**Lua学习**

**1.环境搭建**

Mac下Lua环境搭建

lua源文件下载安装

到官网（<http://www.lua.org/download.html>）安装了lua包，我安装的是 lua-5.3.1

解压之后，命令行cd进入到src目录下，输入make macosx

完成后cd ..到上一层目录， 输入sudo make install

Sublime 安装与配置

下载地址：sublimetext

配置运行环境：打开sublime，顶部导航栏中找到Tools，Tools->Build System->New Build System

用下面这段代码替换打开的文件：

{

"cmd": ["/usr/local/bin/lua", "$file"],

"file\_regex": "^(?:lua:)?\t :([0-9]):?([0-9])",

"selector": "source.lua"

}

保存文件名为lua.sublime-build, 保存在默认位置就好。

新建一个文件，输入print("hello world"), 保存文件后，按快捷键 ⌘+B, 即可看到结果。

可能遇到一些状况

/usr/local/bin/lua: cannot open : No such file or directory 错误就要查看，是否文件已经保存，且后缀名是否为 .lua

终端输入 lua -v 确认是否安装成功lua

**2.语法知识**

* **Lua 基本语法**

**第一个 Lua 程序**

**交互式编程**

Lua 提供了交互式编程模式。我们可以在命令行中输入程序并立即查看效果。

Lua 交互式编程模式可以通过命令 lua -i 或 lua 来启用：

$ lua -i $ Lua 5.3.0 Copyright (C) 1994-2015 Lua.org, PUC-Rio >

**脚本式编程**

我们可以将 Lua 程序代码保持到一个以 lua 结尾的文件，并执行，该模式称为脚本式编程，如我们将如下代码存储在名为 hello.lua 的脚本文件中：

我们也可以将代码修改为如下形式来执行脚本（在开头添加：#!/usr/local/bin/lua）

**注释**

**单行注释**

两个减号是单行注释:

--

**多行注释**

--[[ 多行注释 多行注释 --]]

**标示符**

Lua 标示符用于定义一个变量，函数获取其他用户定义的项。标示符以一个字母 A 到 Z 或 a 到 z 或下划线 \_ 开头后加上0个或多个字母，下划线，数字（0到9）。

最好不要使用下划线加大写字母的标示符，因为Lua的保留字也是这样的。

Lua 不允许使用特殊字符如 @, $, 和 % 来定义标示符。 Lua 是一个区分大小写的编程语言。因此在 Lua 中 Runoob 与 runoob 是两个不同的标示符。以下列出了一些正确的标示符：

mohd zara abc move\_name a\_123 myname50 \_temp j a23b9 retVal

**关键词**

以下列出了 Lua 的保留关键字。保留关键字不能作为常量或变量或其他用户自定义标示符：

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

一般约定，以下划线开头连接一串大写字母的名字（比如 \_VERSION）被保留用于 Lua 内部全局变量。

**全局变量**

在默认情况下，变量总是认为是全局的。

全局变量不需要声明，给一个变量赋值后即创建了这个全局变量，访问一个没有初始化的全局变量也不会出错，只不过得到的结果是：nil。

如果你想删除一个全局变量，只需要将变量赋值为nil。

当且仅当一个变量不等于nil时，这个变量即存在。

* **Lua 数据类型**

Lua 是动态类型语言，变量不要类型定义,只需要为变量赋值。 值可以存储在变量中，作为参数传递或结果返回。

Lua 中有 8 个基本类型分别为：nil、boolean、number、string、userdata、function、thread 和 table。

|  |  |
| --- | --- |
| 数据类型 | 描述 |
| nil | 这个最简单，只有值nil属于该类，表示一个无效值（在条件表达式中相当于false）。 |
| boolean | 包含两个值：false和true。 |
| number | 表示双精度类型的实浮点数 |
| string | 字符串由一对双引号或单引号来表示 |
| function | 由 C 或 Lua 编写的函数 |
| userdata | 表示任意存储在变量中的C数据结构 |
| thread | 表示执行的独立线路，用于执行协同程序 |
| table | Lua 中的表（table）其实是一个"关联数组"（associative arrays），数组的索引可以是数字、字符串或表类型。在 Lua 里，table 的创建是通过"构造表达式"来完成，最简单构造表达式是{}，用来创建一个空表。 |

我们可以使用 type 函数测试给定变量或者值的类型：

**实例**

print(type("Hello world"))      *--> string*

print(type(10.4\*3))             *--> number*

print(type(print))              *--> function*

print(type(type))               *--> function*

print(type(true))               *--> boolean*

print(type(nil))                *--> nil*

print(type(type(X)))            *--> string*

**nil（空）**

nil 类型表示一种没有任何有效值，它只有一个值 -- nil，例如打印一个没有赋值的变量，便会输出一个 nil 值。

对于全局变量和 table，nil 还有一个"删除"作用，给全局变量或者 table 表里的变量赋一个 nil 值，等同于把它们删掉

nil做比较时应该加上双引号""

**type(X)==nil** 结果为 **false** 的原因是因为 **type(type(X))==string**。

**boolean（布尔）**

boolean 类型只有两个可选值：true（真） 和 false（假），Lua 把 false 和 nil 看作是"假"，其他的都为"真"

**number（数字）**

Lua 默认只有一种 number 类型 -- double（双精度）类型（默认类型可以修改 luaconf.h 里的定义）

**string（字符串）**

字符串由一对双引号或单引号来表示。也可以用 2 个方括号 "[[]]" 来表示"一块"字符串。

在对一个数字字符串上进行算术操作时，Lua 会尝试将这个数字字符串转成一个数字:print("2" + 6) = 8.0

使用 # 来计算字符串的长度，放在字符串前面

使用 # 输出的值其实是字符串所占的字节数。

当字符串为英文字符时，由于每个字符只占 1 个字节

当字符串中含有中文字符时，每个中文字符占据两个字节（windows）（mac占三个字节）

**table（表）**

在 Lua 里，table 的创建是通过"构造表达式"来完成，最简单构造表达式是{}，用来创建一个空表。也可以在表里添加一些数据，直接初始化表

Lua 中的表（table）其实是一个"关联数组"（associative arrays），数组的索引可以是数字或者是字符串。

不同于其他语言的数组把 0 作为数组的初始索引，在 Lua 里表的默认初始索引一般以 1 开始。

table 不会固定长度大小，有新数据添加时 table 长度会自动增长，没初始的 table 都是 nil。

**function（函数）**

在 Lua 中，函数是被看作是"第一类值（First-Class Value）"，函数可以存在变量里:

function 可以以匿名函数（anonymous function）的方式通过参数传递

**thread（线程）**

在 Lua 里，最主要的线程是协同程序（coroutine）。它跟线程（thread）差不多，拥有自己独立的栈、局部变量和指令指针，可以跟其他协同程序共享全局变量和其他大部分东西。

