



Lua 5.3

最流行的绑定技术、热更，极速脚本，尽在 Lua。

Auth : 王桂林

Mail : guilin_wang@163.com

Org : 能众软件

Web : <http://edu.nzhsoft.cn>

版本信息：

版本	修订人	审阅人	时间	组织
V1.0	王桂林		2017.10.10	能众软件科技有限公司
V2.0	王桂林		2018.10.01	能众软件科技有限公司

更多学习：



1. Lua 综述.....	1
1. 1. lua 简介.....	1
1. 1. 1. 诞生.....	1
1. 1. 2. 特点.....	1
1. 2. 环境搭建.....	2
1. 2. 1. 下载.....	2
1. 2. 2. linux 工程环境.....	2
1. 2. 3. Win 编译 Lua.....	3
1. 2. 4. Win 集成 lua 环境.....	5
1. 3. 推荐书目.....	8
1. 3. 1. 书目.....	8
1. 3. 2. 文档.....	8
1. 4. 前提.....	8
1. 5. script why.....	9
1. 5. 1. 编译型 vs 解释型.....	9
1. 5. 2. 实战解说.....	9
2. 数据类型与变量.....	10
2. 1. 数据类型.....	10
2. 1. 1. 空 nil.....	10
2. 1. 2. 布尔 boolean.....	10
2. 1. 3. 数值类型 number.....	11
2. 1. 4. 字符串 string.....	11
2. 1. 5. 函数 function.....	12
2. 1. 6. 表 table.....	13
2. 1. 7. 线程 thread.....	14
2. 1. 8. 自定义类型 userdata.....	14
2. 1. 9. 分类.....	15
2. 2. 变量.....	15
2. 2. 1. 命名.....	15
2. 2. 2. 语义.....	15
2. 2. 3. 作用域.....	16
3. 运算符与表达式.....	18
3. 1. 赋值.....	18
3. 1. 1. 运算符.....	18
3. 1. 2. 示例.....	18
3. 1. 3. Tips.....	18
3. 2. 算术.....	18
3. 2. 1. 运算符.....	18
3. 2. 2. 示例.....	18
3. 2. 3. Tips.....	19
3. 3. 关系.....	19
3. 3. 1. 运算符.....	19
3. 3. 2. 示例.....	19

3.3.3. Tips.....	19
3.4. 逻辑.....	20
3.4.1. 运算符.....	20
3.4.2. 示例.....	21
3.4.3. Tips.....	21
3.4.4. ?: 三目运算符.....	21
3.5. 优先级.....	21
4. 控制结构.....	23
4.1. do end.....	23
4.2. 选择 if else.....	23
4.2.1. 单个分支型.....	23
4.2.2. 两个分支 if-else 型.....	24
4.2.3. 多个分支 if-elseif-else 型.....	24
4.3. 循环.....	24
4.3.1. while 循环.....	24
4.3.2. repeat 循环.....	25
4.3.3. for 控制结构.....	26
4.3.4. Tips.....	27
4.4. 跳转.....	27
4.4.1. break.....	27
4.4.2. return.....	28
4.4.3. goto.....	29
5. string 字符串.....	32
5.1. string.....	32
5.1.1. 定义.....	32
5.1.2. 解析.....	32
5.1.3. 操作.....	32
5.1.4. ASCII.....	33
5.1.5. number 与 string.....	33
5.2. string 库.....	34
5.2.1. api 列表.....	34
5.2.2. 测试.....	34
5.3. string.gmatch.....	36
6. Array Table.....	38
6.1. 综述.....	38
6.1.1. 概念.....	38
6.1.2. 下标从 1 开始.....	38
6.2. 一维数组.....	38
6.2.1. 默认下标.....	38
6.2.2. 给定下标(强烈不推荐).....	39
6.3. 二维数组.....	39
6.3.1. 数组的数组.....	39
6.3.2. 一维数组通过控制索引.....	40

6.4. table->array 库.....	41
6.4.1. 常用 Api 列表.....	41
6.4.2. 测试.....	41
6.5. 数组中无哈希(序列).....	45
6.5.1. 序列.....	45
6.5.2. 序列中的 nil 无效值.....	45
7. Map Table.....	48
7.1. 定义.....	48
7.1.1. 引入 i-v ->array.....	48
7.1.2. k-v->hash.....	48
7.1.3. 深入 k.....	48
7.1.4. 遍历.....	49
7.2. 引用.....	50
7.2.1. 表有名字吗? anonymous.....	50
7.2.2. 表可以赋值, 比较吗?	51
7.3. # table.....	52
7.4. 哈希中无数组.....	53
7.4.1. 纯 map.....	53
7.4.2. 混合 table.....	53
8. Function 函数.....	54
8.1. 概述.....	54
8.1.1. 函数的意义.....	54
8.1.2. 定义与示例.....	54
8.2. 函数参数.....	55
8.2.1. 命名.....	55
8.2.2. 实参与形参.....	56
8.2.3. 省()调用.....	57
8.3. 函数返回.....	57
8.3.1. 多参返回.....	57
8.3.2. return 数量.....	58
8.4. 函数是一等公民.....	59
8.4.1. first class(非语法糖).....	59
8.4.2. 普通变量.....	60
8.4.3. 表字段变量.....	60
9. 函数进阶.....	62
9.1. 变参.....	62
9.1.1. 参数 (...).....	62
9.1.2. 变参->表化{ ... }.....	62
9.1.3. 变参->table.pack(...).....	63
9.1.4. 变参之 select(...).....	63
9.1.5. 形参之固定参数.....	64
9.1.6. table.unpack({}).....	64
9.1.7. 命令行参数(...).....	65

9.2. 函数作入参--回调.....	65
9.2.1. table.sort.....	65
9.2.2. sort array.....	65
9.2.3. sort (hash table) array.....	65
9.3. 函数作返回-Closer 闭包.....	66
9.3.1. 什么是闭包.....	66
9.3.2. 上值 upvalue.....	66
9.3.3. 应用.....	68
9.4. ipairs 初试.....	70
9.4.1. 自实现.....	70
9.4.2. 虚变量.....	70
9.5. 递归.....	71
9.5.1. 求阶乘.....	71
9.5.2. 解析.....	71
9.5.3. 语法糖的展开.....	72
9.5.4. 递归尾调用.....	72
9.6. 练习.....	73
10. Iterator 迭代器.....	74
10.1. 概念.....	74
10.1.1. 引入.....	74
10.1.2. 分类.....	74
10.1.3. Generic for.....	74
10.2. 无状态的迭代器.....	75
10.2.1. 引入 square, 3, 0.....	75
10.2.2. 引入 square, 3, 0 -> for squareS(3).....	76
10.2.3. 无状态 ipairs.....	76
10.3. 有状态迭代器.....	77
10.3.1. 闭包引入.....	77
10.3.2. 有状态 ipairs.....	78
10.4. 有状态 vs 无状态.....	78
10.5. 练习.....	79
10.5.1. 设计链表，并设计其迭代函数.....	79
10.5.2. 迭代.....	80
11. 模式匹配 Pattern.....	82
11.1. 匹配函数.....	82
11.1.1. string.find.....	82
11.1.2. string.match.....	83
11.1.3. string.gsub(stitute).....	83
11.1.4. string.gmatch.....	84
11.2. 模式 pattern.....	85
11.2.1. 任一字符.....	85
11.3. 捕获.....	87
11.3.1. 匹配分割.....	87

11.3.2. 匹配复用.....	88
11.3.3. gsub 中替换部分.....	88
11.4. 正则 regex.....	89
11.4.1. 规则.....	89
11.4.2. 常见匹配.....	91
11.4.3. 示例.....	91
12. Meta 元表与元方法.....	93
12.1. 元表与元方法.....	93
12.2. 元表.....	93
12.2.1. 概述.....	93
12.2.2. 示例.....	93
12.3. 算术/关系元方法.....	93
12.3.1. 算术类元方法.....	93
12.3.2. 关系类元方法.....	94
12.4. 表相关的元方法.....	95
12.4.1. __index 读元方法.....	95
12.4.2. __newindex 写元方法.....	97
12.4.3. raw method.....	100
12.4.4. 应用.....	100
12.5. 其它元方法.....	100
12.5.1. __call 元方法.....	100
12.5.2. __tostring 元方法.....	101
12.5.3. Other.....	102
13. OO 面向对象.....	103
13.1. 类与对象.....	103
13.1.1. 表对象.....	103
13.1.2. self 与 this.....	104
13.1.3. metatable 自索引.....	104
13.1.4. 对象.....	106
13.2. 继承.....	108
13.2.1. 父类.....	108
13.2.2. 继承.....	109
13.2.3. 成员私有化.....	110
13.3. cocos2dx-lua 之 class 函数.....	111
13.3.1. 环境搭建.....	111
13.3.2. 继承函数.....	111
13.3.3. 实例分析.....	112
13.4. 面向对象纵深要点.....	113
13.4.1. 函数 self.....	113
13.4.2. 对象 o = o or {}.....	113
13.4.3. 机制 meta.....	113
13.5. 更简单的面向对象.....	113
13.5.1. 复制表实现.....	113

13. 5. 2. 闭包实现.....	115
14. Env 环境.....	117
14. 1. _G.....	117
14. 1. 1. 打印_G.....	117
14. 1. 2. 使用_G.....	118
14. 2. 沙盒 sandbox.....	118
14. 2. 1. 改变环境(lua5.1).....	118
14. 2. 2. 沙盒(lua5.1).....	119
14. 2. 3. __ENV 改写.....	119
14. 3. __ENV 变量(lua5.3).....	119
14. 3. 1. __ENV.....	119
14. 3. 2. __ENV 改写 5.1 的案例.....	120
15. Require 模块.....	122
15. 1. 引入.....	122
15. 1. 1. 定义包.....	122
15. 1. 2. 引用包(调试).....	122
15. 1. 3. 注意事项.....	122
15. 1. 4. 加载次数.....	123
15. 2. require.....	123
15. 2. 1. 定义/原理.....	123
15. 2. 2. 加载流程.....	123
15. 2. 3. require 搜索路径.....	124
15. 3. 实现原理.....	125
15. 3. 1. 测试.....	125
15. 3. 2. 代码.....	125
16. Coroutine 协程.....	126
16. 1. 概述.....	126
16. 1. 1. 官方协程.....	126
16. 1. 2. 协程注解.....	126
16. 2. 接口 Api.....	126
16. 2. 1. 协程状态.....	127
16. 2. 2. 传参示例.....	128
16. 2. 3. 练习(双 while 交互).....	131
16. 3. 应用.....	131
16. 3. 1. 管道-生产者与消费者.....	131
16. 3. 2. 非抢占式线程.....	132
16. 4. lua 包 package 的安装.....	136
16. 4. 1. luarocks.....	136
16. 4. 2. 下载及安装.....	136
17. 标准库.....	138
17. 1. 数学库.....	138
17. 1. 1. 列表.....	138
17. 1. 2. 测试.....	139

17.2. string 库.....	140
17.3. os 库.....	140
17.3.1. 列表.....	140
17.3.2. 测试.....	141
17.4. io 库.....	141
17.4.1. 简介.....	141
17.4.2. 简单模式.....	141
17.4.3. 完全模式.....	144
17.5. debug 库.....	144
17.5.1. 列表.....	144
17.5.2. 测试.....	145
18. 垃圾回收.....	146
18.1. 理论基础.....	146
18.2. 接口介绍.....	146
18.3. 实战操作.....	147
19. Lua 与 C/C++交互栈.....	148
19.1. 交互机制.....	148
19.1.1. 交互原理.....	148
19.1.2. 交互栈.....	148
19.2. API.....	148
19.2.1. push functions (C -> stack).....	148
19.2.2. set functions (stack -> Lua).....	149
19.2.3. get functions (Lua -> stack).....	149
19.2.4. access functions (stack -> C)不出栈.....	149
19.2.5. 栈管理.....	150
19.3. 测试.....	151
19.3.1. 打印栈.....	151
19.3.2. 测试.....	152
19.3.3. test.....	153
19.4. Lua API 及译文.....	153
19.4.1. 官方原文：.....	153
19.4.2. 云风译文：.....	153
20. Lua 扩展 C 程序(C/C++>lua).....	154
20.1. 宿主 /栈 /lua.....	154
20.2. 访问 lua 全局变量.....	154
20.2.1. api.....	154
20.2.2. 测试.....	154
20.3. 访问 lua 全局表字段.....	155
20.3.1. api.....	155
20.3.2. 测试.....	155
20.3.3. cheatsheet.....	156
20.4. 访问 lua 全局函数.....	157
20.4.1. Api.....	157

20. 4. 2. 调用 lua 函数-无参无返回.....	158
20. 4. 3. 调用 lua 函数-有参无返回.....	159
20. 4. 4. 调用 lua 函数-有参有返回.....	159
20. 4. 5. 调用 lua 表字段函数.....	160
20. 4. 6. 自制 lua 编译器.....	161
20. 5. 编译 lua5.3 静态库.....	165
20. 5. 1. VS 静态库工程.....	165
20. 5. 2. Qt 编译静态工程.....	172
21. C 提升 Lua 效率(lua->C/C++).....	177
21. 1. 引入.....	177
21. 1. 1. 从 lua 到 c.....	177
21. 1. 2. ipairs 源码.....	177
21. 1. 3. C 即 Lua, Lua 即 C.....	178
21. 2. 独立栈.....	178
21. 3. Lua 访问 c 入栈值.....	178
21. 3. 1. API.....	178
21. 3. 2. lua 访问 c 中入栈变量.....	179
21. 3. 3. lua 访问 c 中入栈的表.....	180
21. 4. lua 调用 C 函数.....	181
21. 4. 1. API.....	181
21. 4. 2. 调用 C++函数-无参无返回.....	182
21. 4. 3. 调用 C++函数-有参无返回.....	183
21. 4. 4. 调用 C++函数-有参有返回.....	183
21. 4. 5. 调用 C++函数返回 table.....	184
21. 5. C++函数批量注册/表化.....	185
21. 5. 1. API.....	185
21. 5. 2. 循环批量绑定全局函数.....	186
21. 5. 3. 批量表化绑定全局函数.....	188
21. 6. C++ ->dll / so.....	190
21. 6. 1. 生成 so 流程.....	190
21. 6. 2. Skynet 典型模块.....	191
21. 7. C 函数的其它操作.....	192
21. 7. 1. Array Manipulation.....	192
21. 7. 2. String Manipulation.....	192
21. 7. 3. Upvalues.....	192
22. Full Userdata.....	193
22. 1. Userdata.....	193
22. 1. 1. 定义.....	193
22. 1. 2. 接口.....	193
22. 2. 自实现 array.....	193
22. 2. 1. code Lua.....	193
22. 2. 2. code C.....	194
22. 2. 3. Code C->dll/so.....	196

22. 3. 差异化 userdata.....	196
22. 3. 1. 无差异.....	196
22. 3. 2. metatable.....	196
22. 4. Object Oriented.....	199
22. 4. 1. main.c.....	199
22. 4. 2. xx.lua.....	199
22. 5. Class userdata.....	200
23. Light UserData.....	201
23. 1. full vs light.....	201
24. Lua 的状态与线程.....	202
25. LuaBridge.....	203
26. 消消乐 Skynet 服务.....	204
27. 综合练习.....	205
27. 1. 写出运行结果.....	205
27. 1. 1. 表长度.....	205
27. 1. 2. 表引用.....	205
27. 1. 3. 面向对象.....	205
27. 1. 4. 继承.....	206
27. 2. 写程序.....	207
27. 2. 1. 对表进行排序并输出.....	207
27. 2. 2. 请写一个带有不定参数的 lua 函数 并输出所有的参数.....	207

1. Lua 综述

1. 1. lua 简介

1. 1. 1. 诞生

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



Lua 从一开始就是作为一门方便嵌入(其它应用程序)并可扩展的轻量级脚本语言来设计的，因此她一直遵从着简单、小巧、可移植、快速的原则，官方实现完全采用 ANSI C 编写，能以 C 程序库的形式嵌入到宿主程序中。LuaJIT 2 和标准 Lua 5.1 解释器采用的是著名的 MIT 许可协议。

正由于上述特点，所以 Lua 在游戏开发、机器人控制、分布式应用、图像处理、生物信息学等各种各样的领域中得到了越来越广泛的应用。其中尤以游戏开发为最，许多著名的游戏，比如 Escape from Monkey Island、World of Warcraft、大话西游，都采用了 Lua 来配合引擎完成数据描述、配置管理和逻辑控制等任务。即使像 Redis 这样中性的内存键值数据库也提供了内嵌用户 Lua 脚本的官方支持。



1. 1. 2. 特点

1. 变量名没有类型，值才有类型，变量名在运行时可与任何类型的值绑定；
2. 语言只提供唯一一种数据结构，称为表(table)，它混合了数组、哈希，可以用任何类型的；
值作为 key 和 value。提供了一致且富有表达力的表构造语法，使得 Lua 很适合描述复杂的数据；
3. 函数是一等类型，支持匿名函数和正则尾递归(proper tail recursion)；

