阵类的三个属性：面积、长和宽。它还同时实现了输出面积的函数 printArea。

```
-- 元类
Rectangle = {area = 0, length = 0, breadth = 0}

-- 继承类的方法 new
function Rectangle:new (o,length,breadth)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
  self.length = length or 0
  self.breadth = breadth or 0
  self.area = length*breadth;
  return o
end

-- 继承类的方法 printArea
function Rectangle:printArea ()
  print("The area of Rectangle is ",self.area)
end
```

创建对象

创建对象即是为类的实例分配内存空间的过程。每个对象都有自己独立的内存区域，同时还会共享类的数据。

```
r = Rectangle:new(nil,10,20)
```

访问属性

我们可以使用点操作符访问类中属性。

```
print(r.length)
```

访问成员方法

使用冒号操作符可以访问对象的成员方法，如下所示：

```
r:printArea()
```

初始化阶段，调用函数为对象分配内存同时设置初值。这与其它与面向对象的语言中的构造器很相似。其实，构造器本身也就和上面的初始化代码一样，并没有什么特别之处。

完整的例子

让我们一起看一个 Lua 实现面向对象的完整例子。

```
-- 元类
Shape = {area = 0}

-- 基类方法 new
function Shape:new (o,side)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
  side = side or 0
  self.area = side*side;
  return o
end

-- 基类方法 printArea
function Shape:printArea ()
  print("The area is ",self.area)
end

-- 创建对象
myshape = Shape:new(nil,10)

myshape:printArea()
```

运行上面的程序，我们可以得到如下的输出结果：

```
The area is     100
```

Lua 中的继承

继承就是从基对象扩展的过程，正如从图形扩展至矩形、正方形等等。在现实世界中，常用来共享或扩展某些共同的属性和方法。

让我们看一个简单的类扩展的例子。我们有如下的类：

```
 -- 元类
Shape = {area = 0}
-- 基类方法 new
function Shape:new (o,side)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
  side = side or 0
  self.area = side*side;
  return o
end
-- 基类方法 printArea
function Shape:printArea ()
  print("The area is ",self.area)
end
```

我们从上面的类中扩展出正方形类，如下所示：

```
Square = Shape:new()
-- 继承类方法 new
function Square:new (o,side)
  o = o or Shape:new(o,side)
  setmetatable(o, self)
  self.__index = self
  return o
end
```

重写基类的函数

继承类可以重写基类的方法，从而根据自己的实际情况实现功能。示例代码如下所示：

```
-- 继承方法 printArea
function Square:printArea ()
  print("The area of square is ",self.area)
end
```

继承的完整示例

在元表的帮助下，我们可以使用新的 new 方法实现类的扩展（继承）。子类中保存了所有基类的成员变量和方法。

```
 -- Meta class
Shape = {area = 0}
```

```
-- Base class method new
function Shape:new (o,side)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
  side = side or 0
  self.area = side*side;
  return o
end
-- Base class method printArea
function Shape:printArea ()
  print("The area is ",self.area)
end

-- Creating an object
myshape = Shape:new(nil,10)
myshape:printArea()

Square = Shape:new()
-- Derived class method new
function Square:new (o,side)
  o = o or Shape:new(o,side)
  setmetatable(o, self)
  self.__index = self
  return o
end

-- Derived class method printArea
function Square:printArea ()
  print("The area of square is ",self.area)
end

-- Creating an object
mysquare = Square:new(nil,10)
mysquare:printArea()

Rectangle = Shape:new()
-- Derived class method new
function Rectangle:new (o,length,breadth)
  o = o or Shape:new(o)
  setmetatable(o, self)
  self.__index = self
  self.area = length * breadth
  return o
end
```

```
-- Derived class method printArea
function Rectangle:printArea ()
  print("The area of Rectangle is ",self.area)
end

-- Creating an object
myrectangle = Rectangle:new(nil,10,20)
myrectangle:printArea()
```

运行上面的程序，我们可以得到如下的输出结果：

```
The area is     100
The area of square is     100
The area of Rectangle is     200
```