线程跟协程的区别：线程可以同时多个运行，而协程任意时刻只能运行一个，并且处于运行状态的协程只有被挂起（suspend）时才会暂停。

**userdata（自定义类型）**

userdata 是一种用户自定义数据，用于表示一种由应用程序或 C/C++ 语言库所创建的类型，可以将任意 C/C++ 的任意数据类型的数据（通常是 struct 和 指针）存储到 Lua 变量中调用。

* **Lua 变量**

Lua 变量有三种类型：全局变量、局部变量、表中的域。

Lua 中的变量全是全局变量，那怕是语句块或是函数里，除非用 local 显式声明为局部变量。

局部变量的作用域为从声明位置开始到所在语句块结束。

变量的默认值均为 nil。

当变量个数和值的个数不一致时，Lua会一直以变量个数为基础采取以下策略：

a. 变量个数 > 值的个数 按变量个数补足nil b. 变量个数 < 值的个数 多余的值会被忽略

ex:

a, b, c = 0, 1 print(a,b,c) --> 0 1 nil a, b = a+1, b+1, b+2 -- value of b+2 is ignored print(a,b) --> 1 2 a, b, c = 0 print(a,b,c) --> 0 nil nil

应该尽可能的使用局部变量，有两个好处：

* 1. 避免命名冲突。
* 2. 访问局部变量的速度比全局变量更快。
* **Lua函数**

**函数定义：**

optional\_function\_scope function function\_name( argument1, argument2, argument3..., argumentn)

function\_body

return result\_params\_comma\_separated

end

解析：

* **optional\_function\_scope: 该参数是可选的制定函数是全局函数还是局部函数，未设置该参数默认为全局函数，如果你需要设置函数为局部函数需要使用关键字 local。**
* **function\_name: 指定函数名称。**
* **argument1, argument2, argument3..., argumentn: 函数参数，多个参数以逗号隔开，函数也可以不带参数。**
* **function\_body: 函数体，函数中需要执行的代码语句块。**
* **result\_params\_comma\_separated: 函数返回值，Lua语言函数可以返回多个值，每个值以逗号隔开。**

**可变参数**

Lua 函数可以接受可变数目的参数，和 C 语言类似，在函数参数列表中使用三点 **...**表示函数有可变的参数。

**多返回值**

Lua函数可以返回多个结果值，比如string.find，其返回匹配串"开始和结束的下标"（如果不存在匹配串返回nil）。

通过 select("#",...) 来获取可变参数的数量

通常在遍历变长参数的时候只需要使用 **{…}**，然而变长参数可能会包含一些 **nil**，那么就可以用 **select** 函数来访问变长参数了：**select('#', …)**或者 **select(n, …)**

* + **select('#', …) 返回可变参数的长度**
  + **select(n, …) 返回...中的第n个参数**
* **Lua运算符**

**算术运算符**

下表列出了 Lua 语言中的常用算术运算符，设定 A 的值为10，B 的值为 20：

|  |  |  |
| --- | --- | --- |
| 操作符 | 描述 | 实例 |
| + | 加法 | A + B 输出结果 30 |
| - | 减法 | A - B 输出结果 -10 |
| \* | 乘法 | A \* B 输出结果 200 |
| / | 除法 | B / A w输出结果 2 |
| % | 取余 | B % A 输出结果 0 |
| ^ | 乘幂 | A^2 输出结果 100 |
| - | 负号 | -A 输出结果v -10 |

**关系运算符**

下表列出了 Lua 语言中的常用关系运算符，设定 A 的值为10，B 的值为 20：

|  |  |  |
| --- | --- | --- |
| 操作符 | 描述 | 实例 |
| == | 等于，检测两个值是否相等，相等返回 true，否则返回 false | (A == B) 为 false。 |
| ~= | 不等于，检测两个值是否相等，相等返回 false，否则返回 true | (A ~= B) 为 true。 |
| > | 大于，如果左边的值大于右边的值，返回 true，否则返回 false | (A > B) 为 false。 |
| < | 小于，如果左边的值大于右边的值，返回 false，否则返回 true | (A < B) 为 true。 |
| >= | 大于等于，如果左边的值大于等于右边的值，返回 true，否则返回 false | (A >= B) 返回 false。 |
| <= | 小于等于， 如果左边的值小于等于右边的值，返回 true，否则返回 false | (A <= B) 返回 true。 |

**逻辑运算符**

下表列出了 Lua 语言中的常用逻辑运算符，设定 A 的值为 true，B 的值为 false：

|  |  |  |
| --- | --- | --- |
| 操作符 | 描述 | 实例 |
| and | 逻辑与操作符。 若 A 为 false，则返回 A，否则返回 B。 | (A and B) 为 false。 |
| or | 逻辑或操作符。 若 A 为 true，则返回 A，否则返回 B。 | (A or B) 为 true。 |
| not | 逻辑非操作符。与逻辑运算结果相反，如果条件为 true，逻辑非为 false。 | not(A and B) 为 true。 |

**其他运算符**

下表列出了 Lua 语言中的连接运算符与计算表或字符串长度的运算符：

|  |  |  |
| --- | --- | --- |
| 操作符 | 描述 | 实例 |
| .. | 连接两个字符串 | a..b ，其中 a 为 "Hello " ， b 为 "World", 输出结果为 "Hello World"。 |
| # | 一元运算符，返回字符串或表的长度。 | #"Hello" 返回 5 |

**运算符优先级**

从高到低的顺序：

^ not - (unary) \* / + - .. < > <= >= ~= == and or

除了^和..外所有的二元运算符都是左连接的。

* **Lua 字符串**

字符串或串(String)是由数字、字母、下划线组成的一串字符。

Lua 语言中字符串可以使用以下三种方式来表示：

* 单引号间的一串字符。
* 双引号间的一串字符。
* [[和]]间的一串字符。

转义字符：

转义字符用于表示不能直接显示的字符，比如后退键，回车键，等。

|  |  |  |
| --- | --- | --- |
| 转义字符 | 意义 | ASCII码值（十进制） |
| \a | 响铃(BEL) | 007 |
| \b | 退格(BS) ，将当前位置移到前一列 | 008 |
| \f | 换页(FF)，将当前位置移到下页开头 | 012 |
| \n | 换行(LF) ，将当前位置移到下一行开头 | 010 |
| \r | 回车(CR) ，将当前位置移到本行开头 | 013 |
| \t | 水平制表(HT) （跳到下一个TAB位置） | 009 |
| \v | 垂直制表(VT) | 011 |
| \\ | 代表一个反斜线字符''\' | 092 |
| \' | 代表一个单引号（撇号）字符 | 039 |
| \" | 代表一个双引号字符 | 034 |
| \0 | 空字符(NULL) | 000 |
| \ddd | 1到3位八进制数所代表的任意字符 | 三位八进制 |
| \xhh | 1到2位十六进制所代表的任意字符 | 二位十六进制 |