4. 支持词法定界(lexical scoping)和闭包(closure);
5. 提供 `thread` 类型和结构化的协程(coroutine)机制，在此基础上可方便实现协作式多任务；
6. 运行期能编译字符串形式的程序文本并载入虚拟机执行；
7. 通过元表 (metatable) 和元方法 (metamethod) 提供动态元机制 (dynamic metamechanism)，从而允许程序运行时根据需要改变或扩充语法设施的内定语义；
8. 能方便地利用表和动态元机制实现基于原型(prototype-based)的面向对象模型；
9. 从 5.1 版开始提供了完善的模块机制，从而更好地支持开发大型的应用程序；

1.2. 环境搭建

1.2.1. 下载

下载地址：Lua 官网链接：<http://www.lua.org>

1.2.2. linux 工程环境

1.2.2.1. 编译安装

```
wget https://www.lua.org/ftp/lua-5.3.5.tar.gz
tar zxvf lua-5.3.5.tar.gz
cd lua-5.3.5
make linux
make install

--开发头文件
root@nzhsoft:~# cd /usr/local/include/
root@nzhsoft:/usr/local/include# ls
google lauxlib.h luaconf.h lua.h lua.hpp lualib.h
--开发库文件
root@nzhsoft:~# cd /usr/local/lib/
root@nzhsoft:/usr/local/lib# ls *lua.a
liblua.a
```

1.2.2.2. apt-get

```
sudo apt-get install lua5.3          --可执行性文件
lua5.3
sudo apt-get install liblua5.3-dev    --开发包

--开发头文件
root@nzhsoft:~# cd /usr/include/lua5.3/
root@nzhsoft:/usr/include/lua5.3# pwd
/usr/include/lua5.3
--开发库文件
root@nzhsoft:~# whereis liblua5.3.a
liblua5.3: /usr/lib/x86_64-linux-gnu/liblua5.3.so
/usr/lib/x86_64-linux-gnu/liblua5.3.a
```

1. 2. 2. 3. 运行 **lua**

```
--输入命令
root@nzhsoft:~# lua
Lua 5.3.5 Copyright (C) 1994-2018 Lua.org, PUC-Rio
> print("hello lua")
hello lua
```

```
--运行文件
root@nzhsoft:~# lua tst.lua
hello lua
hello nzhsoft
```

lua 是解释器，而 luac 是编译器

```
--先编译后执行 lua
root@nzhsoft:~# luac tst.lua
root@nzhsoft:~# ls
LuaBridge luac.out tst.lua
root@nzhsoft:~# lua luac.out
hello lua
hello nzhsoft
```

1. 2. 3. Win 编译 Lua

1. 2. 3. 1. Qt 编译

新建 Qt 的 c 项目，解压缩 `lua5.3.4.tar.gz`，然后将解压后的 `src` 文中夹中除了 `lua.c` 和 `luac.c` 以外的所有 `.h` 和 `.c` 文件加入到工程中来，编译即可。

```

1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello World!\n");
6     return 0;
7 }
8

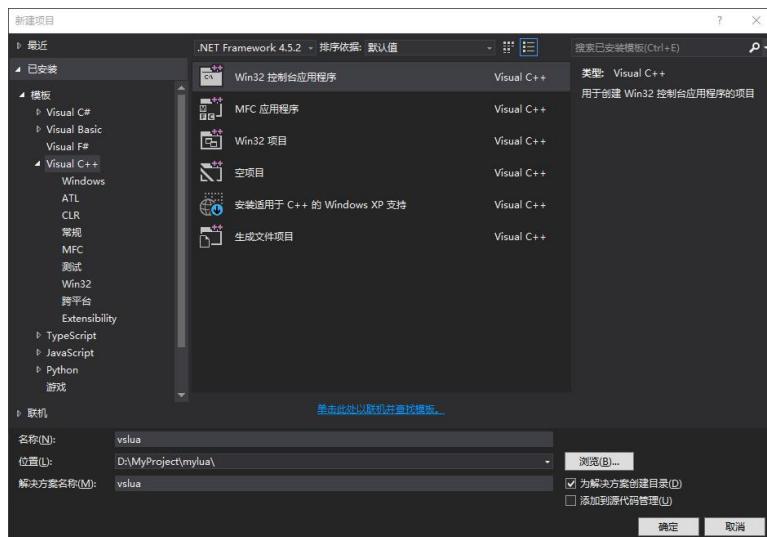
```

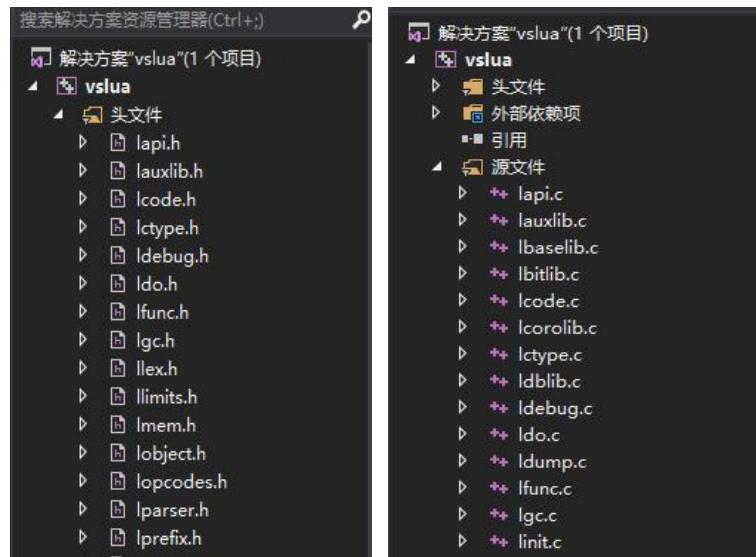
问题 | < > ⚠.

⚠ overriding recipe for target 'debug/main.o'
⚠ ignoring old recipe for target 'debug/main.o'

1. 2. 3. 2. Vs 编译

新建 Vs 的空项目，解压缩 `lua5.3.4.tar.gz`，然后将解压后的 `src` 文中夹中所有 `.h` 加入到 vs 的筛选器头文件中来，然后将解压后的 `src` 文中夹中除了 `lua.c` 和 `luac.c` 以外。`c` 文件加入到筛选器源文件中来。新建 `main.c`，然后完成编译。





1. 2. 3. 3. lua 和 luac

前面在新建工程中将 lua.c luac.c 这两个摘除在外，主要原因是，lua.c 和 luac.c 中分别包含一个 main 函数。即，lua.c 和 luac.c 同其它代码结合在一起是两个完整的工程。

lua.c 可以编译成 lua 交互解析器，而 luac.c 则可以编译成 luac 编译器。推荐大家分别建两个工程，一个叫 lua，一个叫 luac 然后编译后，运行结果。

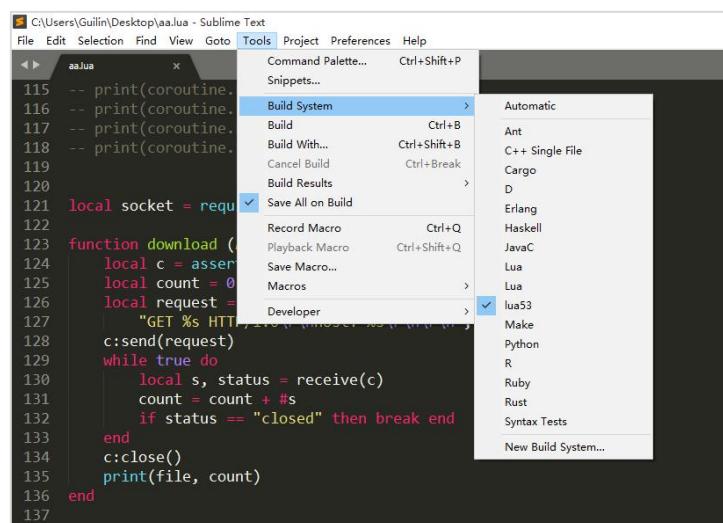
1. 2. 4. Win 集成 lua 环境

1. 2. 4. 1. sublime 环境

1, 将 lua.exe 加入环境变量 PATH

```
C:\Program Files\Java\jdk1.8.0_151\bin;C:\lua
```

2, sublime->Tools->Build System-> New Build System

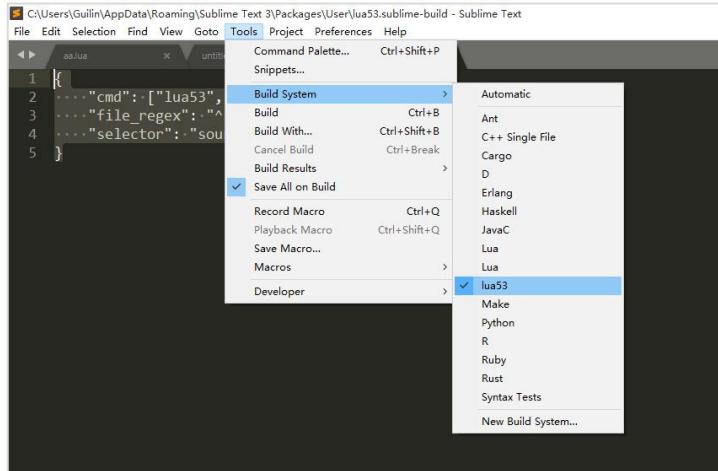


```

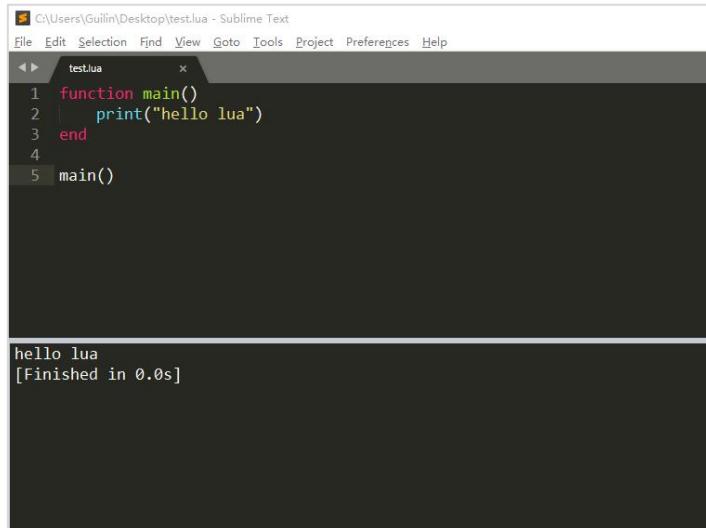
"cmd": ["lua53", "$file"],
"file_regex": "^(?:lua:)?[\t ](...*?)(:[0-9]*):([0-9]*)",
"selector": "source.lua"
}

```

3, 打开新的项目，选择 Build System 中的 lua53



4, 编写程序，ctrl+b 编译



5, Package Control View > Show Console

```

import urllib.request,os,hashlib; h = '6f4c264a24d933ce70df5dedcf1dcaee' +
'ebe013ee18cced0ef93d5f746d80ef60'; pf = 'Package Control.sublime-package'; ipp =
sublime.installed_packages_path();
urllib.request.install_opener( urllib.request.build_opener( urllib.request.ProxyHandler() ) );
by = urllib.request.urlopen( 'http://packagecontrol.io/' + pf.replace(' ', '%20')).read();
dh = hashlib.sha256(by).hexdigest(); print('Error validating download (got %s instead of %s), please try manual install' % (dh, h))
if dh != h else open(os.path.join( ipp, pf), 'wb' ).write(by)

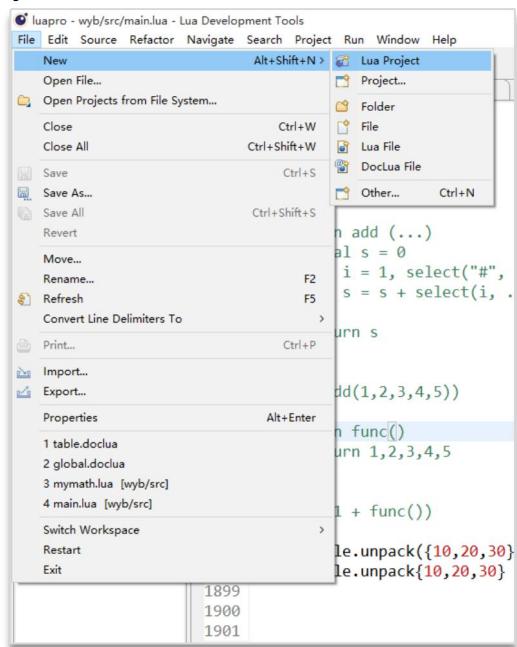
```

6, 插件 LuaFormat ctrl + alt + f

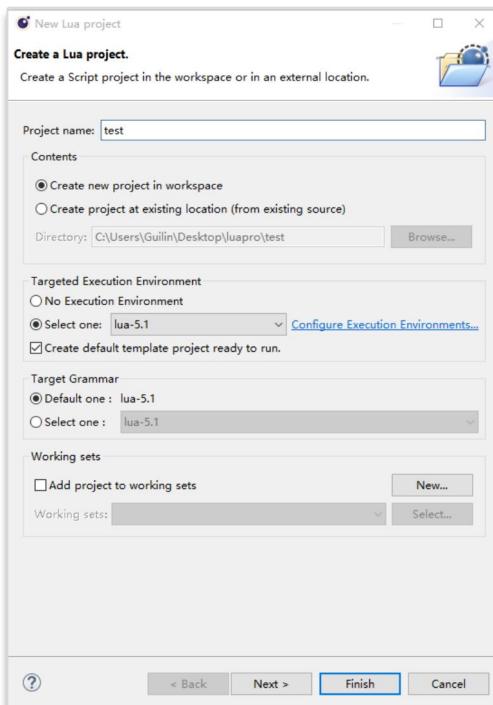
7, 插件 LuaSmartTips

1. 2. 4. 2. Ldt

1, 新建项目 Lua Project



2, 输入工程名称



3 编写程序

1.3. 推荐书目

1.3.1. 书目



1.3.2. 文档

1.3.2.1. 官方文档：

<https://www.lua.org/manual/5.3/>

1.3.2.2. 云风译文：

<http://cloudwu.github.io/lua53doc/manual.html#6.4.1>

1.4. 前提

本教程，假设，你已经懂 C/C++。

1. 5. script why

1. 5. 1. 编译型 vs 解释型

编译型，在每次改动后，都需要重新生成可执行性文件，然后重新部署到生产环境，运行。

解释型语言，只需要一个不变的解析器和随时可变的脚本。

1. 5. 2. 实战解说

1. 5. 2. 1. main.c

1. 5. 2. 2. first.lua

2. 数据类型与变量

2.1. 数据类型

在 C 和 C++ 中，是有变量类型的，可以采用 `typeof(int)` 获取类型大小，亦即变量的大小。在 lua 中，函数 `type` 能够返回一个值或一个变量所属的类型。

即如下所求得的类型，均是 C/C++ 中右值的类型。

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

2.1.1. 空 nil

`nil` 是一种类型，`nil` 类型也只有一种值就是 `nil`。

`nil` 类型有点类似于 C/C++ 中的 `NULL`，其主要作用也是起一个 **标记位** 的作用。the type `nil` has one single value, `nil`, whose main property is to be different from any other value. it usually represents the absence of a useful value。

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

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

2.1.2. 布尔 boolean

布尔类型，`boolean` 类型，可选值只有 `true / false`。

Lua 中 `nil` 和 `false` 为“假”，其它所有值均为“真”。比如 `0` 和空字符串("") 就是“真”。注意，此时的真假判断同 C/C++ 中的真假判断有出入。

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

```

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

if d then
    print("d")
else
    print("not d")
end
if e then
    print("e")
else
    print("not e")
end

```

2.1.3. 数值类型 number

数值类型，表示整数和实数，取值可为任意整数和实数。例：可以使用数学函数 math.floor(向下取整) 和 math.ceil(向上取整) 进行取整操作。

```

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

```

底层实现：一般地，Lua 的 number 类型就是 C/C++ 中的 long long int 或 double 类型来实现的（Standard Lua uses 64-bit integers and double-precision (64-bit) floats）。语法结构的简单是付出空间代价换来的。

2.1.4. 字符串 string

字符串类型，Lua 中其取值，有二种方式：

1、使用一对匹配的单引号。例：'hello'。使用一对匹配的双引号。例："abclua"。建议使用双引号，和 C/C++ 中所学一脉相承。偶尔混和使用，自用妙趣。

2、字符串还可以用一种长括号(即[[]]) 括起来的方式定义。其好处就在于：1 -> 转义并不展开(此举，跟 C++11 中 R"(hello lua)"相似),2 -> 便于多行书写。

```

local str1 = 'hello world'
local str2 = "hello lua"

//偶尔混和使用，自用妙趣
print "i love \"nzhsoft\""
print 'i love "nzhsoft"'
print 'nzhosft is \'great\''
print "nzhosft is 'great'"'

local str3 = [[add\name, 'hello']]