上面的例子中，我们继承基类 Shape 创建了两个子类 Rectange 与 Square。在子类中可以重写基类提供的方法。在这个例子中，子类重写了 printArea 方法。

Web 编程

Lua 是一种非常灵活的语言，它经常被用在各种平台上，包括 web 应用。其中关于 Lua Web 项目最著名的就是 Kepler 项目了。Kepler 社区成立于 2004 年，一直致力于为 Lua 提供开源的 web 组件。

尽管其它开发者也已经推出了许多的 Lua web 应用框架，但是我们还是想主要介绍一下由 Kepler 社区开发的 web 开发组件。

应用与框架

- Orbit 是一个基于 WSAPI 的 MVC web 框架（译注： MVC，模型-视图-控制器）。
- WSAPI 是一套从 Lua web 应用中抽象出来的 web 服务 API，它是许多其它项目的基础。
- Xavante 是一个提供 WSAPI 的 Lua Web 服务器。
- Sputnik 是 Kepler 项目中开发的 wiki/CMS 构架，它也基于 WSAPI。
- CGILua 支持 LuaPages 和 LuaScripts 网络页面的创建，基于 WSAPI，不过已经不再提供支持。

在本教程中，我们会让你了解到在 Web 应用开发中 Lua 可以完成哪些工作。了解更多安装和使用说明，可以参阅kepler website (http://www.keplerproject.org/)

Orbit

Orbit 是一个 MVC 类型的 Lua web 框架。它完全抛弃了 CGILua 的脚本即应用的模型，在此模型中每个 Orbit 应用都可以放在一个文件中，如果你愿意，每个应用也可以被分割在多个文件到。

所有的 Orbit 应用都支持 WSAPI 协议，所以它们也就兼容 Xavante, CGI 和 Fastcgi。它还自带了一个启动器，以启动一个 Xavante 实例便于开发。

安装 Orbit 最简单的方式是使用 LuaRocks。 luarocks 用命令行的方式安装 orbit。因此，首先你需要安装luaRocks (http://luarocks.org/en/Download) 。

如果你没安装所需的依赖，下面的步骤会引导你在 Unix/Linux 环境下搭建 Orbit 环境。

安装 Apache

连接服务器，安装 Apache2。

```
$ sudo apt-get install apache2 libapache2-mod-fcgid libfcgi-dev build-essential
$ sudo a2enmod rewrite
```

```
$ sudo a2enmod fcgid
$ sudo /etc/init.d/apache2 force-reload
```

安装 LuaRocks

```
$ sudo apt-get install luarocks
```

安装 WSAPI，FCGI，Orbit，Xavante

```
$ sudo luarocks install orbit
$ sudo luarocks install wsapi-xavante
$ sudo luarocks install wsapi-fcgi
```

配置 Apache2

```
$ sudo raj /etc/apache2/sites-available/default
```

在配置文件的 的节中增如下的节。如果下面节中有 AllowOverride None， 你需将 None 改为 All，如此 .htaccess 才能覆盖本地配置。

```
<IfModule mod_fcgid.c>
    AddHandler fcgid-script .lua
    AddHandler fcgid-script .ws
    AddHandler fcgid-script .op
    FCGIWrapper "/usr/local/bin/wsapi.fcgi" .ws
    FCGIWrapper "/usr/local/bin/wsapi.fcgi" .lua
    FCGIWrapper "/usr/local/bin/op.fcgi" .op
    #FCGIServer "/usr/local/bin/wsapi.fcgi" -idle-timeout 60 -processes 1
    #IdleTimeout 60
    #ProcessLifeTime 60
</IfModule>
```

配置好后重启服务器使得配置更改生效。

为了使你的应用可以运行，你需要在你的 Orbit 应用根目录下的 .htaccess 文件中添加 +ExecCGI，在本例中根目录为 /var/www。

```
Options +ExecCGI
DirectoryIndex index.ws
```

示例——Orbit

```
 #!/usr/bin/env index.lua
-- index.lua
require"orbit"

-- 声明
 module("myorbit", package.seeall, orbit.new)

-- 处理程序
function index(web)
```

```
  return my_home_page()
end

-- 分配器
myorbit:dispatch_get(index, "/", "/index")

-- 样例页面
function my_home_page()
  return [[
   <head></head>
   <html>
   <h2>First Page</h2>
   </html>
   ]]
end
```