字符串操作：

|  |  |
| --- | --- |
| 序号 | 方法&用途 |
| 1 | **string.upper(argument):**  字符串全部转为大写字母。 |
| 2 | **string.lower(argument):**  字符串全部转为小写字母。 |
| 3 | **string.gsub(mainString, findString, replaceString, num)**  在字符串中替换,mainString为要替换的字符串， findString 为被替换的字符，replaceString 要替换的字符，num 替换次数（可以忽略，则全部替换） |
| 4 | **string.find (str, substr, [init, [end]])**  在一个指定的目标字符串中搜索指定的内容(第三个参数为索引),返回其具体位置。不存在则返回 nil。 |
| 5 | **string.reverse(arg)**  字符串反转 |
| 6 | **string.format(...)**  返回一个类似printf的格式化字符串 |
| 7 | **string.char(arg) 和 string.byte(arg[,int])**  char 将整型数字转成字符并连接， byte 转换字符为整数值(可以指定某个字符，默认第一个字符)。 |
| 8 | **string.len(arg)**  计算字符串长度。 |
| 9 | **string.rep(string, n)**  返回字符串string的n个拷贝 |
| 10 | **..**  链接两个字符串 |
| 11 | **string.gmatch(str, pattern)**  回一个迭代器函数，每一次调用这个函数，返回一个在字符串 str 找到的下一个符合 pattern 描述的子串。如果参数 pattern 描述的字符串没有找到，迭代函数返回nil。 |
| 12 | **string.match(str, pattern, init)**  string.match()只寻找源字串str中的第一个配对. 参数init可选, 指定搜寻过程的起点, 默认为1。  在成功配对时, 函数将返回配对表达式中的所有捕获结果; 如果没有设置捕获标记, 则返回整个配对字符串. 当没有成功的配对时, 返回nil。 |

**字符串格式化**

Lua 提供了 **string.format()** 函数来生成具有特定格式的字符串, 函数的第一个参数是格式 , 之后是对应格式中每个代号的各种数据。

由于格式字符串的存在, 使得产生的长字符串可读性大大提高了。这个函数的格式很像 C 语言中的 printf()。

以下实例演示了如何对字符串进行格式化操作：

格式字符串可能包含以下的转义码:

* %c - 接受一个数字, 并将其转化为ASCII码表中对应的字符
* %d, %i - 接受一个数字并将其转化为有符号的整数格式
* %o - 接受一个数字并将其转化为八进制数格式
* %u - 接受一个数字并将其转化为无符号整数格式
* %x - 接受一个数字并将其转化为十六进制数格式, 使用小写字母
* %X - 接受一个数字并将其转化为十六进制数格式, 使用大写字母
* %e - 接受一个数字并将其转化为科学记数法格式, 使用小写字母e
* %E - 接受一个数字并将其转化为科学记数法格式, 使用大写字母E
* %f - 接受一个数字并将其转化为浮点数格式
* %g(%G) - 接受一个数字并将其转化为%e(%E, 对应%G)及%f中较短的一种格式
* %q - 接受一个字符串并将其转化为可安全被Lua编译器读入的格式
* %s - 接受一个字符串并按照给定的参数格式化该字符串

为进一步细化格式, 可以在%号后添加参数. 参数将以如下的顺序读入:

* (1) 符号: 一个+号表示其后的数字转义符将让正数显示正号. 默认情况下只有负数显示符号.
* (2) 占位符: 一个0, 在后面指定了字串宽度时占位用. 不填时的默认占位符是空格.
* (3) 对齐标识: 在指定了字串宽度时, 默认为右对齐, 增加-号可以改为左对齐.
* (4) 宽度数值
* (5) 小数位数/字串裁切: 在宽度数值后增加的小数部分n, 若后接f(浮点数转义符, 如%6.3f)则设定该浮点数的小数只保留n位, 若后接s(字符串转义符, 如%5.3s)则设定该字符串只显示前n位.

下面的表列出了Lua支持的所有字符类：

单个字符(除 ^$()%.[]\*+-? 外): 与该字符自身配对

* .(点): 与任何字符配对
* %a: 与任何字母配对
* %c: 与任何控制符配对(例如\n)
* %d: 与任何数字配对
* %l: 与任何小写字母配对
* %p: 与任何标点(punctuation)配对
* %s: 与空白字符配对
* %u: 与任何大写字母配对
* %w: 与任何字母/数字配对
* %x: 与任何十六进制数配对
* %z: 与任何代表0的字符配对
* %x(此处x是非字母非数字字符): 与字符x配对. 主要用来处理表达式中有功能的字符(^$()%.[]\*+-?)的配对问题, 例如%%与%配对
* [数个字符类]: 与任何[]中包含的字符类配对. 例如[%w\_]与任何字母/数字, 或下划线符号(\_)配对
* [^数个字符类]: 与任何不包含在[]中的字符类配对. 例如[^%s]与任何非空白字符配对

当上述的字符类用大写书写时, 表示与非此字符类的任何字符配对. 例如, %S表示与任何非空白字符配对.例如，'%A'非字母的字符:

> print(string.gsub("hello, up-down!", "%A", ".")) hello..up.down.    4

在模式匹配中有一些特殊字符，他们有特殊的意义，Lua中的特殊字符如下：

( ) . % + - \* ? [ ^ $

'%' 用作特殊字符的转义字符，因此 '%.' 匹配点；'%%' 匹配字符 '%'。转义字符 '%'不仅可以用来转义特殊字符，还可以用于所有的非字母的字符。

**模式条目可以是：**

* 单个字符类匹配该类别中任意单个字符；
* 单个字符类跟一个 '\*'， 将匹配零或多个该类的字符。 这个条目总是匹配尽可能长的串；
* 单个字符类跟一个 '+'， 将匹配一或更多个该类的字符。 这个条目总是匹配尽可能长的串；
* 单个字符类跟一个 '-'， 将匹配零或更多个该类的字符。 和 '\*' 不同， 这个条目总是匹配尽可能短的串；
* 单个字符类跟一个 '?'， 将匹配零或一个该类的字符。 只要有可能，它会匹配一个；
* %*n*， 这里的 *n* 可以从 1 到 9； 这个条目匹配一个等于 *n* 号捕获物（后面有描述）的子串。
* %b*xy*， 这里的 *x* 和 *y* 是两个明确的字符； 这个条目匹配以 *x* 开始 *y* 结束， 且其中 *x* 和 *y* 保持 *平衡* 的字符串。 意思是，如果从左到右读这个字符串，对每次读到一个 *x* 就 *+1* ，读到一个 *y* 就 *-1*， 最终结束处的那个 *y* 是第一个记数到 0 的 *y*。 举个例子，条目 %b() 可以匹配到括号平衡的表达式。
* %f[*set*]， 指 *边境模式*； 这个条目会匹配到一个位于 *set* 内某个字符之前的一个空串， 且这个位置的前一个字符不属于 *set* 。 集合 *set* 的含义如前面所述。 匹配出的那个空串之开始和结束点的计算就看成该处有个字符 '\0' 一样。