```

```
local str4 = [=[string have a [[]].]=] --[[string have a [[]].]]]

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

[=]=] 等号两侧不可以用空格，主要起到层级/配对的作用。

双引号或单引号的字符串，换行需要加换行符支持，而[[[]]]则不需要

```
str =
" \
<html> \
    <head> \
        <title> </title> \
    </head> \
<body> \
    <p>http://edu.nzhosft.cn</p> \
</body> \
</html> \
"
print(str)

str =
[[[
<html>
    <head>
        <title> </title>
    </head>
<body>
    <p>http://edu.nzhosft.cn</p>
</body>
</html>
]]
print(str)
```

2.1.5. 函数 function

函数，在 Lua 中也是一种数据类型，被称作第一类值(first-class)，也有翻译为一等公民（云风译），其取值有固定函数格式

函数可以存储在变量中，可以通过作参数传递给其他函数，还可以作为其他函数的返回值。其上行为同普通变量无异，普通变量即一等公民。

C/C++98 中，函数的类型多是通过 `typedef` 来提取。直到 C++11 中有 `function` 类模板，才可以自定义函数变量。

```
local function foo()
    print("in the function")
    --dosomething()
```

```

local x = 10
local y = 20
return x + y
end
foo()          --函数调用

local a = foo      --把函数赋给变量

print(a())

local function func( f )
    f()
end

func(foo)        --将函数作参数

-- in the function
-- in the function
-- 30
-- in the function

```

有名函数的定义本质上是：匿名函数对变量的赋值。为说明这一点，其如下写法也是合法的。

```

function foo()
end

```

等价于

```

foo = function ()
end

```

类似的

```

local function foo()
end

```

等价于

```

local foo = function ()
end

```

2.1.6. 表 table

Table 类型中一种基于 k-v 类型，实现了一种抽象的 "map<k, v>"。

"map<k, v>" 是一种具有特殊索引方式的数组，索引 k 通常是字符串(string) 或者 number 类型，但也可以是除 nil 以外的任意类型的值。当下仅关注 index 为 number 和 string 的类型。v 则可以是 lua 中的任意类型。

lua 中的 table 很像是以 string 和 number 为 k 的，C++ 的 STL 中的 hash_map/unordered_map，即 unordered_map<string/number, any>。

```

local corp = {
    web = "www.edu.nzhsoft.cn",      -- k = "web"

```

```

-- v = "edu.nzsoft.cn"
telephone = "17090078295",
-- k = "telephone"
-- v = "17090078295"

staff = {"jack", "Scott", "Gary"},

10086,                                -- key = 1 v = 10086
10010,                                -- key = 2 v = 10010
[10] = 160,                            -- key = 10 v = 160
["city"] = "guangzhou"
}

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

--欲修改之
corp.web      = "www.google.com"
corp["web"]    = "www.baidu.com"
corp[10]       = 360
corp.city     = "beijing"
corp["city"]   = "hang zhou"

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

```

在内部实现上，table 通常实现为一个哈希表和一个数组、或者两者的混合。具体的实现为何种形式，动态依赖于具体的 table 的键分布特点。

规律：

有 key 有则实现为 hash，key 为 string 时有两种表现形式，表内 {web = } {["web"] = } 表外，t.web 和 t["web"]。或 key 为 number 时，表内表外是一个类型。

对于无 key 的类型，此时的 key 类型为 number，下标从 1 开始，下标依次累加。

2.1.7. 线程 thread

见后续章节

2.1.8. 自定义类型 userdata

见后续章节

2.1.9. 分类

Lua 中总共有 8 种数据类型，分别是 nil, boolean, number, string, function, table, thread, userdata。

前 4 种属于基本数据类型，后 4 种属于对象(引用)类型。

2.2. 变量

2.2.1. 命名

变量不过是存储到区域可以操作的名称，即指定内存的别名。它可以容纳不同类型的值，包括函数和表等。

变量名可以由字母，数字和下划线。它必须以字母或下划线开头。大写和小写字母是敏感的，因为 Lua 是区分大小写的。Names (also called identifiers) in Lua can be any string of letters, digits, and underscores, not beginning with a digit and not being a reserved word。Identifiers are used to name variables, table fields, and labels。

此变量名的命名，同 C/C++中的语义基本相符。

2.2.2. 语义

命名虽然同 C/C++中的基本相符，但就其语义来讲，相去甚远。lua 是弱类型/动态型的语言，即，变量没有类型，而数值有类型。



变量的类型是由数值决定的。



Lua is a dynamically typed language。 This means that variables do not have types; only values do。 There are no type definitions in the language。 All values carry their own type。



All values in Lua are **first-class** values。 This means that all values can be stored in variables, passed as arguments to other functions, and returned as results。

```

var = nil           -- nil type
print(var)
var = 199          -- number type
print(var)
var = true          -- boolean type
print(var)
var = function() print "function" end --function type
var()
var = { one = "nzhsoft", two= "japan"}   -- table type
print(var.one)

```

2. 2. 3. 作用域

2. 2. 3. 1. 分类

在Lua，尽管我们没有变量的数据类型，我们基于该变量范围的三种类型。

①全局变量：所有的变量默是全局，除非显式地声明为局部。

②局部变量：当类型被指定为局部的一个变量，它的范围是有限的在自己的范围内使用。

2. 2. 3. 2. scope

C/C++中是以 {} 的方式来划分作用域的。

do end 在lua 中是最小的作用域，此类似于C/C++的大括号{}代码块。lua 中另外一个形式的作用域，就是函数。此举也类似于C/C++函数。

但是，lua 中 scope 是比较弱的，真正决定一个变量作用域的，除了域以外，还要看变量修饰符。

即，**域(函数 / do end)**以内的 **local** 才属于域，域以内的全局依然全局。

```

-- var = 99
-- local var = 99

function func()
    -- var = 100

```

```
local var = 100
print(var)
var = var + 1
end
func()
print(var)
```

2.2.3.3. 变量的作用域

变量的作用范围开始于声明它们之后的第一个语句段，结束于包含这个声明的最内层语句块的最后一个非空语句。

```
x = 10          -- 全局变量
do
    local x = x      -- 新的一个 'x'，它的值现在是 10
    print(x)        --> 10
    x = x + 1
    do             -- 另一个语句块
        local x = x + 1 -- 又一个 'x'
        print(x)        --> 12
    end
    print(x)        --> 11
end
print(x)        --> 10 (取到的是全局的那个)
```

3. 运算符与表达式

3.1. 赋值

3.1.1. 运算符

Lua 可以对多个变量同时赋值，变量列表和值列表的各个元素用逗号分开，赋值语句右边的值会依次赋给左边的变量。 $a, b = 10, 2*x \iff a=10; b=2*x$ 。

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：

- (1) 变量个数>值的个数，多余变量赋值 nil。
- (2) 变量个数<值的个数，多余的值会被忽略。

3.1.2. 示例

```
a, b = 1, 2; c = 1; d = 2
y, x = x, y      -- swap(x,y)
x = 1, 2, 3
x, y = 1
```

3.1.3. Tips

Lua 中的赋值同 C/C++ 中的赋值差别比较大，C/C++ 中常见的如下写法，在 lua 中是不合法的。 $a=b=1$ 或 $\text{if}(a=1) \text{ then } \dots \text{ end}$ 或是 $\text{print}(a=1)$ 的写法。

C/C++ 中表达式是有值的，其值可以零或非零用作逻辑判断，而 lua 中的赋值表达式是没有值的。

3.2. 算术

3.2.1. 运算符

算术运算符	说明
+	加法
-	减法
*	乘法
/	除法(浮点)
^	指数
%	取模(求余)
//	向下取整除法

3.2.2. 示例

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

```

print(2 ^ 10)      -->打印 1024. 求 2 的 10 次方
local num = 1357
print(num % 2)    -->打印 1
print((num % 2) == 1) -->打印 true. 判断 num 是否为奇数
print((num % 5) == 0) -->打印 false. 判断 num 是否能被 5 整数

print(6/3)        --2.0
print(10/3)       --3.3333333333333
print(10/3.0)     --3.333333333333
print(10%3)       --1
print(10%3.1)     --0.7

```

3.2.3. Tips

C/C++中的除法运算，整整得整，整实得实，实实得实。Lua 中的除法运算，皆按实数来算。C/C++中的求余运算，要求参与运算的数必需整数。Lua 中的求余运算，皆为实数运算。

3.3. 关系

3.3.1. 运算符

关系运算符表达式的运算结果，只有 true 和 false。

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

3.3.2. 示例

```

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

```

3.3.3. Tips

3.3.3.1. 类型一致比较

Lua 语言中，运算的类型要相同。`1 == "1"` 运算结果为 `false`，没有意义。"不等于"运算符的写法较为其它语言不同：`~=`。

3.3.3.2. 对象引用比较

在使用 `"=="` 做等于判断时，要注意对于 `table`, `userdata`, `function` 和 `threads` 是对象，在 lua 中对象是引用的方式存在的，赋值、传参、返回均不会引起拷贝行为。

Tables, functions, threads, and (full) userdata values are objects: variables do not actually contain these values, only references to them。Assignment, parameter passing, and function returns always manipulate references to such values; these operations do not imply any kind of copy。

也就是说，只有当两个变量引用同一个对象时，才认为它们相等。而含有相同内容的两个对象，我们认为是不等的。可以看下面的例子：

```
local a = { x = 1, y = 0}
local b = { x = 1, y = 0}

if a == b then
    print("a==b")
else
    print("a~=b")
end

local c = a
if a == c then
    print("a==c")
else
    print("a~=c")
end

--output:
--a~=b
--a==c
```

3.3.3.3. 字符串比较

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

```
str1 = "nzhsoft"
str2 = "nzhsoft"

if str1 == str2 then
    print("str1 == str2")      -- 相等
else
    print("str1 ~= str2")
end
```

3.4. 逻辑

3.4.1. 运算符

逻辑运算符	说明
and	逻辑与

or	逻辑或
not	逻辑非

3.4.2. 示例

在 c 语言中，逻辑运算(`&&` `||` `!`)的结果，只有两种情况 1 或 0，其中 1 表示真，0 表示假。

Lua 中的逻辑运算(`and` `or` `not`)是不同于 c 语言的。逻辑运算的结果，即为参与逻辑运算者之一，除了 `not`。

而 Lua 中逻辑的执行过程是这样的：

a and b	当 a 为真时返回 b，当 a 为假时，返回 a	条件表达式 <code>a?b:a</code>
a or b	当 a 为真时返回 a，当 a 为假时，返回 b	条件表达式 <code>a?a:b</code>
not a	当 a 为真时返回假，当 a 为假时，返回真	条件表达式 <code>a?false:true</code>

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

3.4.3. Tips

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

3.4.4. ?: 三目运算符

```
x = x or b  <=> x?x:b    --如果 x 没有值，则将 b 的值赋值给他，多用于传参默认值

(a and b) or c <=> a?b:c
--a 为真，返回 b 否则返回 c，但是 a 为真，b 是 nil 的时候会返回 c 的值
```

改进之

```
(a and {b} or {c})[1]  --此时 a 为真，表达式为{b}，为真，取其第一个元素的内容
```

3.5. 优先级

Lua 操作符的优先级如下表所示(从高到低)：

优先级
<code>^</code>
<code>not # -</code>

*	/	%
+	-	
.	.	
<>	<=	>=
==	~	=
and		
or		

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

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

4. 控制结构

流程控制语句对于程序设计来说特别重要，它可以用于设定程序的逻辑结构。一般需要与条件判断语句结合使用。

Lua 语言提供的控制结构有 if, while, repeat, for，并提供 break 关键字来满足更丰富的需求。本章主要介绍 Lua 语言的控制结构的使用。

4. 1. do end

lua 中最小的 chunk 结构，是 do end 相当于 c/c++ 中最小作用域结构({})一样。同函数不一样的地方，就在于执行性，函数是需要调用的。

```
do
    print("do ... end")
end

function my_print()
    print("do ... end")
end
my_print()
```

```
num = 10
do
    local num = 1
    print(num)
end

function my_print()
    var = 100
end
-- my_print() -- 执行与不执行有差距吗?
print(var)
```

4. 2. 选择 if else

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

4. 2. 1. 单个分支型

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

4.2.2. 两个分支 if-else 型

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

4.2.3. 多个分支 if-elseif-else 型

lua 的多路选择格式: if then --elseif then-- elseif then-- else --end

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

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

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

4.3. 循环

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

4.3.1. while 循环

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

```
while 表达式 do
    --body
end
```

示例

```
local x = 1
```

```

local sum = 0
while x <= 5 do
    sum = sum + x      --lua 中没有 +=
    x = x + 1          --lua 中没有 x++这样的写法
end
print(sum) -->output 15

```

4. 3. 2. repeat 循环

Lua 中的 repeat 控制结构类似于其他语言(如：C++ 语言) 中的 do-while，但是控制方式是刚好相反的。

简单点说，执行 repeat 循环体后，直到 until 的条件为真时才结束，而其他语言(如：C++ 语言) 的 do-while 则是当条件为假时就结束循环。

```

x = 10
repeat
    print(x)
until false

```

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

while 当假是退出，**repeat** 直到为真时才退出。分别用 while 和 repeat 来求 $1+2+3+\dots+100$ 的和：

```

local sum = 0
local i = 0

-- while i <= 100 do
--     sum = sum + i
--     i = i + 1
-- end

-- print(sum, i)

repeat
    sum = sum + i
    i = i + 1
until i == 101
print(sum, i)

```

Differently from most other languages, in Lua the scope of a local variable declared inside the loop includes the condition:

```

-- computes the square root of 'x' using Newton-Raphson method
local sqr = x / 2
repeat
    sqr = (sqr + x/sqr) / 2

```

```
local error = math.abs(sqr^2 - x)
until error < x/10000           -- local 'error' still visible here
```

C 中的情况是什么样的呢？

```
#include <stdio.h>

int main()
{
    do{
        int i=10;
        printf("%d\n",i--);
    }while(i);    // error: 'i' undeclared (first use in this function
    return 0;
}
```

4.3.3. for 控制结构

for 语句有两种形式：数字 for(numeric for) 和范型 for(generic for)，循环的控制变量默认是局部的，循环完了就没了。

4.3.3.1. numeric for

var 从 exp1 变化到 exp2，每次变化以 exp3 为步长递增 var，并执行一次"执行体"。exp3 是可选的，如果不指定，默认为 1。

```
for var = exp1, exp2, exp3 do
    <执行体>
end
```

① 三个表达式，只计算一次

② 且 var 是一个局部变量。

③ you should not change the value of the control variable: the effect of such changes is unpredictable。

示例

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

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

math.huge, 数学中的最大值。

```
for i = 1, math.huge do
```

```

if (0.3*i^3 - 20*i^2 - 500 >= 0) then
    print(i)
    break
end
end

```

4.3.3.2. generic for

泛型 for 循环通过一个迭代器函数来遍历所有值，类似 C++ 模板库 STL 中的 foreach 语句。

Lua 编程语言中泛型 for 循环语法格式，*i* 是数组索引值，*v* 是对应索引的数组元素值。*ipairs* 是 Lua 提供的一个迭代器函数，用来迭代数组。

```

-- 打印数组 a 的所有值
for i,v in ipairs(a) do
    print(v)
end

```

泛型 for 循环通过一个迭代器(iterator) 函数来遍历所有值：

```

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

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

```

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

4.3.4. Tips

循环，我们最关系的是，起始与结束。while 条件为假时退出，即直到为假时为止。repeat 条件为真是退出，即直到为真是为止。

至于 for, number for 是基于步长加范围的。generic for 是基于迭代器的。

4.4. 跳转

break 和 *return* 可以跳出程序块，而 *goto* 可以跳到任意位置。

4.4.1. break

break 只用于循环结构 for, repeat, while。*break* 充当阈值的作用，到了点，就跳出循环。

```

for i = 1, 10 do

```

```

if i == 5 then
    break;
end
print(i)
end

```

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

官方解释为什么没有 `continue`: <http://www.luafaq.org/#T1.26>

没有提供 `continue`，却也提供了另外一个标准控制语句 `break`，可以跳出当前循环。比如打印 10 以内的偶数：

```

for i = 1, 10 do
    if i % 2 == 0 then
        print(i)
    end
end

```

没有用到 `continue`, `continue` 有过滤的功效，强改逻辑，用到了 `goto` 来越过不需要执行的代码，而进入下一次循环。

```

for i = 1, 10 do
    if i % 2 ~= 0 then
        goto END
    end
    print(i)
::END::
end

```

4.4.2. return

`return` 用于从函数返回一个结果，或是结束函数。当然了，也可以隐式的不写。

出于语法原因，For syntactic reasons, a `return` can appear **only** as the last statement of a block, 换句话说，`return` 只能出现在，语句块的结尾，或是 `end`, `else` 和 `until` 的前面。

```

local i = 1
while a[i] do
    if a[i] == v then return i end
    i = i + 1
end

```

如下：

```

function foo ()
    return --<< SYNTAX ERROR
    -- 'return' is the last statement in the next block
    do return end -- OK

```

```
other statements
end
```

4. 4. 3. goto

A label is visible in the entire block where it is defined, except inside nested blocks where a label with the same name is defined and inside nested functions. A goto may jump to any visible label as long as it does not enter into the scope of a local variable.

4. 4. 3. 1. 引入

goto 语句，可以跳到程序中的任意 Label 标号，其 Label 的书写方式 ::name::。注意标号处的语句是依次执行的。不要理解为 Label 后的语句是函数，等待被调用。

```
i = 0
::Label::
do
    print(i)
    i = i+1
end
if i>3 then
    os.exit()
end
goto Label
```

4. 4. 3. 2. 限制

但不同于其它语言，有如下限制：

1. Label 可见原则：不能在 block 外面跳入 block(因为 block 中的 Label 不可见)

```
goto ok
do
    local i = 1
    print(i)
    ::notok::
    i = i+ 1
    ::ok::
end           -- no visible label 'ok' for <goto>
```

2. 不能跳出或者跳入一个函数。函数也是一种 block

```
function func()
    goto ok
    local i = 1
    print(i)
    i = i+ 1
end
func()
::ok::
print("goto out") -- no visible label 'ok' for <goto>
```

3. 不能跳入本地变量的作用域。

```
do
```

```

goto ok
local i = 1
print(i)
::notok::      -- 本地变量的作用域内，所以无法跳转
i = i + 1
::ok::         -- 本地变量的作用域结束，所以可以跳转。
end           -- <goto notok> at line 884 jumps into the scope of local 'i'

```

4.4.3.3. 应用之 continue 与 redo

A continue statement is simply a goto to a label at the end of a loop block; a redo statement jumps to the beginning of the block.