现在，你可以启动你的浏览器访问 http://localhost:8080 就可以看到下面的结果：

```
First Page
```

Orbit 还提供了另外选项，该选项使得 Lua　代码可以生成 html。

```
 #!/usr/bin/env index.lua
-- index.lua
require"orbit"

function generate()
   return html {
      head{title "HTML Example"},
      body{
         h2{"Here we go again!"}
      }
   }
end

orbit.htmlify(generate)

print(generate())
```

创建表单

简单的表单创建代码如下：

```
 #!/usr/bin/env index.lua
require"orbit"

function wrap (inner)
```

```
    return html{ head(), body(inner) }
end

function test ()
    return wrap(form (H'table' {
        tr{td"First name",td( input{type='text', name='first'})},
        tr{td"Second name",td(input{type='text', name='second'})},
        tr{ td(input{type='submit', value='Submit!'}),
            td(input{type='submit',value='Cancel'})
        },
    }))
end

orbit.htmlify(wrap,test)

print(test())
```

你可以在官网 (http://keplerproject.github.io/orbit/example.html) 找到关于 orbit 更加详细内容。

WSPAI

正如前面所说的，WSAPI 是大多数项目的基础，它内嵌了大量的特性。你现在可以在下面的这些系统平台上使用 WSAPI：

- Windows−
- UNNIX 类系统

WSAPI 支持的服务器和接口包括：

- CGI
- FastCGI
- Xavante

WSAPI 提供了大量库方便我们使用 Lua 进行 Web 应用的开发。下面列出了其支持的部分特征：

- 请求处理
- 输出缓存
- 认证机制
- 文件上传
- 请求隔离

- 复用

下面是 WSAPI 的一个简单例子。

```
 #!/usr/bin/env wsapi.cgi

module(..., package.seeall)
function run(wsapi_env)
  local headers = { ["Content-type"] = "text/html" }

  local function hello_text()
    coroutine.yield("<html><body>")
    coroutine.yield("<p>Hello Wsapi!</p>")
    coroutine.yield("<p>PATH_INFO: " .. wsapi_env.PATH_INFO .. "</p>")
    coroutine.yield("<p>SCRIPT_NAME: " .. wsapi_env.SCRIPT_NAME .. "</p>")
    coroutine.yield("</body></html>")
  end

  return 200, headers, coroutine.wrap(hello_text)
end
```

很容易看出来，上面的代码生成了一个简单的 html 页面。同时，你可以看到使用协程可以将 html 语句一条一条的返回给调用函数。最终返回的是 html 状态码（200）、头部以及 html 页面。

Xavante

Xavante 是一款支持 HTTP 1.1 的 Lua web 服务器。它采用模块化的结构设计，使用 URI 映射处理程序的方式进行路由。 Xavante 目前支持：

- 文件处理程序
- 重定向处理程序
- WSAPI 处理程序<

文件处理程序用于一般文件；重定向处理程序实现 URI 重映射；WSAPI 处理程序用于 WSAPI 应用。
使用示例如下：

```
require "xavante.filehandler"
require "xavante.cgiluahandler"
require "xavante.redirecthandler"

-- Define here where Xavante HTTP documents scripts are located
local webDir = XAVANTE_WEB

local simplerules = {
```

```
  { -- URI remapping example
    match = "^[^%./]*/$",
    with = xavante.redirecthandler,
    params = {"index.lp"}
  },

  { -- cgiluahandler example
    match = {"%.lp$", "%.lp/.*$", "%.lua$", "%.lua/.*$" },
    with = xavante.cgiluahandler.makeHandler (webDir)
  },

  { -- filehandler example
    match = ".",
    with = xavante.filehandler,
    params = {baseDir = webDir}
  },
}

xavante.HTTP{
  server = {host = "*", port = 8080},

  defaultHost = {
    rules = simplerules
  },
}
```