**模式：**

*模式* 指一个模式条目的序列。 在模式最前面加上符号 '^' 将锚定从字符串的开始处做匹配。 在模式最后面加上符号 '$' 将使匹配过程锚定到字符串的结尾。 如果 '^' 和 '$' 出现在其它位置，它们均没有特殊含义，只表示自身。

**捕获：**

模式可以在内部用小括号括起一个子模式； 这些子模式被称为 *捕获物*。 当匹配成功时，由 *捕获物* 匹配到的字符串中的子串被保存起来用于未来的用途。 捕获物以它们左括号的次序来编号。 例如，对于模式 "(a\*(.)%w(%s\*))" ， 字符串中匹配到 "a\*(.)%w(%s\*)" 的部分保存在第一个捕获物中 （因此是编号 1 ）； 由 "." 匹配到的字符是 2 号捕获物， 匹配到 "%s\*" 的那部分是 3 号。

作为一个特例，空的捕获 () 将捕获到当前字符串的位置（它是一个数字）。 例如，如果将模式 "()aa()" 作用到字符串 "flaaap" 上，将产生两个捕获物： 3 和 5 。

* **Lua 数组**

数组，就是相同数据类型的元素按一定顺序排列的集合，可以是一维数组和多维数组。

Lua 数组的索引键值可以使用整数表示，数组的大小不是固定的。

**一维数组**

一维数组是最简单的数组，其逻辑结构是线性表。一维数组可以用for循环出数组中的元素。

array = {"Lua", "Tutorial"} for i= 0, 2 do print(array[i]) end

结果：

nil Lua Tutorial

可以使用整数索引来访问数组元素，如果知道的索引没有值则返回nil。

在 Lua 索引值是以 1 为起始，但你也可以指定 0 开始。

除此外我们还可以以负数为数组索引值

**多维数组**

多维数组即数组中包含数组或一维数组的索引键对应一个数组。

* **Lua 迭代器**

迭代器（iterator）是一种对象，它能够用来遍历标准模板库容器中的部分或全部元素，每个迭代器对象代表容器中的确定的地址

在Lua中迭代器是一种支持指针类型的结构，它可以遍历集合的每一个元素。

**泛型 for 迭代器**

泛型 for 在自己内部保存迭代函数，实际上它保存三个值：迭代函数、状态常量、控制变量。

泛型 for 迭代器提供了集合的 key/value 对，语法格式如下：（k, v为变量列表；pairs(t)为表达式列表。）

for k, v in pairs(t) do print(k, v) end

* **Lua table(表)**

table 是 Lua 的一种数据结构用来帮助我们创建不同的数据类型，如：数组、字典等。

Lua table 使用关联型数组，你可以用任意类型的值来作数组的索引，但这个值不能是 nil。

Lua table 是不固定大小的，你可以根据自己需要进行扩容。

Lua也是通过table来解决模块（module）、包（package）和对象（Object）的。 例如string.format表示使用"format"来索引table string。

**table(表)的构造**

构造器是创建和初始化表的表达式。表是Lua特有的功能强大的东西。最简单的构造函数是{}，用来创建一个空表。可以直接初始化数组:

-- 初始化表 mytable = {} -- 指定值 mytable[1]= "Lua" -- 移除引用 mytable = nil -- lua 垃圾回收会释放内存

当我们为 table a 并设置元素，然后将 a 赋值给 b，则 a 与 b 都指向同一个内存。如果 a 设置为 nil ，则 b 同样能访问 table 的元素。如果没有指定的变量指向a，Lua的垃圾回收机制会清理相对应的内存。

**Table 操作**

以下列出了 Table 操作常用的方法：

|  |  |
| --- | --- |
| 序号 | 方法 & 用途 |
| 1 | **table.concat (table [, sep [, start [, end]]]):**  concat是concatenate(连锁, 连接)的缩写. table.concat()函数列出参数中指定table的数组部分从start位置到end位置的所有元素, 元素间以指定的分隔符(sep)隔开。 |
| 2 | **table.insert (table, [pos,] value):**  在table的数组部分指定位置(pos)插入值为value的一个元素. pos参数可选, 默认为数组部分末尾. |
| 3 | **table.maxn (table)**  指定table中所有正数key值中最大的key值. 如果不存在key值为正数的元素, 则返回0。(**Lua5.2之后该方法已经不存在了,本文使用了自定义函数实现**) |
| 4 | **table.remove (table [, pos])**  返回table数组部分位于pos位置的元素. 其后的元素会被前移. pos参数可选, 默认为table长度, 即从最后一个元素删起。 |
| 5 | **table.sort (table [, comp])**  对给定的table进行升序排序。 |

***注意：***

*当我们获取 table 的长度的时候无论是使用****#****还是****table.getn****其都会在索引中断的地方停止计数，而导致无法正确取得 table 的长度。*

*可以使用以下方法来代替：*

function table\_leng(t) local leng=0 for k, v in pairs(t) do leng=leng+1 end return leng; end

* **Lua 模块与包**

模块类似于一个封装库，从 Lua 5.1 开始，Lua 加入了标准的模块管理机制，可以把一些公用的代码放在一个文件里，以 API 接口的形式在其他地方调用，有利于代码的重用和降低代码耦合度。

Lua 的模块是由变量、函数等已知元素组成的 table，因此创建一个模块很简单，就是创建一个 table，然后把需要导出的常量、函数放入其中，最后返回这个 table 就行。

以下为创建自定义模块 module.lua，文件代码格式如下：

-- 文件名为 module.lua -- 定义一个名为 module 的模块 module = {} -- 定义一个常量 module.constant = "这是一个常量" -- 定义一个函数 function module.func1() io.write("这是一个公有函数！\n") end local function func2() print("这是一个私有函数！") end function module.func3() func2() end return module

**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，只需要简单地调用就可以了。例如：

require("<模块名>")

或者

require "<模块名>"

执行 require 后会返回一个由模块常量或函数组成的 table，并且还会定义一个包含该 table 的全局变量。

**加载机制**

对于自定义的模块，模块文件不是放在哪个文件目录都行，函数 require 有它自己的文件路径加载策略，它会尝试从 Lua 文件或 C 程序库中加载模块。

require 用于搜索 Lua 文件的路径是存放在全局变量 package.path 中，当 Lua 启动后，会以环境变量 LUA\_PATH 的值来初始这个环境变量。如果没有找到该环境变量，则使用一个编译时定义的默认路径来初始化。