```

i = 0
while i < 10 do
::redo::
    i = i + 1
    if i % 2 == 1 then
        goto continue    -- 条件不满足,下一个 continue
    else
        print(i)
        goto redo       -- 条件满足,再来一次 redo
    end
::continue::
end

```

4.4.3.4. 应用之状态机

可以从一个 Label 跳到另外一个 Label。

```

goto room1
::room1::
do
    print("in room 1")
    local move = io.read()
    if move == "south" then goto room3
    elseif move == "east" then goto room2
    else
        print("invalid move")
        goto room1 -- stay in the same room
    end
end

::room2::
do
    print("in room 2")
    local move = io.read()
    if move == "south" then goto room4
    elseif move == "west" then goto room1
    else

```

```
    print("invalid move")
    goto room2
end

::room3::
do
    print("in room 3")
    local move = io.read()
    if move == "north" then goto room1
    elseif move == "east" then goto room4
    else
        print("invalid move")
        goto room3
    end
end

::room4::
do
    print("in room 4")
    print("Congratulations, you won!")
end
```

到命令终端中运行，lua53.exe room.lua 依次输入，即可 io.read()从键盘读取输入的数据。

5. string 字符串

5.1. string

5.1.1. 定义

Lua 中有三种方式表示字符串：

使用一对匹配的单引号.例: 'hello'.

使用一对匹配的双引号.例: "abclua".

字符串还可以用一种长括号(即[[]]) 括起来的方式定义.

5.1.2. 解析

对于前两种方式，没有什么好解释的。

我们把两个正的方括号(即[D]间插入 n 个等号定义为第 n 级正长括号。就是说，0 级正的长括号写作 [[，一级正的长括号写作 [=，如此等等。反的长括号也作类似定义；举个例子，4 级反的长括号写作]====]。

一个长字符串可以由任何一级的正的长括号开始，而由第一个碰到的同级反的长括号结束。

```
html = [[
<html>
<head></head>
<body>
    <a href="http://edu.nzhsoft.cn/">能众软件</a>
</body>
</html>
]]
print(html)
```

整个词法分析过程将不受分行限制，不处理任何转义符，并且忽略掉任何不同级别的长括号。这种方式描述的字符串可以包含任何东西，当然本级别的反长括号除外。例：[[abc\nbc]]，里面的 "\n" 不会被转义。

另外

Lua 的字符串是不可改变的值，不能像在 c 语言中那样直接修改字符串的某个字符，而是根据修改要求来创建一个新的字符串。

Lua 也不能通过下标来访问字符串的某个字符。

5.1.3. 操作

```
print("nzhsoft");
print('nzhsoft');
print([==[ [nzh\nsoft]==]);;

//偶尔混和使用，自用妙趣
print "i love \"nzhsoft\""
```

```
print 'i love "nzhsoft"'
print 'nzhosft is \'great\''
print "nzhosft is 'great'"
```

```
str = "nzhosft"
print(str[1]) --nil
```

5.1.4. ASCII

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

5.1.5. number 与 string

number 与 string 之间，界限。注意 10 ..11 的时候 10 的后面有一个空格。

```
print(10 + '11')
print("11" + 10)
print(10..11)

print(type(10 ..11))

--清清楚楚，明明白白
print tonumber("11") + 10
print tostring(10 .. "111")
```

5.2. string 库

任何一门语言，字符串都是重中之重，企业中 90% 的时间，都在跟字符串打交道，所以系统提供的函数，不可多得。

5.2.1. API 列表

S.N.	函数及其功能
0	.. 连接两个字符串。若连接两个数字，数字会默认转化为字符串，如果数字在前，比如 1234 .."Lua"。1234 和 .. 之间要加空格。
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.format(...) 返回格式化后的字符串。
4	string.len(arg) : 返回输入字符串的长度。#操作符也可以求长度
5	string.rep(string, n) : 将输入字符串 string 重复 n 次后的新字符串返回。
6	string.char(...) 接收 0 个或更多的整数(整数范围：0~255)，返回这些整数所对应的 ASCII 码字符组成的字符串。
7	string.byte(s [, i [, j]]) 返回字符 s[i]、s[i + 1]、s[i + 2]、……、s[j] 所对应的 ASCII 码。 i 的默认值为 1，即第一个字节，j 的默认值为 i。 若 j 不为空，返回 i,j 范围内的以 tab 间隔的 ASCII 码
8	string.find(s, p [, init [, plain]]) 在 s 字符串中第一次匹配 p 字符串。若匹配成功，则返回 p 字符串在 s 字符串中出现的开始位置和结束位置；若匹配失败，则返回 nil。 第三个参数 init 默认为 1，并且可以为负整数，当 init 为负数时，表示从 s 字符串的 string.len(s) + init 索引处开始向后匹配字符串 p。第四个参数默认为 false，当其为 true 时，只会把 p 看成一个字符串对待。
9	string.sub(s, i [, j]) 返回字符串 s 中，索引 i 到索引 j 之间的子字符串。 当 j 缺省时，默认为 -1，也就是字符串 s 的最后位置 -2 为倒数第二个位置。 当索引 i 在字符串 s 的位置在索引 j 的后面时，将返回一个空字符串。
10	string.gsub(mainString, findString, replaceString) : 将 mainString 中的所有 findString 用 replaceString 替换并返回替换结果。
11	string.reverse(arg) : 将输入字符串颠倒并返回。

5.2.2. 测试

5.2.2.1. upper/lower

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

```
t = {"ax","Cy","mk","Bjlk"}
for k,v in ipairs(t) do
    print(k,v)
end
table.sort( t, function (a,b)
    return string.lower(a)<string.lower(b)
end )
for k,v in ipairs(t) do
    print(k,v)
end
print(table.concat(t,"-"))
```

5.2.2.2. format

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()`。

```
s4 = "Lua"
s5 = "nzhsoft"
print(string.format("Basic formatting %s==%s",s4,s5))

-- Date formatting
date = 2; month = 1; year = 2014
print(string.format("Date formatting %02d/%02d/%03d", date, month, year))

-- Decimal formatting 小数格式
print(string.format("%.4f",1/3))

print(string.format("%c", 83)) --输出 S
print(string.format("%+d", 17.0)) --输出+17
print(string.format("%05d", 17)) --输出 00017
print(string.format("%o", 17)) --输出 21

print(string.format("%x", 13)) --输出 d
print(string.format("%X", 13)) --输出 D
print(string.format("%e", 1000)) --输出 1.000000e+03
print(string.format("%E", 1000)) --输出 1.000000E+03
print(string.format("%6.3f", 13)) --输出 13.000

print(string.format("%s", "monkey")) --输出 monkey
print(string.format("%10s", "monkey")) --输出      monkey
```

5. 2. 2. 3. **char/byte/len**

```
--字符与整数相互转换
print(string.char())
print(string.char(48, 49, 50, 51, 52))          --01234
print(type(string.char(48, 49, 50, 51, 52)))      --string
print(string.byte("12345"));
print(string.byte("12345", 3));                   -- 51
print(string.byte("12345", 1, 5));                --49 50 51 52 53
print(string.len("12345"))                      --5
print(string.rep("nzhsoft", 5))                  --nzhsoftnzhsoftnzhsoftnzhsoftnzhsoft
```

5. 2. 2. 4. **sub/replce/find/reverse**

```
--string.sub
s2 = "Lua nzhsoft"
print(string.sub(s2,2))                         --ua nzhsoft
print(string.sub(s2,2,5))                       --ua n
print(string.sub(s2,2,-2))                      --ua nzhsof 去除头尾

-- replacing strings
ns = string.gsub(s2,"nzhsoft","Language")
print("The new string is",ns)

-- find strings
s3 = "Lua Tutorial"
print(string.find(s3,"nzhsoft"))
print(string.find(s3,"nzhsoft",-1))

--reverse
rs = string.reverse(s3)
print("The new string is",rs)
```

5. 3. **string.gmatch**

学习，迭代和正则以后，有详解。

6. Array Table

6.1. 综述

`table` 是 Lua 中唯一的数据结构，其他语言所提供的数据结构，如：`arrays`、`records`、`lists`、`queues`、`sets` 等，Lua 都是通过 `table` 来实现，并且在 lua 中 `table` 很好的实现了这些数据结构。

6.1.1. 概念

在 lua 中通过整数下标访问 `table` 中元素，即是数组。并且数组大小不固定，可动态增长。

数组是有序的对象的装置，它可以是包含含有多个行和列的行或多维阵列的集合的单个二维数组。

6.1.2. 下标从 1 开始

在 Lua 中，数组下标从 1 开始计数。

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

确实，对于我们数数来说，总是从 1 开始数的，而从 0 开始对于描述偏移量是比较有利，如果内存型语言，C/C++。

而 Lua 最初设计是一种类似 XML 的数据描述语言，所以索引(index) 反应的是数据在里面的位置，而不是偏移量。

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

只有键值从 1 开始连续的元素放在 `table` 里，才可以作为数组 `sequence` 使用，否则针对 `sequence` 的操作行为都是未定义的。

6.2. 一维数组

6.2.1. 默认下标

一维数组可以用一个简单的表结构来表示，可以初始化，使用一个简单的 for 循环读取。如下例子所示。

程序

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

输出结果

```
nil
Lua
nzhsoft
```

6.2.2. 给定下标(强烈不推荐)

正如在上面的代码中看到，当我们试图访问索引中是不存在的数组中的元素，则返回 nil。在 Lua 索引通常开始于索引 1，但也可以使用非 1 起始索引。下例显示使用负数索引数组，我们初始化使用 for 循环数组。

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

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

6.3. 二维数组

多维数组可以用两种方式来实现。

- 数组的数组(一维数组中的每个元素，又是一个数组)
- 一维数组通过控制索引(一维数组，二维逻辑)

6.3.1. 数组的数组

Lua 中有两种表示矩阵的方法，一是"数组的数组"。也就是说，table 的每个元素是另一个 table。例如，可以使用下面代码创建一个 n 行 m 列的矩阵：

```
-- Initializing the array
array = {}
for i=1,3 do
    array[i] = {}
    for j=1,3 do
        array[i][j] = i*j
    end
end
-- Accessing the array
for i=1,3 do
    for j=1,3 do
        io.write(array[i][j], " ")
    end
    io.write("\n");
end
```

当我们运行上面的代码之后，将得到下面的输出。

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

本质探究：

```
for k,v in pairs(array) do
    print(k,v)
end
-- 1  table: 00388128
-- 2  table: 00387CF0
-- 3  table: 00387E80
```

6.3.2. 一维数组通过控制索引

```
array = {}

Rows = 3
Columns = 3

for i = 1,Rows do
    for j = 1, Columns do
        array[(i-1)*Columns + j] = i + j
    end
end

for i =1, Rows do
    for j = 1,Columns do
        io.write(array[(i-1)*Columns + j]," ")
    end
    io.write("\n")
end

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

-- 2 3 4
-- 3 4 5
-- 4 5 6

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

```
-- 8 5
-- 9 6
```

6.4. table->array 库

6.4.1. 常用 Api 列表

```
--table 本身也是一个对象，它的成员函数，可以操作其它 table 对象
for k,v in pairs(table) do
    print(k, v)
end

sort      function: 62995930
remove   function: 62995d00
concat   function: 62996060
move     function: 62995a00
insert   function: 62995ec0
unpack   function: 62995080
pack     function: 62995220
```

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。 <code>sep</code> 是分割符， <code>i, j</code> 是下标和区间起始。
2	table.insert(table, [pos,]value): 在表中指定位置插入一个值，其它值后移。 <code>pos</code> 是下标索引， <code>value</code> 是插入值，在不指定位置的情况下，插在尾部。
3	table.maxn(table): 返回表中最大的数值索引。仅适用于 5.1 版本， 5.3 采用#运算符
4	table.remove(table[, pos]): 从表中移除并返回指定位置的元素，然后将其后的元素向前移动填充删除元素后造成的空洞。若不指定位置，则该函数删除序列的最后一个元素。
5	table.sort(table[, comp]): 根据指定的(可选)比较方法对表进行排序操作。

6.4.2. 测试

6.4.2.1.

Lua 语言提供了获取序列长度的操作符#，求字符串的长度也可以使用此操作符。于而言返回对应序列的长度。

```
print(#"abc")
print(#{1,2,3,4,5,6})
```

常用的#号操作：末尾操作，遍历

```
t = {11,22,33,44,55,66}
print(t[#t])      -- 输出最后一个元素
t[#t] = nil       -- 移除最后一个元素
t[#t +1 ] = 999   -- 把 999 添加到序列最后
```

对于中间存在空洞 nil 的列表而言，序列长度的操作是不可靠的。它只能用于序列(所有元素不为 nil)。序列 sequence 是由指定的 n 个正数数值类型的键所组成的集合形成的表，特别的，不包含数值类型键的表，说法是长度为零的序列。

```
t2 = {1,2,3}
print(#t2)      --3

t = {x = 1,y = 2, z = 3}
print(#t)      -- 0
```

虽然，这是 lua 比较奇葩的语法格式，但是大多数列表其实都是序列(比如，不能为 nil 的文件行)。

正因为如此，在多数情况下，长度操作符都是安全的。在确实需要处理存在空洞的列表时，应该将列表的长度显示的保存起来。

6.4.2.2. concat

```
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))
--基于索引连接 fruits
print("Concatenated string ",table.concat(fruits, " ", 1,3))
```

6.4.2.3. insert&remove

insert 和 remove 在没有指定下标的时候，指的是从尾部插入和删除。基于 table 的插入和删除是高效的。

```
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])
print("all fruit is:", table.concat(fruits, " ", "))

-- 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("all fruit is:", table.concat(fruits, " ", ""))
print("The maximum elements in table is", #fruits)
```

```

print("The last element is", fruits[5])
table.remove(fruits)
print("all fruit is:", table.concat(fruits, ", "))
print("The previous last ement is", fruits[5])

```

借助这两个函数，可以轻松的是实现，栈(Stack),队列(Queue)和双端队列(Double queue)。以栈为例：`t = {}` Push 操作可以使用 `table.insert(t, 1, x)` 实现，Pop 操作可以使用 `table.remove(t, 1)`。

```

t = {}

table.insert(t,1,"a")
table.insert(t,1,"b")
table.insert(t,1,"c")

for i=1,#t do
    print(table.remove(t,1))
end

```

6.4.2.4. sort

`sort` 支持默认的排序规则，也可以人为的指定排序规则，此时的回调函数是 first-class 级别的，所以用起来，很方便。el

```

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

comp = function(sa, sb)      --回调函数
    return sa > sb
end

table.sort(fruits, comp)    --回调传参
print("sorted table")
for k, v in ipairs(fruits) do
    print(k, v)
end

```

6.4.2.5. move

Lua5.3 对于移动表中的元素引入了一个更通用的函数，`table.move(a,f,e,t)` 调用该函数，可以将表 `a` 中从索引 `f` 到 `e` 的元素(闭区间)，移动到本表 `a` 位置 `t` 上。

```
t = {"a","b","c"}

table.move(t,1,#t,2)
t[1] = "m"

for i=1,#t do
    print(t[i])
end
```

如上代码，实现了在列表 t 的开头，插入了一个元素。也可以采用此方法删除一个元素。注意，显示的删除。

```
t = {"a","b","c"}

table.move(t,1,#t,2)
t[1] = "m"

for i=1,#t do
    print(t[i])
end
print("====")
table.move(t,2,#t,1)
for i=1,#t do
    print(t[i])
end
print("====")
t[#t] = nil

for i=1,#t do
    print(t[i])
end
```