如果使用 Xavante 虚拟机，xavante.HTTP 需要被修改为如下：

```
xavante.HTTP{
  server = {host = "*", port = 8080},

  defaultHost = {},

  virtualhosts = {
    ["www.sitename.com"] = simplerules
  }
}
```

Lua web 组件

- Copas：基于协程的分配器，可用于 TCP/IP 服务器。
- Cosmo：一个安全模板引擎，保护应用免受来自模板的任何代码攻击。

- Coxpcall：封装 Lua 原生的 pcall 和 xpcall 函数，提供协程兼容的版本。
- LuaFileSystem：以可移植的方式访问底层目录结构和文件属性。
- Rings:提供在 Lua 中创建新的 Lua state 的方法。

结束语

根据我们的需求，我们可以找到很多适合我们的 Lua web 框架和组件。下面列出了另外一些可用的框架：

- Moontalk：提供高效的开发方式，为 Lua 开发的动态 web 应用提供容器，适于各种复杂程度的应用开发。
- Lapis：使用 MoonScript(或 Lua)的 web 应用框架，它可以运行在 OpenResty 服务器中。
- Lua Server Pages：一个 Lua 脚本引擎插件，此插件提供了完善的嵌入 web 开发方案。

充分利用这些 Web 框架，它可以帮助你实现更加丰富的 web 功能。

数据库访问

简单的数据操作，我们用文件就可以处理。但是，某些时候文件操作存在性能、扩展性等问题。这时候，我们就需要使用数据库。LuaSQL 是一个提供数据库操作的库，它支持多种 SQL 数据库的操作。包括：

- SQLite
- MySQL
- ODBC

在本教程中，我们会讲解用 Lua 语言对 MySQL 数据库与 SQLite 数据库进行操作。这些操作具有一般性，它们也可以移植到其它类型 SQL 数据库中。首先让我们看一下如何操作 MySQL 数据库。

MySQL 数据库环境设置

为了下面的例子可以正确演示，我们需要首先初始化数据库设置。我们假设你已经完成了如下的工作：

- 安装 MySQL 数据库，使用默认用户名 root， 默认密码为： 123456。
- 已经创建数据库 test。
- 已经阅读过关于 MySQL 的基本教程，并掌握了 MySQL 的基本知识。

导入 MySQL

假设你已经安装配置正确了，那么我们可以使用 require 语句导入 sqlite 库。安装过程中会产生一个存储数据相关文件的目录 libsql。

```
mysql = require "luasql.mysql"
```

我们可以通过 mysql 变量访问 luasql.mysql 中的 mysql 表，该表中存存储数据库操作相关的函数。

建立连接

先初始化 MySQL 的环境，再建立一个连接。如下所示：

```
local env  = mysql.mysql()
local conn = env:connect('test','root','123456')
```

上面的程序会与已存在的 MySQL 数据库 test 建立连接。

执行函数

LuaSQL 库中有一个 execute 函数，此函数可以完成所有数据加操作，包括创建、插入、更新等操作。其语法如下所示：

```
conn:execute([[ 'MySQLSTATEMENT' ]])
```

执行上面的语句这前，我们需要保证与 MySQL 数据库的连接 conn 是打开的，同时将 MySQLSTATEMENT 更改为合法的 SQL 语句。

创建表

下面的示例演示如何创建一个数据库表。例子中为表创建了两个属性分别为 id 和 name，其类型分别为整数和 v char。

```
mysql = require "luasql.mysql"

local env  = mysql.mysql()
local conn = env:connect('test','root','123456')
print(env,conn)

status,errorString = conn:execute([[CREATE TABLE sample2 (id INTEGER, name TEXT);]])
print(status,errorString )
```

运行上面的程序后，数据库中创建了一个表 sample，该表有两列，属性名分别为 id 和 name。

```
MySQL environment (004BB178)    MySQL connection (004BE3C8)
0    nil
```

如果发生错误，则函数将返回一个错误消息，成功执行则返回 nil。下面是错误消息的一个例子：

```
LuaSQL: Error executing query. MySQL: You have an error in your SQL syntax; check the manual that corresponds to yo
```