当然，如果没有 LUA\_PATH 这个环境变量，也可以自定义设置，在当前用户根目录下打开 .profile 文件（没有则创建，打开 .bashrc 文件也可以），例如把 "~/lua/" 路径加入 LUA\_PATH 环境变量里：

#LUA\_PATH export LUA\_PATH="~/lua/?.lua;;"

文件路径以 ";" 号分隔，最后的 2 个 ";;" 表示新加的路径后面加上原来的默认路径。

接着，更新环境变量参数，使之立即生效。

source ~/.profile

这时假设 package.path 的值是：

/Users/dengjoe/lua/?.lua;./?.lua;/usr/local/share/lua/5.1/?.lua;/usr/local/share/lua/5.1/?/init.lua;/usr/local/lib/lua/5.1/?.lua;/usr/local/lib/lua/5.1/?/init.lua

那么调用 require("module") 时就会尝试打开以下文件目录去搜索目标。

/Users/dengjoe/lua/module.lua; ./module.lua /usr/local/share/lua/5.1/module.lua /usr/local/share/lua/5.1/module/init.lua /usr/local/lib/lua/5.1/module.lua /usr/local/lib/lua/5.1/module/init.lua

如果找过目标文件，则会调用 package.loadfile 来加载模块。否则，就会去找 C 程序库。

搜索的文件路径是从全局变量 package.cpath 获取，而这个变量则是通过环境变量 LUA\_CPATH 来初始。

搜索的策略跟上面的一样，只不过现在换成搜索的是 so 或 dll 类型的文件。如果找得到，那么 require 就会通过 package.loadlib 来加载它。

**C 包**

Lua在一个叫loadlib的函数内提供了所有的动态连接的功能。这个函数有两个参数:库的绝对路径和初始化函数。所以典型的调用的例子如下:

local path = "/usr/local/lua/lib/libluasocket.so" local f = loadlib(path, "luaopen\_socket")

loadlib函数加载指定的库并且连接到Lua，然而它并不打开库（也就是说没有调用初始化函数），反之他返回初始化函数作为Lua的一个函数，这样我们就可以直接在Lua中调用他。

如果加载动态库或者查找初始化函数时出错，loadlib将返回nil和错误信息。我们可以修改前面一段代码，使其检测错误然后调用初始化函数：

local path = "/usr/local/lua/lib/libluasocket.so" -- 或者 path = "C:\\windows\\luasocket.dll"，这是 Window 平台下 local f = assert(loadlib(path, "luaopen\_socket")) f() -- 真正打开库

一般情况下我们期望二进制的发布库包含一个与前面代码段相似的stub文件，安装二进制库的时候可以随便放在某个目录，只需要修改stub文件对应二进制库的实际路径即可。

将stub文件所在的目录加入到LUA\_PATH，这样设定后就可以使用require函数加载C库了。

* **Lua 元表(Metatable)**

在 Lua table 中我们可以访问对应的key来得到value值，但是却无法对两个 table 进行操作。

因此 Lua 提供了元表(Metatable)，允许我们改变table的行为，每个行为关联了对应的元方法。

例如，使用元表我们可以定义Lua如何计算两个table的相加操作a+b。

当Lua试图对两个表进行相加时，先检查两者之一是否有元表，之后检查是否有一个叫"\_\_add"的字段，若找到，则调用对应的值。"\_\_add"等即时字段，其对应的值（往往是一个函数或是table）就是"元方法"。

有两个很重要的函数来处理元表：

* **setmetatable(table,metatable): 对指定 table 设置元表(metatable)，如果元表(metatable)中存在 \_\_metatable 键值，setmetatable 会失败。**
* **getmetatable(table): 返回对象的元表(metatable)。**

以下实例演示了如何对指定的表设置元表：

mytable = {} -- 普通表 mymetatable = {} -- 元表 setmetatable(mytable,mymetatable) -- 把 mymetatable 设为 mytable 的元表

以上代码也可以直接写成一行：

mytable = setmetatable({},{})

以下为返回对象元表：

getmetatable(mytable) -- 这回返回mymetatable

**\_\_index 元方法**

这是 metatable 最常用的键。

当你通过键来访问 table 的时候，如果这个键没有值，那么Lua就会寻找该table的metatable（假定有metatable）中的\_\_index 键。如果\_\_index包含一个表格，Lua会在表格中查找相应的键。

如果\_\_index包含一个函数的话，Lua就会调用那个函数，table和键会作为参数传递给函数。

\_\_index 元方法查看表中元素是否存在，如果不存在，返回结果为 nil；如果存在则由 \_\_index 返回结果。

mytable = setmetatable({key1 = "value1"}, { \_\_index = function(mytable, key) if key == "key2" then return "metatablevalue" else return nil end end }) print(mytable.key1,mytable.key2)

实例输出结果为：

value1    metatablevalue

**总结**

*Lua 查找一个表元素时的规则，其实就是如下 3 个步骤:*

* 1.在表中查找，如果找到，返回该元素，找不到则继续
* 2.判断该表是否有元表，如果没有元表，返回 nil，有元表则继续。
* 3.判断元表有没有 \_\_index 方法，如果 \_\_index 方法为 nil，则返回 nil；如果 \_\_index 方法是一个表，则重复 1、2、3；如果 \_\_index 方法是一个函数，则返回该函数的返回值。

**\_\_newindex 元方法**

\_\_newindex 元方法用来对表更新，\_\_index则用来对表访问 。

当你给表的一个缺少的索引赋值，解释器就会查找\_\_newindex 元方法：如果存在则调用这个函数而不进行赋值操作。

\_\_add 键包含在元表中，并进行相加操作。 表中对应的操作列表如下：(**注意：\_\_**是两个下划线)

|  |  |
| --- | --- |
| 模式 | 描述 |
| \_\_add | 对应的运算符 '+'. |
| \_\_sub | 对应的运算符 '-'. |
| \_\_mul | 对应的运算符 '\*'. |
| \_\_div | 对应的运算符 '/'. |
| \_\_mod | 对应的运算符 '%'. |
| \_\_unm | 对应的运算符 '-'. |
| \_\_concat | 对应的运算符 '..'. |
| \_\_eq | 对应的运算符 '=='. |
| \_\_lt | 对应的运算符 '<'. |
| \_\_le | 对应的运算符 '<='. |