`table.move` 还支持使用一个表作为可选参数 `table.move(a, f, e, t, {})`，当参数是一个表时，该函数将一个表中的元素 clone 第二个表中。

```
t = {"a","b","c"}

cp = table.move(t,1,#t,1,{})
for k,v in ipairs(t) do
    print(k,v)
end
print("====")
for k,v in ipairs(cp) do
    print(k,v)
end
print("====")
```

将列表 t 中的所有元素，拷贝到新表中并返回。下面，将列表 t 中所有的元素拷贝到 t2 表中的末尾。

```
t = {"a","b","c"}  
44
```

```
t2 = {"x","y","z"}
cat = table.move(t,1,#t,#t2,t2)

for k,v in ipairs(cat) do --ipairs(t2)
    print(k,v)
end
```

6.5. 数组中无哈希(序列)

6.5.1. 序列

数组，就要保持其为纯的数组。如下，这种数组应该当作哈希来处理，不能称其为数组。

```
a = {[1] = 1,[999] = 999}           --1
print(#a)
t = {[2] = 1,[999] = 999}           --0
print(#t)
```

真正的数组，必须满足，key 值为整型，下标从 1 开始，且连续，如果这些条件满足，不能用数组的方式处理之。

```
a2 = {[1] = 1,[2] = 999}           --2
print(#a2)

a2 = {[1.1] = 1,[2] = 999}
print(#a2)

t2 = {1,999}                      --2
print(#t2)
```

数组在 lua 中，多翻译为序列 sequense(云风译版)，这个说法比较贴切，引意为索引有序的队列。故看到序列，即数组的意思。

6.5.2. 序列中的 nil 无效值

6.5.2.1. # ipairs 失效

数组中的 nil 影响求长度，长度不定的数组，无法通过下标对其遍历，迭代函数 ipairs 遍历则会中止。

```
array = {1,2,3,4,5,6,7,8,9,10}
print("len =",#array)
for k,v in ipairs(array) do
    print(k,v)
end

print("====")
array2 = {1,2,3,4,nil,6,7,8,9,10}
print("len =",#array2)
for k,v in ipairs(array2) do
    print(k,v)
```

```
end
print("-----")
for i=1,#array2 do
    print(i,array2[i])
end
```

```
len = 10
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
10 10
=====
len = 4
1 1
2 2
3 3
4 4
-----
1 1
2 2
3 3
4 4
-----
[Finished in 0.1s]
```

6. 5. 2. 2. pairs 有效

```
array2 = {11,22,33,44,nil,66,77,88,99,100,nil}
for k,v in pairs(array2) do
    print(k,v)
end
```

```
1  11
2  22
3  33
4  44
6  66
7  77
8  88
9  99
10 100
[Finished in 0.1s]
```

6.5.2.3. 解决方案

数组中尽可能不要放置无效值 nil，实在无可而为之，则可以采用记录数组长度的方式或用 ipairs 来遍历。

```
t = {11,22,33,nil,44,55,nil}
tn = 7
for i=1,tn do
    -- if t[i] then
    --     print(t[i])
    -- end
    print(t[i])
end
```

```
t = {11,22,33,nil,44,55,nil}

for _,v in pairs(t) do
    print(v)
end
```

7. Map Table

7.1. 定义

7.1.1. 引入 i-v->array

本章中讲的 table 跟前一章讲的是同一个 table，前一章讲的是 table 中 k 为整数的情况，初始表 `t = {"r", "g", "b"}` 和 `t = {[1] = "r", [2] = "g", [3] = "b"}`，是等价的，通常我们会写为第一种形式。

对于其访问方式，格式也是固定的，`print(t[1])`；`t[1] = 1000`；如果 k 为其它类型，如何，比如，最常见的 string 类型呢？

7.1.2. k-v->hash

7.1.2.1. 初始化

`t = {x = 0, y = 0}` `<->` `t = {[x] = 0, [y] = 0}` 两种表的初始化方式是等价的。表中有两个键值对，且 key 为 string 类型。

7.1.2.2. 成员赋值

`t.x = 100; t.y = 200;` `<->` `t["x"] = 300; t["y"] = 400`

7.1.2.3. 成员访问

`print(t.x);` `<->` `print(t["y"]);`

7.1.2.4. 小结

数组的初始化，存入单值，哈希初始化，放的是键值对。数组的访问方式是统一的`[]`，哈希的访问方式有两种，一种是`.`，一种是`[]`。

7.1.3. 深入 k

7.1.3.1. 引例

```
a = {}           -- create a table and assign its reference
k = "x"
a[k] = 10        -- new entry, with key="x" and value=10
print(a["x"],a[k])

a[20] = "great"    -- new entry, with key=20 and value="great"
k = 20
print(a[k],a[20])

a["x"] = a["x"] + 1  increments entry "x"
print(a["x"])
```

7.1.3.2. a.x with a["x"]

```
a = {}
x = "y"
a[x] = 10      -- put 10 in field "y"
```

```
print(a[x])    --> 10 -- value of field "y"
print(a.x)     --> nil -- value of field "x" (undefined)
print(a.y)     --> 10 -- value of field "y"  a.y =>a["y"]
```

7.1.3.3. index in any type

可以用任意类型的值来作数组的索引，但这个值不能是 nil。比如：t = {} t[key] = "edu.nzhsoft.cn"，由报错为 table index is nil。

```
i = 10; j = "10"; k = "+10"; -- k = "x10"
a = {}
a[i] = "number key"
a[j] = "string key"
a[k] = "another string key"
print (a[i]) --> number key
print (a[j]) --> string key
print (a[k]) --> another string key
print (a[tonumber(j)]) --> number key
print (a[tonumber(k)]) --> number key

a = {}
a[2.0] = 10           -->float key
a[2.1] = 20

print(a[2])
print(a[2.0])
print(a[2.1])
```

7.1.4. 遍历

7.1.4.1. map table 无长度可求

```
stu = {
    name = "guilin",
    sex = "male",
    age = 18,
    org = "edu.nzhsoft.cn"
}

-- map table 的长度，不可求，故 ipairs 迭代不可用
print("len of stu = ",#stu)
for k,v in ipairs(stu) do
    print(k,v)
end
```

7.1.4.2. 迭代函数 pairs

```
-- pairs 可用，且无序
for k,v in pairs(stu) do
```

```
print(k,v)
end
```

7.2. 引用

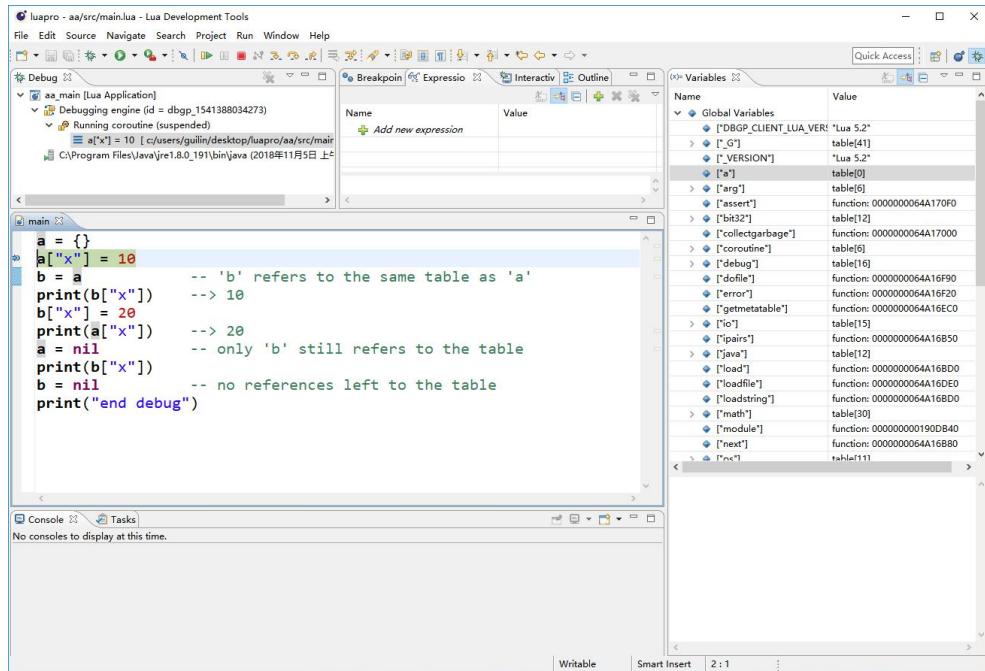
7.2.1. 表有名字吗？ anonymous

我们用到的所有的 table 都是匿名的，所有的变量，只是对其引用而已。我们对于 table 的操作仅仅是对其引用的操作。

对于 Table 的引用，类似于 C++ 中 shared_ptr 对于对象的托管一样，采用了引用计数的方式。

```
a = {}
a["x"] = 10
b = a          -- 'b' refers to the same table as 'a'
print(b["x"])  --> 10
b["x"] = 20
print(a["x"])  --> 20
a = nil        -- only 'b' still refers to the table
print(b["x"])
b = nil        -- no references left to the table
print("end debug")
```

能过 ldt 调试，发现，`b=a` 以后，即 `b` 引用了 `a`，`b` 中的内容同 `a` 中内容无异。当 `a=nil` 以后，`a` 引用中的内容消失，`b=nil` 以后，`b` 消失了，即，表消失了。



通过调试也发现，所谓全局变量，就是常驻虚拟机的变量，而局部变量，则是用时开辟，用即消失。

7.2.2. 表可以赋值，比较吗？

7.2.2.1. 只有引用

表不存在赋值，只有引用。假设 `a` 引用一张表，`b` 也引用一张表，现在将 `b = a`，只能说 `b` 不再引用原先的表，现在同 `a` 引用同一个表。切勿理解为将 `a` 指向的表中的内容，拷贝到了 `b` 引用的表中。

7.2.2.2. 拷贝 copy

若要真的实现一份拷贝，如何来作呢？`table.move` 可以吗？

```
t = {x = 1,y = 2, z = 3}
print(#t)
cp = table.move(t,1,#t,1,{})
for k,v in pairs(cp) do
    print(k,v)
end
```

如果上述操作失败怎么办？

```
t = {x = 1,y = 2, z = 3}

function copy(a,b)
    a = a or {}
    for k,v in pairs(b) do
        a[k] = v
    end
    return a
end

cp = copy(cp,t)

for k,v in pairs(cp) do
    print(k,v)
end
```

7.2.2.3. 比较 ==

两张表，可以比较吗？

```
a = {1,2,3}
b = {1,2,3}

if a == b then
    print("a == b")
else
    print("a != b")
    print(a,b)
end

r = a

if r == a then
```

```

print("r == a")
print(r,a)
else
    print("r != a")
end

```

7.3. # table

表中元素的个数并不是很容易获取，结果取决于你怎么做（或你怎么定义“长度”）。这可能不是个意外，因为 Lua 提供了强大的表并支持灵活的索引方式（数字或其它 Lua 类型，除了 nil）。

Lua 中的表有两部分：“数组”部分（使用 `t = {1, 2, 3}` 生成）和“哈希”部分（使用 `t = {a = "foo", ["b"] = 2}` 生成）；这两者可以灵活地结合在一起。

`#table` 返回最短的“数组”部分长度（没有任何缺口，即下标连续）而 `table.maxn(t)` 返回最长的“数组”部分（Lua 5.3 移除了这个函数）。“哈希”部分没有定义长度。两者都可以使用 `pairs` 方法进行遍历，同时允许你对其中的元素进行计数。

```

print(#{1, 2, 3, 4})           -- 4
print(#{1, 2, 3, 4, 5})       -- 5
print(#{1, 2, [10] = 100, 3, 4, 5})   -- 5
print(#{1, 2, [10] = 100, 3, 4, 5, name="nzhsoft", age=10}) -- 5

```

有了 nil 以后，一切变的迷离起来

```

print(#{1, 2, nil, 3})
print(#{1, 2, nil, 3, nil})

```

然而，`print(#{1, 2, nil, 3})` 打印 4 却不是想像中的 2，`print(#{1, 2, nil, 3, nil})` 打印的则是 2。

Despite all these discussions, most lists we use in our programs are sequences (e.g., a file line cannot be nil) and, therefore, most of the time the use of the length operator is safe. If you really need to handle lists with holes, you should store the length explicitly somewhere.

```

t = {1,nil,3,nil,4,nil,5,nil,6}
n = 9
for i=1, n do
    if t[i] ~=nil then
        print(t[i])
    end
end

```

只有键值从 1 开始连续的元素放在 `table` 里，才可以作为数组 `sequence` 使用，否则针对 `sequence` 的操作行为都是未定义的。

怎么才算连续呢？lua 把 value 为 nil 的键值对视为不存在。所以，`{1,2,nil,4}` 就不是一个 `sequence`，因为 3 号位是不存在的。当你用 # 取这个 `table` 的 `sequence` 长度时，到底是 2 还是 4 就是未定义的。

7.4. 哈希中无数组

7.4.1. 纯 map

哈希，就要保持其为纯纯的哈希。即仅用于存入键值对使用，若混有数组的部分，语义不明确是其一，还可能要单独处理。

```
a = {x = 1,y = 999}
print(#a)
```

7.4.2. 混合 table

```
a = {111,222,x = 1,y = 999,999}

for k,v in ipairs(a) do      --数组部分
    print(k,v)
end

for i=1,#a do              --数组部分
    print(i,a[i])
end

for k,v in pairs(a) do     --全部
    print(k,v)
end
```

8. Function 函数

8.1. 概述

8.1.1. 函数的意义

函数是一组一起执行任务的语句。可以把代码放到独立的函数中。怎么划分代码功能之间的不同，但在逻辑上划分通常是让每个函数执行特定的任务。

无论面向对象，还是面向过程，函数都是必不可少的。功能抽象化到独立的函数，可以避免重复造轮子。

Lua 语言提供了程序可以调用大量的内置方法。例如，方法 `print()` 打印作为输入传参数在控制台中。

8.1.2. 定义与示例

8.1.2.1. 格式(语法糖)

```
optional_function_scope
function function_name( arg1, arg2, arg3..., argn)
    function_body
    return result_params_comma_separated
end
```

1. 函数适用范围 `scope`: 可选，无或 `local`，`local` 表示本地函数，否则即为全局函数。
2. 函数名称 `function_name`: 这是函数的实际名称。函数名和参数列表一起构成了函数签名。尽量作到见名知义。
3. 参数 `argn`: 即形参，可选参数。**无类型修饰，本地变量，函数退出时自动销毁**，尽量作到见名知义。
4. 函数体 `body`: 方法主体包含了定义方法做什么的语句的集合。
5. 返回 `result`: 无此指定返回类型，在 Lua 可以返回多个返回值，借助 `return` 关键字，多个返回值用","隔开。

8.1.2.2. 示例

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

8.1.2.3. 调用

当创建一个 Lua 函数，给什么样的功能，必须做一个定义。要使用一个方法，将不得不调用该函数来执行定义的任务。

当程序调用一个函数，程序的控制转移到被调用的函数。被调用函数进行定义的任务和在执行它的 `return` 语句或当其功能的终端到达时，程序控制返回到主程序。

而调用只是需要传递所需的参数以及方法名的方法，如果方法返回一个值，那么你可以存储返回的值。例如：

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

-- calling a function
print("The maximum of the two numbers is ", max(10,4))
print("The maximum of the two numbers is ", max(5,6))
```

当我们运行上面的代码中，将得到下面的输出。

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

8.1.2.4. 参数与返回

函数对于实参与形参的对应，并不要求，同样对于接收返回值的变量个数也没有要求。基本规则是，有则接之，无则丢掉或是 nil。

```
function func(a, b, c, d)
    print("in func")
    print(a)
    print(b)
    print(c)
    print(d)
    print("out func")
    return a, b, c, d
end

local aa, bb, cc, dd

--aa,bb,cc,dd = func(1,2)
aa, bb, cc, dd, ee = func(1, 2, 3, 4, 5, 6)

print(aa)
print(bb)
print(cc)
print(dd)
print(ee)
```

8.2. 函数参数

8.2.1. 命名

lua 中的函数常规参数并无类型可言，书写时尽量做到见名知义即可。lua 的形参没有类型，往往要看其调用者，才知其类型。

8.2.2. 实参与形参

8.2.2.1. 赋值规则

从实参与到形参，本质也是赋值的行为，多传值的参数会被丢掉，未被赋值的参数会为置为 nil。

```
local test = function(a,b,c,d,e,f,g)
    print(a,b,c,d,e,f,g)
end

test(1,2,3,4,5,6,7,8,9,10)
test(1,2,3,4)
```

8.2.2.2. 默认参数

nil 不能参与算术和关系运算，通常作入参处理，也就是 lua 中常见的默认参数。采用 or 运算符来实现。