插入语句

MySQL 插入语句的示例如下所示：

```
 conn:execute([[INSERT INTO sample values('11','Raj')]])
```

更新语句

MySQL 更新语句的示例如下所示：

```
conn:execute([[UPDATE sample3 SET name='John' where id ='12']])
```

删除语句

MySQL 删除语句的示例如下所示：

```
conn:execute([[DELETE from sample3 where id ='12']])
```

查找语句

成功查找返回后，我们需要循环遍历返回的所有行以取得我们需要的数据。查找语句的示例如下：

```
cursor,errorString = conn:execute([[select * from sample]])
row = cursor:fetch ({}, "a")
while row do
  print(string.format("Id: %s, Name: %s", row.id, row.name))
  -- reusing the table of results
  row = cursor:fetch (row, "a")
end
```

上面的代码中，我们先打开了一个 MySQL 连接。通过 execute 函数返回的游标(cursor)，我们可以使用游标遍历返回的表，取得我们查找的数据。

完整示例

下面这个例子用到了所有上面提到的数据的操作函数，请看下面这个完整的例子：

```
mysql = require "luasql.mysql"

local env  = mysql.mysql()
local conn = env:connect('test','root','123456')
print(env,conn)

status,errorString = conn:execute([[CREATE TABLE sample3 (id INTEGER, name TEXT)]])
print(status,errorString )

status,errorString = conn:execute([[INSERT INTO sample3 values('12','Raj')]])
```

```
print(status,errorString )

cursor,errorString = conn:execute([[select * from sample3]])
print(cursor,errorString)

row = cursor:fetch ({}, "a")
while row do
  print(string.format("Id: %s, Name: %s", row.id, row.name))
  row = cursor:fetch (row, "a")
end
-- close everything
cursor:close()
conn:close()
```

运行上面的程序，我们可以得到如下的输出结果：

```
MySQL environment (0037B178)    MySQL connection (0037EBA8)
0    nil
1    nil
MySQL cursor (003778A8)    nil
Id: 12, Name: Raj
```

执行事务

事务是数据库中保证数据一致性的一种机制。事务有以下四个性质：

- 原子性：一个事务要么全部执行要么全部不执行。
- 一致性：事务开始前数据库是一致状态，事务结束后数据库状态也应该是一致的。
- 隔离性：多个事务并发访问时，事务之间是隔离的，一个事务的中间状态不能被其它事务可见。
- 持久性： 在事务完成以后，该事务所对数据库所做的更改便持久的保存在数据库之中，并不会被回滚。

事务以 START_TRANSACTION 开始，以 提交（commit）或 回滚（rollback）语句结束。

事务开始

为了初始化一个事务，我们需要先打开一个 MySQL 连接，再执行如下的语句：

```
conn:execute([[START TRANSACTION;]])
```

事务回滚

当需要取消事务执行时，我们需要执行如下的语句回滚至更改前的状态。

```
conn:execute([[ROLLBACK;]])
```

提交事务

开始执行事务后，我们需要使用 commit 语句提交完成的修改内容。

```
conn:execute([[COMMIT;]])
```

前面我们已经了解了 MySQL 的基本知识。接下来，我们将解释一下基本的 SQL 操作。请记住事务的概念，虽然我们在 SQLite3 中我们不在解释它，但是它的概念在 SQLite3 中同样适用。

导入 SQLite

假设你已经安装配置正确了，那么就可以使用 require 语句导入 sqlite 库。安装过程中会产生一个存储数据相关文件的目录 libsql。

```
 sqlite3 = require "luasql.sqlite3"
```

通过 sqlite3 变量可以访问提供的所有数据库操作相关函数。

建立连接

我们先初始化 sqlite 环境，然后为该环境创建一个连接。语法如下：

```
local env  = sqlite3.sqlite3()
local conn = env:connect('mydb.sqlite')
```

上面的代码会与一个 sqlite 文件建立连接，如果文件不存在则创建新的 sqlite 文件并与该新文件建立连接。

执行函数

LuaSQL 库中有一个 execute 函数，此函数可以完成所有数据加操作，包括创建、插入、更新等操作。其语法如下所示：

```
conn:execute([[ 'SQLite3STATEMENT' ]])
```