**\_\_call 元方法**

\_\_call 元方法在 Lua 调用一个值时调用

**\_\_tostring 元方法**

\_\_tostring 元方法用于修改表的输出行为

* **Lua 协同程序(coroutine)**

**什么是协同(coroutine)？**

Lua 协同程序(coroutine)与线程比较类似：拥有独立的堆栈，独立的局部变量，独立的指令指针，同时又与其它协同程序共享全局变量和其它大部分东西。

协同是非常强大的功能，但是用起来也很复杂。

**线程和协同程序区别**

线程与协同程序的主要区别在于，一个具有多个线程的程序可以同时运行几个线程，而协同程序却需要彼此协作的运行。

在任一指定时刻只有一个协同程序在运行，并且这个正在运行的协同程序只有在明确的被要求挂起的时候才会被挂起。

协同程序有点类似同步的多线程，在等待同一个线程锁的几个线程有点类似协同。

**基本语法**

|  |  |
| --- | --- |
| 方法 | 描述 |
| coroutine.create() | 创建coroutine，返回coroutine， 参数是一个函数，当和resume配合使用的时候就唤醒函数调用 |
| coroutine.resume() | 重启coroutine，和create配合使用 |
| coroutine.yield() | 挂起coroutine，将coroutine设置为挂起状态，这个和resume配合使用能有很多有用的效果 |
| coroutine.status() | 查看coroutine的状态  注：coroutine的状态有三种：dead，suspend，running，具体什么时候有这样的状态请参考下面的程序 |
| coroutine.wrap（） | 创建coroutine，返回一个函数，一旦你调用这个函数，就进入coroutine，和create功能重复 |
| coroutine.running() | 返回正在跑的coroutine，一个coroutine就是一个线程，当使用running的时候，就是返回一个corouting的线程号 |

coroutine.running就可以看出来,coroutine在底层实现就是一个线程。

当create一个coroutine的时候就是在新线程中注册了一个事件。

当使用resume触发事件的时候，create的coroutine函数就被执行了，当遇到yield的时候就代表挂起当前线程，等候再次resume触发事件。

* **Lua 文件 I/O**

Lua I/O 库用于读取和处理文件。分为简单模式（和C一样）、完全模式。

* 简单模式（simple model）拥有一个当前输入文件和一个当前输出文件，并且提供针对这些文件相关的操作。
* 完全模式（complete model） 使用外部的文件句柄来实现。它以一种面对对象的形式，将所有的文件操作定义为文件句柄的方法

简单模式在做一些简单的文件操作时较为合适。但是在进行一些高级的文件操作的时候，简单模式就显得力不从心。例如同时读取多个文件这样的操作，使用完全模式则较为合适。

打开文件操作语句如下：

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

mode 的值有：

|  |  |
| --- | --- |
| 模式 | 描述 |
| r | 以只读方式打开文件，该文件必须存在。 |
| w | 打开只写文件，若文件存在则文件长度清为0，即该文件内容会消失。若文件不存在则建立该文件。 |
| a | 以附加的方式打开只写文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留。（EOF符保留） |
| r+ | 以可读写方式打开文件，该文件必须存在。 |
| w+ | 打开可读写文件，若文件存在则文件长度清为零，即该文件内容会消失。若文件不存在则建立该文件。 |
| a+ | 与a类似，但此文件可读可写 |
| b | 二进制模式，如果文件是二进制文件，可以加上b |
| + | 号表示对文件既可以读也可以写 |

**简单模式**

简单模式使用标准的 I/O 或使用一个当前输入文件和一个当前输出文件。

在以上实例中我们使用了 io."x" 方法，其中 io.read() 中我们没有带参数，参数可以是下表中的一个：

|  |  |
| --- | --- |
| 模式 | 描述 |
| "\*n" | 读取一个数字并返回它。例：file.read("\*n") |
| "\*a" | 从当前位置读取整个文件。例：file.read("\*a") |
| "\*l"（默认） | 读取下一行，在文件尾 (EOF) 处返回 nil。例：file.read("\*l") |
| number | 返回一个指定字符个数的字符串，或在 EOF 时返回 nil。例：file.read(5) |

其他的 io 方法有：

* **io.tmpfile():返回一个临时文件句柄，该文件以更新模式打开，程序结束时自动删除**
* **io.type(file): 检测obj是否一个可用的文件句柄**
* **io.flush(): 向文件写入缓冲中的所有数据**
* **io.lines(optional file name): 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**

*-- 以只读方式打开文件*

file = io.open("test.lua", "r")

*-- 设置默认输入文件为 test.lua*

io.input(file)

*-- 输出文件第一行*

print(io.read())

*-- 关闭打开的文件*

io.close(file)

*-- 以附加的方式打开只写文件*

file = io.open("test.lua", "a")

*-- 设置默认输出文件为 test.lua*

io.output(file)

*-- 在文件最后一行添加 Lua 注释*

io.write("--  test.lua 文件末尾注释")

*-- 关闭打开的文件*

io.close(file)

**完全模式**

通常我们需要在同一时间处理多个文件。我们需要使用 file:function\_name 来代替 io.function\_name 方法。

*-- 以只读方式打开文件*

file = io.open("test.lua", "r")

*-- 输出文件第一行*

print(file:read())

*-- 关闭打开的文件*

file:close()

*-- 以附加的方式打开只写文件*

file = io.open("test.lua", "a")

*-- 在文件最后一行添加 Lua 注释*

file:write("--test")

*-- 关闭打开的文件*

file:close()

read 的参数与简单模式一致。

其他方法:

* **file:seek(optional whence, optional 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),失败则返回nil加错误信息。参数 whence 值可以是:**
  + "set": 从文件头开始
  + "cur": 从当前位置开始[默认]
  + "end": 从文件尾开始
  + offset:默认为0

不带参数file:seek()则返回当前位置,file:seek("set")则定位到文件头,file:seek("end")则定位到文件尾并返回文件大小

* **file:flush(): 向文件写入缓冲中的所有数据**
* **io.lines(optional file name): 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() <=> io.input():lines(); 读取默认输入设备的内容，但结束时不关闭文件,如：

for line in io.lines("main.lua") do 　　print(line) end

* **Lua 错误处理**

程序运行中错误处理是必要的，在我们进行文件操作，数据转移及web service 调用过程中都会出现不可预期的错误。如果不注重错误信息的处理，就会造成信息泄露，程序无法运行等情况。

任何程序语言中，都需要错误处理。错误类型有：

* 语法错误
* 运行错误

**语法错误**

语法错误通常是由于对程序的组件（如运算符、表达式）使用不当引起的。

**运行错误**

运行错误是程序可以正常执行，但是会输出报错信息。

**错误处理**

我们可以使用两个函数：assert 和 error 来处理错误。

**local** **function** add(a,b)

   assert(type(a) == "number", "a 不是一个数字")

   assert(type(b) == "number", "b 不是一个数字")