```
local test = function(a,b,c,d,e,f,g)
    a = a or 0
    b = b or 0
    c = c or 0
    d = d or 0
    e = e or 0
    f = f or 0
    g = g or 0
    print(a,b,c,d,e,f,g)
end

test(1,2,3,4,5,6,7,8,9,10)
test(1,2,3,4)
```

8.2.2.3. nil 入参

nil 值不能参与算术和关系运算，lua 中，仅有 false 和 nil 表示假，故能参与的仅为逻辑运算。

```
a = 10
-- print(a + nil)
-- print(a > nil)

print( true or nil )
print( true and nil )
print( not nil )
```

```
local max = function (a,b)
    if a > b then
        return a
    else
        return b
    end
```

```
end

local m = max(1,200)
print(m)
```

```
local m = max(1)
print(m)
attempt to compare nil with number
```

8.2.2.4. 入参检查

尽量采用默认参数的方式，来避免 nil 参与运算。

8.2.3. 省()调用

函数调用时，都需要使用一对圆括号把参数列表括起来。即使被调用的函数不需要参数，也需要一对空括号，对于这一规则，唯一的例外就是，当函数只有一个参数且该参数是字符串或表的构造器时，括号是可选的。

```
print "hello lua"    --print ("hello lua")
print [[hello lua]]  --print ([[hello lua]])
print {a = 1,b = 2} --print ({a = 1,b = 2})
type{}              --type({})
```

8.3. 函数返回

8.3.1. 多参返回

8.3.1.1. string.find

```
s,e = string.find("nzhsoft lua education","lua")
print(s,e)    -- 9 11
```

8.3.1.2. max data 与 index

```
function maximum(t)
    local mi = 1
    local d = t[mi]
    for i=1,#t do
        if t[i] > d then
            d = t[i]; mi = i
        end
    end
    return d,mi
end
print(maximum({1,5,9 ,4,33,0}))
```

8.3.1.3. 缺少 return 分支

C++中对于缺少分支的行为，给出的警告是这样的，warning: control reaches end of non-void function [-Wreturn-type]。

一旦走了，缺少分支，完成的赋值，结果是不确定的。

```
#include <stdio.h>

int func(int n )
{
    if (n >100)
        return -1;
}

int main()
{
    int m = func(1);
    printf("m = %d\n",m);
    return 0;
}
```

lua 中对于缺少分支的行为，完成的赋值，一律是 nil

```
local func = function ( n )
    if n >100 then return 111,999 end
end

local a,b = func(101)
print(a,b)           --111 999
local a,b = func(99)
print(a,b)           --nil nil
```

8.3.2. return 数量

8.3.2.1. 返回运算规则

lua 语言，根据函数的被调用情况调整回值的数量。当被当作一条单独语句时，其所有返回值都会被丢弃。当函数作为表达式，将只保留第一个返回值。

只有当函数调用是一系列表达式中最后一个(或唯一一个表达式)时，其所有的近回值才能被获取到。

```
function f123() return 1, 2, 3 end
function f456() return 4, 5, 6 end
print(f123())      -- prints 1, 2, 3
print(f456())      -- prints 4, 5, 6
print((f123()))   -- prints 1
print((f456()))   -- prints 4

print(f123(), f456()) -- prints 1, 4, 5, 6
print(f456(), f123()) -- prints 4, 1, 2, 3

x,y,z = f123(),99,999
print(x,y,z)

x,y,z,a =999,f456()
print(x,y,z,a)
```

```

t = {f456()}
for k,v in pairs(t) do
    print(k,v)
end

function f0() end
print("====")
-- t2 = {f456(),f123()}
t2 = {f456(),f123(),99}
for k,v in pairs(t2) do
    print(k,v)
end

function abc(a,b,c)
    print(a,b,c)
end

abc(f456())
abc(f456(),f123())

function func()
    -- return f123()
    -- return f123(),f456()
    -- return(f123())
    return f123()+f456()
end
print(func())

```

8.3.2.2. 位置

`return` 用于结束函数，返回参数，但必须是语句块中的最后一句，即 `return` 只能出现在语句块的结尾，或是 `end`, `else` 和 `until` 的前面

```

function foo()
    print(1)
    return math.pi
    print(2)
end
foo()

```

'end' expected (to close 'function' at line 1683) near 'print'

8.4. 函数是一等公民

8.4.1. first class(非语法糖)

lua 中的函数，严格遵循，词法定界(lexical scoping)的第一类值。

Lua 中所有函都是匿名的(anonymous)，比如，print 实际指的时保存该函数的变量，但是前面我们介绍的，不是传统函数的方式吗？有函数命名，入参，返回的概念。

请注意，前面介绍的，只是一种函数的语法糖形式。`function max(num1,num2) end`, 其本质是 `max = function (num1,num2) end`，这很好的体现了，匿名性和可赋值性。

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

第一类值，意味着 lua 语言中的函数与其它常见类型的值(数值和字符串)具有相同的权限，也就意味着，可以将函数存储在变量中(全局变量和局部变量中)或表中。也可以将某个函数作为参数传递给其它函数，还可以将某个函数作为其它函数的返回值。

8.4.2. 普通变量

函数作为 first-class 一等公民，可以像其它类型一样，赋给变量或是传参或作返回值。这一点，有点类似于 C 语言中的函数指针。

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

function add(num1, num2, functionPrint) --打印函数作为入参用于显示
    result = num1 + num2
    functionPrint(result)
end
myprint(10)
add(2, 5, myprint)
```

8.4.3. 表字段变量

赋值表内字段，在表内或是方式 1，是其本质行为，方式 2 也是一种语法糖行为，这样会让你看上去，更像是传统函数的定义。

```
t = {
    add = function (x, y) --表内
        return x + y;
    end
}
print(t.add(1, 2))

t.minus = function (x, y) --表外 1
    return x - y
end

print(t.minus(100, 50))

function t.mul(x, y) --表外 2
```

```
    return x * y  
end  
  
print(t.mul(100, 55))
```

表内字段，也可以是函数的形式，给面向对象，提供了基础模型，至少在形式达成了成员函数的样子。

9. 函数进阶

9.1. 变参

9.1.1. 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。

```
print("edu","nzhsoft","cn")      --变参函数
io.write("edu","nzhsoft","cn")
str = string.format("%5s %10d","a",100)    --变参函数
print(str)

function func(...)
    local a,b,c = ...   --直接赋值
    a = a or 0
    b = b or 0
    c = c or 0
    print("a=",a)
    print("b=",b)
    print("c=",c)
    print("size of ...",#{...})
end

func(1,2)
func(1,2,3,4,5)
```

9.1.2. 变参->表化{ ... }

可以在 Lua 中创建一个具有可变参数的函数，...作为它的参数。我们可以通过看一个例子，利用可变参数该函数将返回平均值。

```
function add( ... )
    local sum = 0
    local arg = {...}
    -- for i=1, #arg do
    --     sum = sum + arg[i]
    -- end
    for _,v in ipairs(arg) do
        sum = sum + v
    end
    return sum
end

print(add(1,2,3,4,5,6,7,8,9,10))
```

```
print(add(nil,1,2,3,4,5,6,nil,7,8,9))
```

9.1.3. 变参->table.pack(...)

表化的过程中，若表中有空洞 nil 的存在。则该表长度不可得，迭代遍历也会有中途停止。

table.pack(...)会返回一个{...}的表，但最后一个 n 域，记录的参数的个数。这样，再加一判断，就可以解决了表化后表中有 nil 值的问题。

```
function add(...)
    local arg = table.pack(...)
    local sum = 0
    for i = 1, arg.n do
        if arg[i] ~= nil then
            sum = sum + arg[i]
        end
    end
    return sum
end
print(add(1,2,3,4,5,6,7,8,9,10))
print(add(1,2,3,4,5,6,nil,7,8,9))
```

9.1.4. 变参之 select(...)

另一种遍历函数的可变长参数的方法是使用函数 select。函数 select 总是具有一个固定参数 n，以及数量可变的参数。如果能数为 n，那么函数总是返回第 n 个参数后的所有参数；若固定参数为#，则返回参数的总数。

If the selector is a number n, select returns all arguments after the n-th argument; otherwise, the selector should be the string "#", so that select returns the total number of extra arguments.

```
print(select(1, "a", "b", "c"))      --> a b c
print(select(2, "a", "b", "c"))      --> b c
print(select(3, "a", "b", "c"))      --> c
print(select("#", "a", "b", "c"))     --> 3
print(select("#", "a", nil, "b", nil, "c", nil)) --无惧 nil
```

更为常见的使用方式是，将 select 返回值作单参参与运算。

More often than not(往往，多半)，we use select in places where its number of results is adjusted to one, so we can think about select(n, ...) as returning its n-th extra argument.

```
function add ...
    for i = 1, select("#", ...) do
        print(select(i,...))
    end
end
print(add(1,2,3,4,5))
```

引例，将函数返值，参与运算。

```
function func()
```

```

    return 1, 2, 3, 4, 5
end
print (1 + func())

```

将 select 返回值，参与运算。

```

function average( ... )
    local sum = 0
    local len = select("#", ...)
    local nillen = 0
    for i=1,len do
        if select(i,...) then
            sum = sum +select(i,...)
        else
            nillen = nillen +1
        end
    end
    return sum /(len -nillen)
end

print(average(1,2,3,nil,4,5,6,nil,7,8,9,10))

```

少量参数时，select 的作法要比 table.pack(...)的作法效率要高，因为这样，避免了每次调用表创建的开销。对于大量参数，select 的开销，将超过表的创建。

9.1.5. 形参之固定参数

fmt 就是属于不变参部分，称为固定参数，固定参数可以有任意的数量，但是固定参数必须放在变长参数之前。

```

function write( fmt,... )
    return io.write(string.format(fmt,...))
end

write("%d %f %s",12,12.5,"Abc")

```

9.1.6. table.unpack({})

从语义来看，table.unpack 是 table.pack 的逆向函数，pack 完成将参数列表变为 lua 的表结构，而 unpack 将一个表结构转一个参数列表，其结果可用于一个函数的参数。

```

table.unpack (list [, i [, j]]) 返回列表中的元素。
这个函数等价于 return list[i], list[i+1], ..., list[j]
i 默认为 1 , j 默认为 #list。

```

```

print(table.unpack({10, 20, 30})) --> 10 20 30
a, b = table.unpack{10, 20, 30} --> a=10, b=20, 30 is discarded

```

unpack 的一个重要应用就是，可将参打包后传入函数。比如，可将原字符串，与需要查找的字符串，打包到表中传入。unpack 还支持传入，第 2 个和第 3 个参数。第 2 个参数表示起始，第 3 个参数表示数量。

```

print(string.find("hello", "ll"))

f = string.find
a = {"hello", "ll"}

print(f(table.unpack(a)))

print(table.unpack({"Sun", "Mon", "Tue", "Wed"}, 2, 4)) --范围

```

9.1.7. 命令行参数(...)

当拉起一个脚本文件时，脚本文件的名称，及其参数也可以通过...获得。本质：文件被拉起的过程同函数调用是一样的。

假设有文件，tst.lua，内容如下。现运行 lua53 tst.lua edu nzhosft cn

```

a, b ,c = ...
print(a,b,c)           -- edu nzhosft cn

```

9.2. 函数作入参--回调

9.2.1. table.sort

表标准库提供了，函数 table.sort，该函数以一个表为参数对其中的元素进行排序，这种函数必须提供排序的多样式，比如按升序或降序，或按表中的键等。

函数 sort 并没有试图穷尽所有的排序方式，而是的供了一个可选参数，也就 所谓的排序函数，即入参函数。这是一种聪明的选择。

9.2.2. sort array

```

array = {1,3,5,7,9,2,4,6,8,10}
table.sort(array,function (a,b)
    -- return a < b
    return a > b
end)

for _,v in ipairs(array) do
    print(v)
end

```

9.2.3. sort (hash table) array

```

network = {
    {name = "server1",ip = "192.168.1.110"},
    {name = "server3",ip = "192.168.1.110"},
    {name = "server2",ip = "192.168.1.110"},
    {name = "server4",ip = "192.168.1.110"}
}

function compare(a,b)
    return a.name < b.name

```

```

end

-- table.sort(network, function(a,b)
--     return a.name > b.name
-- end)

table.sort(network,compare)

for _,v in ipairs(network) do
    for name, ip in pairs(v) do
        print(name,ip)
    end
end

```

9.3. 函数作返回-Closer 闭包

9.3.1. 什么是闭包

闭包，简而言之就是函数内部定义的函数。严格意义上来说，还包括了他所能截获的外部变量。外部变量，即，外函数内的局部变量。

```

function func()
    local function foo()
        print("返回函数内部定义的函数")
    end
    return foo
end

-- function func()
--     return function ()
--         print("返回函数内部定义的函数")
--     end
-- end

f = func();
f()

```

9.3.2. 上值 upvalue

C++中的 lambda 就是一种闭包，通过[]来截获外部的非全局的局部变量。但是用法跟C++中区别比较大。在 C++中强调的是简短函数的就地书写，而在 lua 中强调的是截获非全局的局部变量返回。

此时返回的闭包，是包含有状态的。状态就是上值，即 upvalue。

upvalue 是一种变量，称为非局部变量(non-local variable)，是指不是在局部作用范围内定义的一个变量，但同时又不是一个全局变量，外部函数的局部变量，功能上有点类似于 c 语言中的 static 变量。

upvalue 的意义在于状态记录与共享。

```
function foo(n)
```

```

print("common function ",n)
end

foo(2018)
foo(2018)
foo(2018)

function func(n) --函数参数，是函数作用域内的 local 变量，也是一种上值
    local function foo()
        print("返回函数内部定义的函数", n)
        n = n + 1
    end
    return foo
end

f = func(2018);
f()
f()
f()
-- =====
f2 = func(2018);
f2()

```

上例中的，n 对于 func 来说，是个非局部变量，但又不是全局变量。n 的值在多次调用中实现了累加。

同时返回的多个闭包，**共享一个 upvalue**，其中一个修改会影响到其它，而不同次返回的闭包之间是相互**独立**的。

```

function func(n)
    local function foo1()
        print(n)
    end
    local function foo2()
        n = n + 10
    end
    return foo1,foo2
end

f1,f2 = func(2015)
f1() -- 打印 2015

f2()
f1() -- 打印 2025

f2()
f1() -- 打印 2035
=====
f3 = func(2000) --注意此时 f3 同 f1 和 f2 不再共享一个 upvalue
f3()

```

```
f2() --再次试图加 10
f3()
```

9.3.3. 应用

9.3.3.1. 求和

打印 $0+1+\dots+n$ 的和
 $0+\dots+93=4371$
 $0+\dots+94=4465$
 $0+\dots+95=4560$
 $0+\dots+96=4656$
 $0+\dots+97=4753$
 $0+\dots+98=4851$
 $0+\dots+99=4950$
 $0+\dots+100=5050$

实现

```
local function getSum()
    local n= 0
    return function(i)
        n = n + i
        return n
    end
end

local sum  = getSum()

for i=1,100 do
    print(string.format("0 +.. + %d = %d",i,sum(i)))
end
```

9.3.3.2. 斐波那契数列

斐波那契数列指的是这样一个数列 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368.....