执行上面的语句这前，我们需要保证与 MySQL 数据库的连接 conn 是打开的，同时将 SQLite3STATEMENT 更改为合法的 SQL 语句。

创建表

下面的示例演示如何创建一个数据库表。例子中为表创建了两个属性分别为 id 和 name，其类型分别为整数和 v char。

```
sqlite3 = require "luasql.sqlite3"

local env  = sqlite3.sqlite3()
local conn = env:connect('mydb.sqlite')
print(env,conn)

status,errorString = conn:execute([[CREATE TABLE sample ('id' INTEGER, 'name' TEXT)]])
print(status,errorString )
```

运行上面的程序后，数据库中创建了一个表 sample，该表有两列，属性名分别为 id 和 name。

```
SQLite3 environment (003EC918)    SQLite3 connection (00421F08)
0    nil
```

如果发生错误，则函数将而一个错误消息；若成功执行则返回 nil。下面是错误消息的一个例子：

```
LuaSQL: unrecognized token: ""'id' INTEGER, 'name' TEXT)"
```

插入语句

插入语句的示例如下所示：

```
 conn:execute([[INSERT INTO sample values('11','Raj')]])
```

查找语句

查找返回后，我们需要循环遍历每行以取得我们需要的数据。查找语句的示例如下：

```
cursor,errorString = conn:execute([[select * from sample]])
row = cursor:fetch ({}, "a")
while row do
  print(string.format("Id: %s, Name: %s", row.id, row.name))
  -- reusing the table of results
```

```
  row = cursor:fetch (row, "a")
end
```

上面的代码中，我们先打开了一个 sqlite3 连接。通过 execute 函数返回的游标(cursor)，我们可以遍历返回的表，以取得我们查找的数据。

完整示例

下面这个例子用到了所有上面提到的数据的操作函数，请看下面这个完整的例子：

```
sqlite3 = require "luasql.sqlite3"

local env  = sqlite3.sqlite3()
local conn = env:connect('mydb.sqlite')
print(env,conn)

status,errorString = conn:execute([[CREATE TABLE sample ('id' INTEGER, 'name' TEXT)]])
print(status,errorString )

status,errorString = conn:execute([[INSERT INTO sample values('1','Raj')]])
print(status,errorString )

cursor,errorString = conn:execute([[select * from sample]])
print(cursor,errorString)

row = cursor:fetch ({}, "a")
while row do
  print(string.format("Id: %s, Name: %s", row.id, row.name))
  row = cursor:fetch (row, "a")
end
-- close everything
cursor:close()
conn:close()
env:close()
```

运行上面的程序，我们可以得到如下的输出结果：

```
SQLite3 environment (005EC918)    SQLite3 connection (005E77B0)
0    nil
1    nil
SQLite3 cursor (005E9200)    nil
Id: 1, Name: Raj
```

使用 libsql 库我们可以执行所有的数据库操作。所以，看完这些例子后，请自己多做一些练习。

游戏开发

Lua 语言因其结构和语法的简洁性而在各类游戏引擎中被广泛使用。游戏对图形画面要求非常苛刻，这无疑需消耗大量的内存空间，而这些内存空间的管理是非常棘手的问题。Lua 语言有自动的垃圾回收机制，这种自动化的内存管理机制也使得 Lua 受到游戏引擎开发者的青睐。著名的 Lua 游戏引擎主要包括：

- Cornoa SDK
- Gideros Mobile
- ShiVa3D
- Moai SDK
- LOVE
- CryEngine

上面每一个游戏引擎都是基于 Lua 的，并且每一个都提供了丰富的 API。我们下面看一下每一款游戏引擎的特点。

Corna SDK

这是一款支持 iPhone，iPad，Android 平台的移动设备游戏引擎。它提供了一个免费版本的 SDK, 不过该免费版本的功能也受到限制。你可以在需要的时候升级到其它版本。

Gorona SDK 提供了如下的特征：

- 物理与冲突处理接口
- Web 和网络接口
- 游戏网络接口
- 广告接口
- 数据分析接口
- 数据库和文件处理接口
- 加密和数学计算接口
- 音频和多媒体接口