**return** a+b

**end**

add(10)

assert首先检查第一个参数，若没问题，assert不做任何事情；否则，assert以第二个参数作为错误信息抛出。

**error函数**

语法格式：

error (message [, level])

功能：终止正在执行的函数，并返回message的内容作为错误信息(error函数永远都不会返回)

通常情况下，error会附加一些错误位置的信息到message头部。

Level参数指示获得错误的位置:

* Level=1[默认]：为调用error位置(文件+行号)
* Level=2：指出哪个调用error的函数的函数
* Level=0:不添加错误位置信息

**pcall 和 xpcall、debug**

Lua中处理错误，可以使用函数pcall（protected call）来包装需要执行的代码。

pcall接收一个函数和要传递给后者的参数，并执行，执行结果：有错误、无错误；返回值true或者或false, errorinfo。

语法格式如下

if pcall(function\_name, ….) then -- 没有错误 else -- 一些错误 end

简单实例：

**实例**

> =pcall(**function**(i) print(i) **end**, 33)

33

true

> =pcall(**function**(i) print(i) error('error..') **end**, 33)

33

false        stdin:1: error..

> function f() return false,2 end > if f() then print '1' else print '0' end 0

pcall以一种"保护模式"来调用第一个参数，因此pcall可以捕获函数执行中的任何错误。

通常在错误发生时，希望落得更多的调试信息，而不只是发生错误的位置。但pcall返回时，它已经销毁了调用桟的部分内容。

Lua提供了xpcall函数，xpcall接收第二个参数——一个错误处理函数，当错误发生时，Lua会在调用桟展开（unwind）前调用错误处理函数，于是就可以在这个函数中使用debug库来获取关于错误的额外信息了。

debug库提供了两个通用的错误处理函数:

* debug.debug：提供一个Lua提示符，让用户来检查错误的原因
* debug.traceback：根据调用桟来构建一个扩展的错误消息

>=xpcall(function(i) print(i) error('error..') end, function() print(debug.traceback()) end, 33) 33 stack traceback: stdin:1: in function <stdin:1> [C]: in function 'error' stdin:1: in function <stdin:1> [C]: in function 'xpcall' stdin:1: in main chunk [C]: in ? false nil

* **Lua 调试(Debug)**

Lua 提供了 debug 库用于提供创建我们自定义调试器的功能。Lua 本身并未有内置的调试器，但很多开发者共享了他们的 Lua 调试器代码。

Lua 中 debug 库包含以下函数：

|  |  |
| --- | --- |
| 序号 | 方法 & 用途 |
| 1. | **debug():**  进入一个用户交互模式，运行用户输入的每个字符串。 使用简单的命令以及其它调试设置，用户可以检阅全局变量和局部变量， 改变变量的值，计算一些表达式，等等。  输入一行仅包含 cont 的字符串将结束这个函数， 这样调用者就可以继续向下运行。 |
| 2. | **getfenv(object):**  返回对象的环境变量。 |
| 3. | **gethook(optional thread):**  返回三个表示线程钩子设置的值： 当前钩子函数，当前钩子掩码，当前钩子计数 |
| 4. | **getinfo ([thread,] f [, what]):**  返回关于一个函数信息的表。 你可以直接提供该函数， 也可以用一个数字 f 表示该函数。 数字 f 表示运行在指定线程的调用栈对应层次上的函数： 0 层表示当前函数（getinfo 自身）； 1 层表示调用 getinfo 的函数 （除非是尾调用，这种情况不计入栈）；等等。 如果 f 是一个比活动函数数量还大的数字， getinfo 返回 nil。 |
| 5. | **debug.getlocal ([thread,] f, local):**  此函数返回在栈的 f 层处函数的索引为 local 的局部变量 的名字和值。 这个函数不仅用于访问显式定义的局部变量，也包括形参、临时变量等。 |
| 6. | **getmetatable(value):**  把给定索引指向的值的元表压入堆栈。如果索引无效，或是这个值没有元表，函数将返回 0 并且不会向栈上压任何东西。 |
| 7. | **getregistry():**  返回注册表表，这是一个预定义出来的表， 可以用来保存任何 C 代码想保存的 Lua 值。 |
| 8. | **getupvalue (f, up)**  此函数返回函数 f 的第 up 个上值的名字和值。 如果该函数没有那个上值，返回 nil 。  以 '(' （开括号）打头的变量名表示没有名字的变量 （去除了调试信息的代码块）。 |
| 10. | **sethook ([thread,] hook, mask [, count]):**  将一个函数作为钩子函数设入。 字符串 mask 以及数字 count 决定了钩子将在何时调用。 掩码是由下列字符组合成的字符串，每个字符有其含义：  **'c':**每当 Lua 调用一个函数时，调用钩子；  **'r':**每当 Lua 从一个函数内返回时，调用钩子；  **'l':**每当 Lua 进入新的一行时，调用钩子。 |
| 11. | **setlocal ([thread,] level, local, value):**  这个函数将 value 赋给 栈上第 level 层函数的第 local 个局部变量。 如果没有那个变量，函数返回 nil 。 如果 level 越界，抛出一个错误。 |
| 12. | **setmetatable (value, table):**  将 value 的元表设为 table （可以是 nil）。 返回 value。 |
| 13. | **setupvalue (f, up, value):**  这个函数将 value 设为函数 f 的第 up 个上值。 如果函数没有那个上值，返回 nil 否则，返回该上值的名字。 |
| 14. | **traceback ([thread,] [message [, level]]):**  如果 message 有，且不是字符串或 nil， 函数不做任何处理直接返回 message。 否则，它返回调用栈的栈回溯信息。 字符串可选项 message 被添加在栈回溯信息的开头。 数字可选项 level 指明从栈的哪一层开始回溯 （默认为 1 ，即调用 traceback 的那里）。 |

**调试类型**

* 命令行调试
* 图形界面调试

命令行调试器有：RemDebug、clidebugger、ctrace、xdbLua、LuaInterface - Debugger、Rldb、ModDebug。

图形界调试器有：SciTE、Decoda、ZeroBrane Studio、akdebugger、luaedit。

* **Lua 垃圾回收**

Lua 采用了自动内存管理。 这意味着你不用操心新创建的对象需要的内存如何分配出来， 也不用考虑在对象不再被使用后怎样释放它们所占用的内存。

Lua 运行了一个**垃圾收集器**来收集所有**死对象** （即在 Lua 中不可能再访问到的对象）来完成自动内存管理的工作。 Lua 中所有用到的内存，如：字符串、表、用户数据、函数、线程、 内部结构等，都服从自动管理。

Lua 实现了一个增量标记-扫描收集器。 它使用这两个数字来控制垃圾收集循环： 垃圾收集器间歇率和垃圾收集器步进倍率。 这两个数字都使用百分数为单位 （例如：值 100 在内部表示 1 ）。