这个数列从第 3 项开始，每一项都等于前两项之和。

实现：

```
-- 0 1 1 2 3 5 8 11
-- a b
local function getFibonacci()
    local a,b = 0,1
    return function ()
        b ,a = a + b, b
        return a
    end
end
-- 0 1 1 2 3 5 8 11
```

```
-- a b
-- 0 1 1 2 3 5 8 11
-- a b
-- 0 1 1 2 3 5 8 11
-- a b

local f = getFibonacci()

for i = 1,50 do
    print(f())
end
```

9.3.3.3. 迭代

闭包最常用的一个应用就是实现迭代器。所谓迭代器就是一种可以遍历一种集合中所谓元素的机制。

每个迭代器都需要在每次成功调用之间保持一些状态，这样才能知道它所在的位置及如何迭代到下一个位置。在 C++ 模板库 STL 中每一种容器均有自实现的迭代器。通过 `++itr` 从 `begin` 到 `end` 的过度，实现成员的遍历。

```
vector<int> vi={1,2,3}
vector<int>::iterator itr= vi.begin(); --获取迭代器①
for(; itr != vi.end(); ++itr)           --调用迭代器②
{
    cout<<*itr<<endl;                 --获取内容 ③
}
```

闭包刚好适合这种场景。比如下面的代码：

```
array = {10,20,30,40,50,60,70}

function getItr(tab)
    local i = 0           --upvalue
    return function()
        i = i + 1
        return tab[i]
    end
end

itr = getItr(array)      --获取带有状态的闭包①

while true do
    local item = itr()  --调用闭包函数②③
    if item == nil then break end
    print(item)
```

```
end
```

9.4. ipairs 初试

9.4.1. 自实现

学习了闭包与上值的概念以后，可以自实现 ipairs

```
tab = {"lua", "nzhsoft"}  
  
function traverse(collection)  
    local index = 0  
    local count = #collection  
    return function()           -- 闭包函数  
        index = index + 1  
        if index <= count then  
            return index, collection[index] -- 返回迭代器的当前元素  
        end  
    end  
end  
  
-- for k,v in ipairs(tab) do  
--     print(k,v)  
-- end  
  
for k, v in traverse(tab) do  
    print(k, v)  
end
```

9.4.2. 虚变量

当我们在自实现 ipairs 时，可以返回一个值。常规 for 中的 kv 可以只写一个，或写一个只能打印索引。比如上面的示例可以写为：

```
array = {"Lua", "Tutorial"}  
  
for v in ipairs(array)do  
    print(v)  
end
```

而对于 ipairs，是由系统提供，必须返回两个值，而有时 k 变量并不我们所需要的，可以用虚变量(_)来占位。

```
for _, v in ipairs(array)do  
    print(v)  
end
```

```

function iterator(collection)
    local index = 0
    local count = #collection
    -- 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

-- iterator

for v in iterator(array) do
    print(v)
end

```

9.5. 递归

函数自身调用自己或是间接调用自己的行为，称为递归。

9.5.1. 求阶乘

9.5.1.1. n!

```

local function fact(n)
    if n == 0 then return 1
    else return n * fact(n-1)
    end
end

print(fact(5))

```

上述代码是没有问题的，但是采用了语法糖的行式。可不可以用更本质的写法来重写上述代码呢？

9.5.1.2. 问题引出

```

local fact = function (n )
    if n == 0 then return 1
    else return n * fact(n-1)
    end
end

print(fact(5))
--attempt to call a nil value (global 'fact')

```

9.5.2. 解析

当lua语言，编译 fact(n-1)时，局部的 fact 尚未定义完毕，因此这个表达式会尝试调用 global fact 而非局部 fact。可以通过先定义局部变量，再定义函数的方式来解决这个问题。

```

local fact
fact = function (n)
    if n == 0 then return 1
    else return n * fact(n-1)
end
print(fact(5))

```

9.5.3. 语法糖的展开

lua 语言展开局部函数的语法糖时，使用的并不是基本的函数定义，相反形如：

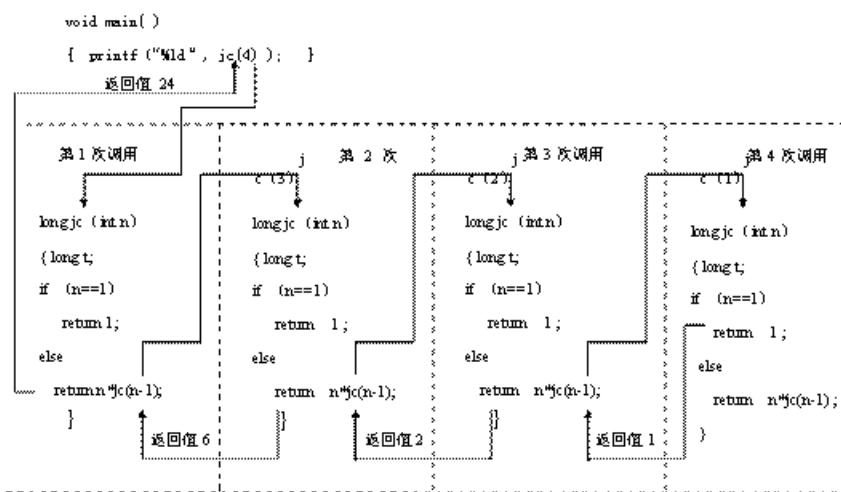
```
local function foo(params) body end
```

的定义，会被展开为，

```
local foo; foo = function(params) body end
```

9.5.4. 递归尾调用

9.5.4.1. 递归



9.5.4.2. 什么是尾调用

尾调用(tail call)，是被当作函数调用使用的跳转，当一个函数的最后一个动作是调用另外一个函数而没有再进行其它的工作时，就形成了尾调用。

如下代码就够成了尾调用：

```
function f(x) x = x + 1; return g(x) end
```

当函数 f 调用完 g 之后，f 不再需要进行其它的工作。这样被调用的函数执行结束后，程序不再需要返回最初的调用者。这样在尾调用之后，程序就不再需要在调用栈中保存有关调用函数的任何信息。当 g 调用完毕，程序的执行路径会直接返回到调用 f 的位置。

Lua 语言解释器，就利用这一点，使得在进行尾调用时，不使用任何额外的栈空间，将这种现象称为尾调用消除。由于尾调用不会使用栈空间，所以一个程序中能够嵌套的尾调用是多限的。

9.5.4.3. 尾递归

递归，最大的问题是，容易造成栈空间溢出，而如下的递归则不会造成溢出，因为构成了尾递归。

```
function foo(n)
    if n > 0 then return foo(n-1) end
end
foo(10000)
```

9.6. 练习

10. Iterator 迭代器

10.1. 概念

迭代器是一种结构，使能够遍历所谓的集合或容器中的元素。在 Lua 中，这些集合通常是指，那些用于创建各种数据结构的 Table。

10.1.1. 引入

```
array = {"lua", "nzhsoft"}
for key,value in ipairs(array) do
    print(key, value)
end
```

上面的示例使用由 Lua 中提供的默认 ipairs 迭代函数。

10.1.2. 分类

在 Lua 中，我们使用函数来表示迭代器。基于这些迭代器的功能状态保持，有两种主要类型：无状态的迭代器，有状态迭代器。

10.1.3. Generic for

10.1.3.1. 泛型 for 与其等价关系

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

```
for var_1,...,var_n in explist do block end
do
    local _f,_s,_v = explist
    while true do
        local var_1,... ,var_n = _f(_s,_v)
        if var_1 == nil then break end
        _v = var_1

        -----
        block
        -----

    end
end
```

- explist 只会被计算一次。它返回三个值，一个 迭代器 函数_f，一个 状态_s，一个 迭代器的初始值_v。
- f, s, 与 v 都是不可见的变量。这里给它们起的名字都只是为了解说方便。
- 你可以使用 break 来跳出 for 循环。

- 环变量 var_i 对于循环来说是一个局部变量；你不能在 for 循环结束后继续使用。如果你需要保留这些值，那么就在循环跳出或结束前赋值到别的变量里去。

10.1.3.2. ipairs 与其等价关系

```
for k, v in ipairs(t) do print(k, v) end
```

```
for k, v in ipairs do block end
do
    local _f, _s, _v = ipairs
    while true do
        local k, v = _f(_s, _v)
        if _var == nil then break end
        _v = k

        block

    end
end
```

10.2. 无状态的迭代器

由名字本身就可以明白，这类型的迭代器功能不保留任何状态。

现在让我们来看看使用打印 n 个数的平方的简单的函数，来创建我们自己的迭代器的例子。

10.2.1. 引入 square, 3, 0

```
function square(mx, ins)
    if ins < mx then
        ins = ins + 1
        return ins, ins * ins
    end
end

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

```
do
    local _f,_s,_v = square,3,0
    while true do
        local i,n = _f(_s,_v)
        if _v == nil then break end
        _v = i
```

```

        print(i,n)
    end
end

```

10.2.2. 引入 square, 3, 0 -> for squareS(3)

square, 3, 0 如何改的更像是 ipairs(3)呢？

```

function square(mx, ins)
    if ins < mx then
        ins = ins + 1
        return ins, ins * ins
    end
end

function squareS(mx)
    return square, mx, 0
end

do
    local _f,_s,_v = squareS(3)
    while true do
        local i,n = _f(_s,_v)
        if _v == nil then break end
        _v = i
        print(i,n)
    end
end

```

```

for i,n in squareS(6) do
    print(i,n)
end

```

10.2.3. 无状态 ipairs

迭代的状态包括被遍历的表（循环过程中不会改变的状态常量）和当前的索引下标（控制变量），myIpairs 和迭代函数 myIter 都很简单。

```

arr = {10, 20, 30, nil, 40, 50, 60}

-- for k, v in ipairs(arr) do
--     print(k, v)
-- end

function iterator(arr, i)
    i = i + 1

```

```

local v = arr[i]
if v then
    return i, v
end
end

function traverse(arr)
    return iterator, arr, 0
end

for k, v in traverse(arr) do
    print(k, v)
end

```

10.3. 有状态迭代器

10.3.1. 闭包引入

继续改进上一个示例：

```

arr = {10, 20, 30, nil, 40, 50, 60}
function iterator(arr)
    local i = 0
    return function()
        i = i + 1
        if arr[i] then return i, arr[i] end
    end
end

for k, v in iterator(arr) do
    print(k, v)
end

```

```

for k, v in ipairs do block end
do
    local _f, _s, _v = ipairs
    while true do
        local k, v = _f(_s, _v)
        if _var == nil then break end
        _v = k

        block
    end

```

```
end
```

此时的 `traverse` 只返回了一个参数，赋给了 `_f`，即 `iterator` 的中的闭包函数，也就是 `_s` 和 `_v` 值此时均是 `nil`。再次调用迭代函数时传入的 `_s`, `_v` 均是 `nil`，即便第二次，传入更新后的 `_v`，由于迭代函数无接收参数，也没有了意义。因为此时的 `_s, _v` 均以 `upvalue` 的形式存在于闭包中，所以此时的迭代没有参数，也不再需要参数。

10.3.2. 有状态 ipairs

再次改进，加一些判断：

```
function iterator( arr )
    local idx = 0
    local count = #arr
    return function( )
        idx = idx + 1
        if idx <= count then
            return idx, arr[ idx ]
        end
    end
end

function traverse( arr )          --是可有可无的
    return iterator( arr )
end

do
    local _f, _s, _v = traverse( arr )
    while true do
        local k, v = _f( _s, _v )
        _v = k
        if k == nil then break end
        print( k, v )
    end
end
```

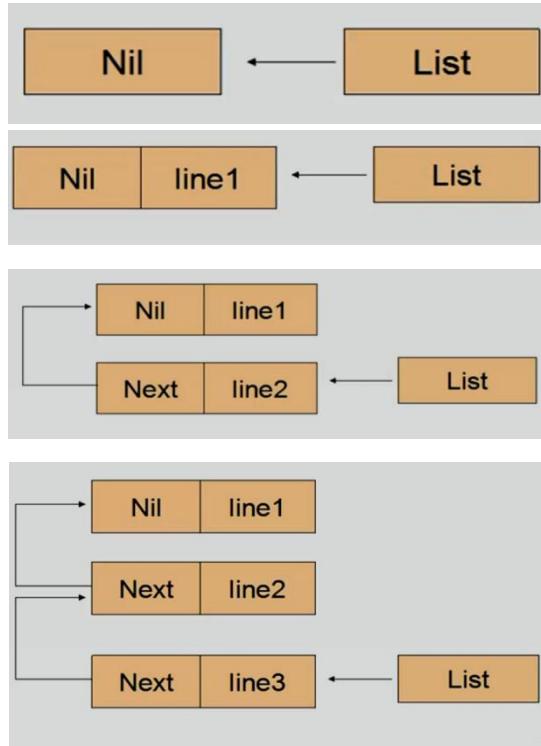
10.4. 有状态 vs 无状态

尽可能的尝试编写无状态的迭代器，无状态的迭代器将所有的状态都保存在泛型 `for` 中，不需要在开始循环时创建任何新的对象。如果无法使用无状态迭代器，那就就使用闭包函数创建迭代器。

10.5. 练习

10.5.1. 设计链表，并设计其迭代函数

10.5.1.1. 图示



10.5.1.2. 链表的创建与遍历

```

arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

head = nil

local i = 1
while true do
    if arr[ i ] then
        head = { value = arr[ i ], next = head }
        i = i + 1
    else
        break;
    end
end

-- while head do
--     print(head.value)
--     head = head.next
-- end

```

```
-- print(head.value,head.next.value,head.next.next.value)
```

10.5.2. 迭代

10.5.2.1. 无状态

```
function iterator( head, next )
    if next == nil then
        return head
    else
        return next.next
    end
end

function traverse( head )
    return iterator, head, nil
end

for v in traverse( head ) do
    print( v.value )
end
```

10.5.2.2. 有状态

```
arr = {11,22,33,44,55,66}

head = nil
local i = 1
while true do
    if arr[i] then
        head = {value=arr[i],next = head}
        i = i +1
    else
        break
    end
end

function traverse(head)
    local next = head
    return function ()
        head = next
        if head then
            next = head.next
            return head
        end
    end
end

for v in traverse(head) do
```

```
    print(v.value)
end
```

11. 模式匹配 Pattern

Lua 语言，不支持 POSIX 正则表达式，也没有使用 Perl 正则表达式来进行模式匹配 (pattern matching)。Lua 提供了自实现版本的模式匹配，仅有 600 行代码实现，而要实现 POSIX 正则则需要 4000 行代码。

11.1. 匹配函数

字符串标准库提供了基于模式(pattern)的 4 个函数。我们已经初步了解了函数 find 和 gsub，其余的两个函数是 match 和 gmatch(Global Match)。

11.1.1. string.find

11.1.1.1. 释义

```
string.find (s, pattern [, init [, plain]])
```

查找第一个字符串 s 中匹配到的 pattern。如果找到一个匹配，find 会返回 s 中关于它起始及终点位置的索引；否则，返回 nil。第三个可选数字参数 init 指明从哪里开始搜索；默认值为 1，同时可以是负值。第四个可选参数 plain 为 true 时，关闭模式匹配机制。此时函数仅做直接的“查找子串”的操作，而 pattern 中没有字符被看作魔法字符。注意，如果给定了 plain，就必须写上 init。

如果在模式中定义了捕获，捕获到的若干值也会在两个索引之后返回。

11.1.1.2. 测试

```
s = "hello world"
i,j = string.find(s,"hello")
print(i,j)
print(string.sub(s,i,j))
print(string.find(s,"world"))
i,j = string.find(s,"l")
print(i,j)
print(string.find(s,"llll"))
```

如上，string.find 的简单搜索 plain search，所谓简单搜索就是进行单纯的查找子字符串。若字符串中，有特殊含义的字符，比如 [，可能会导致搜索失败。

```
string.find("a[word]", "[") -- malformed pattern (missing '[')
```

此时就用到，string.find 的可选参数，第 3 和第 4 个参数，第 3 个参数用于说明搜索的起始位置，第 4 个参数用于说明是否是简单搜索。需要注意的是，如果没有第 3 个参数，是不能传入第 4 个参数的。

```
print(string.find("a[word]", "[", 1, true)) -- 2 2
```

实战：记录换行位置与下标

```
s = [[
abc
efgg
xyz
```

```
]]
local t = {}
local i = 0
while true do
    i = string.find(s, "\n", i+1)
    if i == nil then break end
    t[#t+1] = i
end

for k,v in ipairs(t) do
    print(k,v)
end
```

11.1.2. string.match

11.1.2.1. 释义

简单搜索同 string.find，返回的是查找到的子串，而非起始结束位置，但是他的用处并不在此，其真正的用途在于与模式匹配结合。

```
print(string.match("hello world","wor")) -- wor
```

11.1.2.2. 测试

```
date = "Today is 17/7/1990"
d = string.match(date,"%d+/%d+/%d+")
print(d) -- 17/7/1990
```

11.1.3. string.gsub(stitute)

11.1.3.1. 释义

string.gsub(s, pat, repl [, n])

gsub 这个名字来源于 Global SUBstitution，有 3 个必选参数，分别是：目标字符串，模式，替换字符串，其基本用法是，将目标字符串中所有出现模式的地方，换成替换字符串。第 4 个参数用于限制替换的次数，并将次数返回。

如果 repl 是张表，每次匹配时都会用第一个捕获物作为键去查这张表。

如果 repl 是个函数，则在每次匹配发生时都会调用这个函数。所有捕获到的子串依次作为参数传入。

任何情况下，模板中没有设定捕获都看成是捕获整个模板。

如果表的查询结果或函数的返回结果是一个字符串或是个数字，都将其作为替换用串；而在返回 false 或 nil 时不作替换（即保留匹配前的原始串）。

11.1.3.2. 测试

```
s = string.gsub("nzhosft is cute","cute","great")
print(s) --nzhosft is great
s = string.gsub("all lii","l","x")
```

```
print(s) --axx xii

s = string.gsub("nzhosft is great","abc","xyz")
print(s)
```

```
s,n= string.gsub("all lii","l","x",2)
print(s,n) --axx lii
```

11.1.4. string.gmatch

11.1.4.1. 释义

`string.gmatch (s, pattern)`

返回一个迭代器函数，每一次调用这个函数，返回一个在字符串 s 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。

11.1.4.2. 测试

```
local str = "12china34,is,56great"
local itr_func = string.gmatch(str, "%d+") --%d 表示一个数字,+表示若干个

-- 查看 func_itor 类型
print("itr_func is", itr_func) --itr_func is function: 00722970
-- 第一次调用函数 itr_func
print("itr_func ret is ", itr_func()) --itr_func ret is      12
-- 再次调用函数 itr_func
print("itr_func ret is ", itr_func()) --itr_func ret is      34
-- 再次调用函数 itr_func
print("itr_func ret is ", itr_func()) --itr_func ret is      56
-- 再次调用函数 itr_func
print("itr_func ret is ", itr_func()) --itr_func ret is

for v in string.gmatch(str, "%d+") do
    print(v) -- 12 34 56
end
```

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