相比于使用 iOS 或 Android 系统原生 API， 使用上面的接口可以让我们的开发效率更高。

Gideros Mobile

Gideros 提供 iOS 和 Android 跨平台的软件开发工具包（SDK）。它是一个免费的游戏引擎，其主要的优点包括：

- 集成开发环境：它提供一套集成开发环境，使用应用开发变得容易许多。
- 即时测试：在游戏开发过程中，通过 wifi 在 1 秒之内就可以在真实设备上测试应用。为开发者省去了导出和部署应用的时间。
- 插件：支持使用插件的方式扩展。导入的代码（C, C++, Java，Obje-C），Lua 可以直接解释执行。目前网络上已有了大量的开源插件可供使用。
- 面向对象编程：Gideros 提供了自己的类系统，支持 OOP 标准，开发可以开发干净的 OOP 代码。
- 原生速度：基于 C/C++ 和 openGL,应用可以以原生的运行，完全利用 CPU 和 GPU 的处理能力。

ShiVa3D

这一款 3D 的游戏引擎，它提供了图形化的编辑器，可以为 Web、终端、移动设备开发应用或游戏。它支持多个平台，包括：Windows，Mac，Linux，iOS，Android，BlackBerry，Palm OS，Wii，WebOS。

它主要的特点包括：

- 标准插件
- 网格修改接口
- 集成开发环境
- 内置 Terrain，Ocean 与 动画编辑器
- 支持 ODE 物理引擎
- 完全的光线映射控制
- 实时预览
- Collada 交换格式的支持

ShiVa3D 的 Web 版本是免费的，但其它的版本是收费版本。

Moai SDK

Moai SDK 是跨平台的移动游戏开发引擎，它支持 iPhone，iPad 以及 Android 系统。Moai 平台包括 Moai SDK，开源的引擎，以及 Moai 云。 Moai 云是一个 SaaS 平台，提供游戏部署的服务。不过，Moai 云平台已经关闭，现在只有游戏引擎是可用的。

LOVE

LOVE 是一个开源的 2D 游戏的开始框架，它支持 Windows，Mac OS X 以及 Linux 多个平台。

它主要提供以下的开发接口：

- 音频接口
- 文件系统接口
- 键盘和操纵杆接口
- 数据计算 API
- 窗口和鼠标接口
- 物理接口
- 系统和定时器接口

CryEngine

CryEngine 是由德国的游戏引擎开发商 Cryteck 开发的游戏引擎。到目前为止，它已由第一代引擎发展到了第四代，是一个高级的游戏开发解决方案。它目前支持 PC，Xbox 360，PlayStation3，以及 WiiU。

它主有以下的优点：

- 视觉效果就像自然光线，态柔和阴影，实时动态全局光照，光传输容量控制，颗粒底纹，镶嵌等。
- 角色动画系统与人物个性化系统。
- 参数骨骼动画和独特的专用人脸动画编辑器。
- 人工智能系统如多层导航网格战术角度系统。还提供了设计师友好的 AI 编辑系统。
- 游戏混合及分析，数据驱动的音响系统的动态声音和互动音乐等。

结束语

每个款游戏引擎都有着自己的优势以及不足之处。正确的选择游戏引擎会让你的开发变得容易和有趣得多。所以，在选择之前，请先仔细斟酌你的需求，分析哪一款游戏引擎真正的适合你，然后再决定使用它。

标准库

Lua 标准库利用 C 语言 API 实现并提供了丰富的函数，它们内置于 Lua 语言中。该标准库不仅可以提供 Lua 语言内服务，还能提供外部服务，比如文件或数据库的操作。