垃圾收集器间歇率控制着收集器需要在开启新的循环前要等待多久。 增大这个值会减少收集器的积极性。 当这个值比 100 小的时候，收集器在开启新的循环前不会有等待。 设置这个值为 200 就会让收集器等到总内存使用量达到 之前的两倍时才开始新的循环。

垃圾收集器步进倍率控制着收集器运作速度相对于内存分配速度的倍率。 增大这个值不仅会让收集器更加积极，还会增加每个增量步骤的长度。 不要把这个值设得小于 100 ， 那样的话收集器就工作的太慢了以至于永远都干不完一个循环。 默认值是 200 ，这表示收集器以内存分配的"两倍"速工作。

如果你把步进倍率设为一个非常大的数字 （比你的程序可能用到的字节数还大 10% ）， 收集器的行为就像一个 stop-the-world 收集器。 接着你若把间歇率设为 200 ， 收集器的行为就和过去的 Lua 版本一样了： 每次 Lua 使用的内存翻倍时，就做一次完整的收集。

**垃圾回收器函数**

Lua 提供了以下函数**collectgarbage ([opt [, arg]])**用来控制自动内存管理:

* **collectgarbage("collect"): 做一次完整的垃圾收集循环。通过参数 opt 它提供了一组不同的功能：**
* **collectgarbage("count"): 以 K 字节数为单位返回 Lua 使用的总内存数。 这个值有小数部分，所以只需要乘上 1024 就能得到 Lua 使用的准确字节数（除非溢出）。**
* **collectgarbage("restart"): 重启垃圾收集器的自动运行。**
* **collectgarbage("setpause"): 将 arg 设为收集器的 间歇率 （参见 §2.5）。 返回 间歇率 的前一个值。**
* **collectgarbage("setstepmul"): 返回 步进倍率 的前一个值。**
* **collectgarbage("step"): 单步运行垃圾收集器。 步长"大小"由 arg 控制。 传入 0 时，收集器步进（不可分割的）一步。 传入非 0 值， 收集器收集相当于 Lua 分配这些多（K 字节）内存的工作。 如果收集器结束一个循环将返回 true 。**
* **collectgarbage("stop"): 停止垃圾收集器的运行。 在调用重启前，收集器只会因显式的调用运行。**

以下演示了一个简单的垃圾回收实例:

**实例**

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

print(collectgarbage("count"))

mytable = nil

print(collectgarbage("count"))

print(collectgarbage("collect"))

print(collectgarbage("count"))

* **Lua 面向对象**

面向对象编程（Object Oriented Programming，OOP）是一种非常流行的计算机编程架构。

以下几种编程语言都支持面向对象编程：

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

**面向对象特征**

* 1） 封装：指能够把一个实体的信息、功能、响应都装入一个单独的对象中的特性。
* 2） 继承：继承的方法允许在不改动原程序的基础上对其进行扩充，这样使得原功能得以保存，而新功能也得以扩展。这有利于减少重复编码，提高软件的开发效率。
* 3） 多态：同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。在运行时，可以通过指向基类的指针，来调用实现派生类中的方法。
* 4）抽象：抽象(Abstraction)是简化复杂的现实问题的途径，它可以为具体问题找到最恰当的类定义，并且可以在最恰当的继承级别解释问题。

**Lua 中面向对象**

我们知道，对象由属性和方法组成。LUA中最基本的结构是table，所以需要用table来描述对象的属性。

lua 中的 function 可以用来表示方法。那么LUA中的类可以通过 table + function 模拟出来。

至于继承，可以通过 metetable 模拟出来（不推荐用，只模拟最基本的对象大部分时间够用了）。

Lua 中的表不仅在某种意义上是一种对象。像对象一样，表也有状态（成员变量）；也有与对象的值独立的本性，特别是拥有两个不同值的对象（table）代表两个不同的对象；一个对象在不同的时候也可以有不同的值，但他始终是一个对象；与对象类似，表的生命周期与其由什么创建、在哪创建没有关系。对象有他们的成员函数，表也有

**创建对象**

创建对象是为类的实例分配内存的过程。每个类都有属于自己的内存并共享公共数据。

**访问属性**

我们可以使用点号(.)来访问类的属性：

**访问成员函数**

我们可以使用冒号 **:** 来访问类的成员函数：

内存在对象初始化时分配。

**Lua 继承**

继承是指一个对象直接使用另一对象的属性和方法。可用于扩展基础类的属性和方法。

**函数重写**

Lua 中我们可以重写基础类的函数，在派生类中定义自己的实现方式

* **Lua 数据库访问**

本文主要为大家介绍 Lua 数据库的操作库：[LuaSQL](http://luaforge.net/projects/luasql/)。他是开源的，支持的数据库有：ODBC, ADO, Oracle, MySQL, SQLite 和 PostgreSQL。

本文为大家介绍MySQL的数据库连接。

LuaSQL 可以使用 [LuaRocks](https://luarocks.org/) 来安装可以根据需要安装你需要的数据库驱动。

LuaRocks 安装方法：

$ **wget** http:**//**luarocks.org**/**releases**/**luarocks-2.2.1.tar.gz

$ **tar** zxpf luarocks-2.2.1.tar.gz

$ **cd** luarocks-2.2.1

$ .**/**configure; **sudo** **make** bootstrap

$ **sudo** luarocks **install** luasocket

$ lua

Lua 5.3.0 Copyright **(**C**)** 1994-2015 Lua.org, PUC-Rio

**>** require "socket"

Window 下安装 LuaRocks：<https://github.com/keplerproject/luarocks/wiki/Installation-instructions-for-Windows>

安装不同数据库驱动：

luarocks **install** luasql-sqlite3

luarocks **install** luasql-postgres

luarocks **install** luasql-mysql

luarocks **install** luasql-sqlite

luarocks **install** luasql-odbc

你也可以使用源码安装方式，Lua Github 源码地址：<https://github.com/keplerproject/luasql>

Lua 连接MySql 数据库：

**实例**

require "luasql.mysql"

*--创建环境对象*

env = luasql.mysql()

*--连接数据库*

conn = env:connect("数据库名","用户名","密码","IP地址",端口)

*--设置数据库的编码格式*

conn:execute"SET NAMES UTF8"

*--执行数据库操作*

cur = conn:execute("select \* from role")

row = cur:fetch({},"a")

*--文件对象的创建*

file = io.open("role.txt","w+");

**while** row **do**

    var = string.format("%d %s**\n**", row.id, row.name)

    print(var)

    file:write(var)

    row = cur:fetch(row,"a")

**end**

file:close()  *--关闭文件对象*

conn:close()  *--关闭数据库连接*

env:close()   *--关闭数据库环境*