这些标准库使用标准的 C API 接口实现，它们作为独立的 C 语言模块提供给使用者。主要包括以下的内容：

- 基本库，包括协程子库
- 模块库
- 字符串操作
- 表操作
- 数学计算库
- 文件输入与输出
- 操作系统工具库
- 调试工具库

基本库

在本教程中，我们已经在很多地方都使用了基本库的内容。下面的表中列出了相关的函数与链接。

S.N.	库或者方法
1	错误处理库，包括错误处理函数，比如 assert，error等，详见错误处理 (./error-handling.md) 。
2	内存管理，包括与垃圾回收相关的自动内存管理的内容，详见垃圾回收 (./garbage-collection.md) 。
3	dofile([filename])，打开指定文件，并将文件内容作为代码执行。如果没有传入参数，则函数执行标准输入的内容。错误会传递至函数调用者。
4	_G，全局变量，它存储全局的环境。Lua 本身不使用此变量。
5	getfenv([f])，返回指定函数使用的当前环境（作用域），可以通过函数名或栈深度值指定函数。1 表示调用 getfenv 的函数。如果传入的参数不是函数或者 f 为 0,则返回全局环境。f 的默认值为 1。
6	getmetatable(object)：如果对象没有元表，则返回 nil。如果对象的元表有 __metable 域，则返回该值；否则返回对象的元表。
7	ipairs(t)，用于遍历表，此函数返回三个值：next 函数，表 t, 以及 0。
8	load(func[,chunkname])，使用函数 func 加载一个块（代码块），每次调用func必须返回与先前的结果连接后的字符串
9	loadfile([filename]),与 load 函数相似，此函数从文件中或标准输入（没指定文件名时）读入代码块。
10	loadstring(string,[,chunkname])，与 load 类似，从指定字符串中获得代码块。

11	next(table[,index])，此函数用于遍历表结构。第一参数为表，第二个参数是一个索引值。返回值为指定索引的下一个索引与相关的值。
12	pairs(t),用于遍历表，此函数返回三个值：next 函数，表 t, 以及 nil。
13	print(...)，打印输出传入参数。
14	rawequal(v1,v2)，判断 v1 与 v2 是否相等，不会调用任何元方法。返回布尔值。
15	rawget(table,index)，返回 table[index]，不会调用元方法。table 必须是表，索引可以是任何值。
16	rawset(table,index,value)，等价于 table[index] = value，但是不会调用元方法。函数返回表。
17	select(index,...)，如果 index 为数字n,那么 select 返回它的第 n 个可变实参，否则只能为字符串 "#",这样select会返回变长参数的总数。
18	setfenv(f,table)，设置指定函数的作用域。可以通过函数名或栈深度值指定函数。1 表示调用 setfenv 的函数。返回值为指定函数。特别地，如果 f 为 0，则改变当前线程的运行环境，这时候函数无返回值。
19	setmetatable(table,metatable)，设置指定表的元表（不能从 Lua 中改变其它类型的元表，其它类型的元表只能从 C 语言中修改）。如果 metatable 为 nil，则删除表的元表；如果原来的元表有 __metatable 域，则出错。函数返回 table。
20	tonumber(e[,base])，将参数转换为数值。如果参数本身已经是数值或者是可以转换为数值的字符串，则 tonumber 返回数值，否则返回 nil。
21	tostring(e)，将传递的实参以合理的格式转换为字符串。精确控制字符串的转换可以使用 string.format 函数。
22	type(v)，以字符串的形式返回输入参数的类型。该函数的返回值可以取字符串：nil，number，string，boolean，table，function，thread，userdata。
23	unpack(list[,i[,j]])，从指定的表中返回元素。
24	_VERSION，存储当前解释器版本信息的全局变量。该变量当前存储的内容为：Lua 5.1。（译注：与解释器版本有关）
25	协程库，包括协程相关的函数，详见协程 (./coroutines.md)

模块库

模块库提供了加载模块的基本函数。它在全局作用域内提供了 require 函数。其它的函数都是通过包管理的模块库提供的。详细内容请参见模块 (./modules.md) 。

字符串操作库

详细内容请参见字符串 (./strings.md) 。

表操作库

详细内容请参见表 (./tables.md) 。

数学函数库

详细内容请参见数学函数库 (./math-library.md) 。

文件 IO 库

详细内容请参见文件 IO (./file-io.md) 。

操作系统工具库

详细内容请参见操作系统工具库 (./operating-system-facilities.md) 。

调试工具库

详细内容请参见调试 (./debugging.md) 。

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/lua/