```
-- 查找属性对 --%w 代表一个数字/字线 +代表若干
local attrstr = "nb=nzshoft, nb=nzshoft, name(lua"
print("\noutput attr pair capture using loop:")
t = {}
for k,v in string.gmatch(attrstr, "(%w+)=(%w+)") do
    print(k, v)
```

```
t[k] = v
end
```

11.2. 模式 pattern

11.2.1. 任一字符

Lua 中使用百分号%作为转义字符(h 此类于 c 中的 printf)，所有被转义的字符都具有特殊的含义。而所有被转义的非字符代表其本身，比如%%，则表示其本身%。

11.2.1.1. 单字符

补转义的字符，仅表示“任意”一个字符。

转义字符	语义	补集
%a	代表所有字母	%A
%c	代表所有控制字符	%C
%d	代表所有数字(0-9)	%D
%l	代表所有小写字母	%L
%u	代表所有大写字母	%U
%s	代表所有空白字符	%S
%p	代表所有标点字符	%P
%g	代表除了空格外的所有可打印字符	%G
%w	代表所有包含字母或者数子的字符	%W
%x	表示所有 16 进制数字符号	

如下 gsub，依据模式串%a，将所有字符替换为.，而%A，则将所有非字符替换为.，注意 gsub 反回第二个参数为替换的次数，可以加0，将其省略。

```
print(string.gsub("hello ,up-down","%a",'.'))  
print(string.gsub("hello ,up-down","%A",'.'))
```

11.2.1.2. 字符集

可以使用字符集，来创建自定义的字符分类，只需要在方括号内，将单个字符和字符分组合起来即可。表示“任意”一个字符。

charset	示例
[set]	<ul style="list-style-type: none"> [135] 匹配，含有 1 或 3 或 5 的文本 [a-z] 匹配，含有 abcdef....xyz 中任意一个字符的文本 [1-9] 匹配，含有 12...9 中任意一个数字的文本 [%w_] 匹配，所有以下划线结尾的字母和数字。 [%[%]] 匹配，方括号 %d == [0-9] %w == [0-9a-fA-F]
[^set]	<ul style="list-style-type: none"> 得到字符集的补集 [^0-7] 代表所有 8 进制以外的字符 [^\n] 代表除换行符以外的其它字符 [^\s] == %S

如下，字符串中分别用_替换了 135，用空格，替换了 AEIOUaeiou

```
text = [[
ab1cd2efg3xyz5
---AEIOUaeiou---
]]

print(string.gsub(text, "[135]", "_"))
print(string.gsub(text, "[AEIOUaeiou]", " "))
```

11.2.1.3. 魔法字符

字符	转义	示例	本义
%	用于转义	%a	%%
.	表示任意一个字符	a. 匹配 ax ab ay	%.
[]	字符集		%[%]
()	捕获		%(%)

以下四个魔法字符，称为可选修饰符 modifier

+	重复前一项 1 次或多次，尽可能多	ax+ 匹配 ax axxx	%+
*	重复前一项 0 次或多次，尽可能多	ax* 匹配 a ax axxx	%*
-	重复前一项 0 次或多次，尽可能少	ax- 匹配 a ax axxx	%-
?	重复前一项 0 次或 1 次	ax? 匹配 a ax	%?

修饰符+，用于匹配字符分类中一个或多个字符，它总是能获得与模式相匹配的最长序列，如下%a+，代表一个或多个字母(即一个单词)。%d+代表一个数字。

```
text = [[
one, and two; and three
1,22,333,555,abcd
]]
-- print(string.gsub(text,"%a+","nzhosft"))

for v in string.gmatch(text,"%d+") do
    print(v)
end
```

修饰符*类似于+号，但是接受对应字符出现零次的情况。该修饰符一个典型的用法就是在模式的部分之间匹配可选的空格。比如，匹配() () ()

```
text = [[
()
()
()
]]
]]
```

```
print(string.gsub(text, "%(%s*%)", "{}"))
```

%(%s*%), 蓝色代表(的本意, 红色代表 0 个或多个空白字符。

修饰符-, 则会尽可能少的匹配, 再来一个案例: 删除掉 c 语言中所有的注释, `/**/ -> /*.*/* -> /*%.%0*/`, 如下代码:

```
c = "int x; /*x*/ int y; /*y*/"
print((string.gsub(c, /*.*.%0*/, ""))) --int x;
```

此时我们会发现, `.`会尽可能长的匹配, 因此, 程序中第一个`/*`和只会和最后一个`*/`相匹配。

而如果, 采用`-`则会尽可能少的匹配, 先找到第一个`/*`匹配。

```
c = "int x; /*x*/ int y; /*y*/"
print((string.gsub(c, /*.-%0*/, ""))) --int x; int y;
```

修饰符? 用于匹配一个**可选**的字符, 即有或无的选择, 假设我们想在文本中, 寻找一个整数, 这个整数包含一个可选的正负字符, 那么就可以使用模式, `[+-]?%d+`来完整表达这个需求。

```
digit = [
123xxxx+456xxx-111
]
for v in string.gmatch(digit, "[+-]?%d+") do
    print(v)
end
```

另外,

模式**%b**, 匹配成对的字符串, `%bxy` 表示 x 为起始字符, y 为结束字符。模式**%b()**匹配左括号开始, 并以右括号结束的子串。

```
s = "a (enclosed (in) parentheses ) line"
print((string.gsub(s, "%b()", ""))) --a line
```

常用的还有, `%b() %b[] %b{} %b<>`, 其实可以是任意不同的字符作为其分隔符之用。

11.3. 捕获

捕获(capture), 机制允许根据一个模式从目标字符串中抽出与该模式匹配的内容来用于后续用途, 具体操作, 把需要捕获的内容用一对圆括号括起来指定捕获。

11.3.1. 匹配分割

利用 `string.match` 会将捕获到的值作为单独的结果返回, 换言之, 匹配分割返回。`%a+` 表示一个非空的字母序列, `%s*`表示一个可能为空的空白序列。

```
pair = "org = nzhosft"
```

```
k,v = string.match(pair,"(%a+)%s*=%s*(%a+)")
print(k,v) --org nzhosft
```

模式中%a+被放到两个圆括号中，因此在匹配时就能捕获它们。

```
date = "Today is 17/7/1990"
d,m,y = string.match(date,"(%d+)/(%d+)/(%d+)")
print(d,m,y) --print(d,m,y)
```

11.3.2. 匹配复用

在模式中，形如%on(其中n是一个数字)，表示匹配第n个捕获的副本。举一个典型的例子，假设在一个字符串中寻找单引号或双引号括起来的子串，可以尝试使用[""].-[""]，但是在这种情形下，遇到，“It's all right”就会遇到问题。

```
s = [[then he said: "It's all right"]]
print(string.match(s,"([\"'].-[\\"'])")) --"It'
```

要解决这个问题，可以捕获第一个引号，用它来指明第二个引号，第一个捕获的是引号本身，第二个捕获的是引号中的内容。

```
s = [[then he said: "It's all right"]]

q,qc = string.match(s,"([\"])(.-)%1")
print(q,qc) --" It's all right
```

另外一个典型的例子是，匹配lua中的长字符串：

```
s = "a = [=[[[ something ]]]==] ]=]; print(a)"
p = "%[=(*)%[(.*)%]%"1%"

print(string.match(s,p)) --= [[ something ]] ==]
```

11.3.3. gsub 中替换部分

将字符串s中，所有的（或是在n给出时的前n个）pattern都替换成repl，并返回其副本。repl可以是字符串、表、或函数。gsub还会在第二个返回值返回一共发生了多少次匹配。

11.3.3.1. 如果 repl 是一个字符串

如果repl是一个字符串，那么把这个字符串作为替换品。字符%是一个转义符：repl中的所有形式为%d的串表示第d个捕获到的子串，d可以是1到9。串%0表示整个匹配。串%%表示单个%。

```
x = string.gsub("hello world", "(%w+)", "%1 %1")
print(x) --> x="hello hello world world"

x = string.gsub("hello world", "%w+", "%0 %0")
print(x) --> x="hello hello world"
x = string.gsub("hello world", "%w+", "%0 %0", 1)
print(x) --> x="hello hello world"

x = string.gsub("hello world from Lua", "(%w+)%s*(%w+)", "%2 %1")
print(x) --> x="world hello Lua from"
```

去除字符串，两端的空格，`^$`保证了，我们可以获得整个字符串，中间的`.-`只会匹配尽可能省的内容，所以两个`%s*`便可匹配到首尾两端的空格。

```
function trim(s)
    s = (string.gsub(s, "^%s*(.-)%s*$", "%1"))
    return s
end
s='\t a bc d '
print(trim(s)) --输出： a bc d, 开头的 \t, 结尾的空格 都被 trim 掉了
```

11.3.3.2. 如果 `repl` 是张表

如果 `repl` 是张表，每次匹配时都会用第一个捕获物作为键去查这张表。

```
local t = {name="lua", version="5.3"}
x = string.gsub("$name-$version.tar.gz", "%$(%w+)", t)
print(x) --> x="lua-5.3.tar.gz"
```

11.3.3.3. 如果 `repl` 是个函数

如果 `repl` 是个函数，则在每次匹配发生时都会调用这个函数。所有捕获到的子串依次作为参数传入。

```
x = string.gsub("4+5 = $return 4+5$", "%$(-)%$",
                 function (s)
                     return load(s)()
                 end)
print(x) --> x="4+5 = 9"
print(load("return 4+5")())
```

```
x = string.gsub("PATH = $PATH, os = $OS", "%$(%w+)", os.getenv)
print(x) --> x="PATH = C:\Python27\;, os = Windows_NT"
```

任何情况下，模板中没有设定捕获都看成是捕获整个模板。

如果表的查询结果或函数的返回结果是一个字符串或是个数字，都将其作为替换用串；而在返回 `false` 或 `nil` 时不作替换（即保留匹配前的原始串）。

11.4. 正则 regex

正则简言之，匹配文本的式子。匹配以行为单位。

11.4.1. 规则

11.4.1.1. 起始与结束

11.4.1.2. 任意一字符

`[.]` \. 仅表示一个点

11.4.1.3. 匹配次数? +*

11.4.1.4. 创建子串

11.4.1.5. |与\

11.4.1.6. 匹配次数{n,m}

11.4.1.7. 元字符

正则	描述	示例
^	行起始标记	^tux 匹配以 tux 起始的行
\$	行尾标记	tux\$ 匹配以 tux 结尾的行
.	匹配任意一个字符	Hack. 匹配 Hackl 和 Hacki,但是不匹配 Hackl2 和 Hackil
[]	匹配包含在[]中的任意一个字符	coo[kl] 匹配 cook 和 cool
[^]	匹配除[]中字符之外的任意一个字符	9[^01] 匹配 92 93 96 但是不能匹配 90 91 ^[^a-zA-Z] 以非小写字母起始的行。 ^[^a-zA-Z] 匹配不用字母开始的行
[-]	匹配[]中指定范围内的任意一个字符	[a-z] 匹配 a-z 中的任意一个字符, [1-5] 匹配 1-5 任意一个数字
?	匹配之前项的 0 次或 1 次 等价于{0, 1}	colou?r 匹配 color 或是 colour, 但不能是 colourur
+	匹配之前项的 1 次或多次 等价于{1, }	number-9? 匹配 number-9 number99 但不能是 number-
*	匹配之前项的 0 次或多次 等价于{0, }	co*I 匹配 col cool cooooool
()	创建一个用于匹配的子串	ma(tri)?x 匹配 max 或是 matrix
{n}	匹配之前项的 n 次	[0-9]{3} 匹配任意一个三位数, 也可以表示为 [0-9][0-9][0-9]
{n,}	之前的项, 至少匹配 n 次	[0-9]{2,} 匹配任意一个两位或是更多位的数
{n,m}	指定之前的项目所必需匹配的最小次数和最大次数。	[0-9]{2,5} 匹配从 2 位数到 5 位数之间的任意一个数字
	或 匹配 两边的任意一项	oct(1st 2nd) 匹配 oct1st 或是 oct2nd
\	转义符可以将上面介绍的字符进行转义	a\b 匹配 a.b, 但不能匹配 axb, 通过转义不再具有特殊含义, 而仅表示.
\d	匹配一个数字字符。	等价于 [0-9]。
\D	匹配一个非数字字符。	等价于 [^0-9]。
\w	匹配字母、数字、下划线。	等价于 '[A-Za-z0-9_]'。
\W	匹配非字母、数字、下划线。	等价于 '[^A-Za-z0-9_]'。
\s	匹配任何空白字符, 包括空格、制表符、换页符等等。	等价于 [\f\n\r\t\v]。
\S	匹配任何非空白字符。	等价于 [^\f\n\r\t\v]。

{n} {n,} {n,m} 常与定界符, 一起使用, 用于避免重复的数据匹配。

11.4.2. 常见匹配

11.4.2.1. 数字

- 数字: `^[0-9]*$`
- n 位的数字: `^\d{n}$`
- 至少 n 位的数字: `^\d{n,}$`
- m-n 位的数字: `^\d{m,n}$`
- 零和非零开头的数字: `^(0|[1-9][0-9]*)$`
- 非零开头的最多带两位小数的数字: `^([1-9][0-9]*)+([.][0-9]{1,2})?${1,2}`
- 带 1-2 位小数的正数或负数: `^(+-)\d{1,2}(\.\d{1,2})?${1,2}`
- 正数、负数、和小数: `^(+-|.)?\d{1,2}(\.\d{1,2})?${1,2}`
- 有两位小数的正实数: `^([0-9]+([.][0-9]{2}))?${1,2}`
- 有 1~3 位小数的正实数: `^([0-9]+([.][0-9]{1,3}))?${1,3}`

11.4.2.2. 字符

- 英文和数字: `^[A-Za-z0-9]+$ 或 ^[A-Za-z0-9]{4,40}$`
- 长度为 3-20 的所有字符: `^.{3,20}$`
- 由 26 个英文字母组成的字符串: `^[A-Za-z]+$`
- 由 26 个大写英文字母组成的字符串: `^[A-Z]+$`
- 由 26 个小写英文字母组成的字符串: `^[a-z]+$`
- 由数字和 26 个英文字母组成的字符串: `^[A-Za-z0-9]+$`
- 由数字、26 个英文字母或者下划线组成的字符串: `^\w+$ 或 ^\w{3,20}$`

11.4.3. 示例

11.4.3.1. qq/ip

qq 号 4-10 位数字

- `^[0-9]{4,-10}$`
- `[0-9]\{1,3\}.[0-9]\{1,3\}.[0-9]\{1,3\}.[0-9]\{1,3\}`

11.4.3.2. 帐号与密码

帐号是否合法(字母开头，允许 5-16 字节，允许字母数字下划线)，密码(以字母开头，长度在 6~18 之间，只能包含字母、数字和下划线)。

- `^[a-zA-Z][a-zA-Z0-9_]{4,15}$`
- `^[a-zA-Z]\w{5,17}$`

11.4.3.3. id

身份证号(15 位、18 位数字)，最后一位是校验位，可能为数字或字符 X

- `(^\d{15}$)|(^\d{18}$)|(^\d{17}(\d|X|x)$)`

11. 4. 3. 4. email

```
● ^\w+([-.\w+]*)@\w+([-.\w+]*)\.\w+([-.\w+]*)$
```

11. 4. 3. 5. 日期

- 日期格式: ^\d{4}-\d{1,2}-\d{1,2}
- 一年的 12 个月(01~09 和 1~12): ^(0?[1-9]|1[0-2])\$
- 一个月的 31 天(01~09 和 1~31): ^((0?[1-9])|((1|2)[0-9])|30|31)\$

12. Meta 元表与元方法

12.1. 元表与元方法

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。

比如，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，之后检查元表中是否有一个叫“_add”的字段，若找到，则调用对应的值。“_add”等字段，其对应的值(往往是一个函数或是 table)就是“元方法”。

lua 中的此举，可类比于 C/C++ 中的运符符重载，让自定义类型，也可以实现基本数据类型的行为。

12.2. 元表

12.2.1. 概述

有两个很重要的函数来处理元表，设置元表，会返回本表的引用，获取元表则会得到本表的元表。

元表的主要用途就是在元表中写元方法。

<code>setmetatable(table, metatable):</code>	对指定 table 设置元表(metatable)
<code>getmetatable(table):</code>	返回对象的元表(metatable)。

12.2.2. 示例

```

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

t1 = {}; t2 = {}
print("t1:", t1)
print("t2:", t2)
print("set", setmetatable(t1, t2))          -- 普通表， t2 元表
print("get", getmetatable(t1))
print(setmetatable({}, {}))

```

12.3. 算术/关系元方法

元方法，即元表中特定的方法。这些方法，均为系统指定，双下划线开头，有特定语境下的特殊意义。

12.3.1. 算术类元方法

12.3.1.1. __add

现在两个表 a 和表 b，要实现两表相加，该如何作到呢？

```
a = { 1, 2, 3 }
```

```

b = { 4, 5, 6 }

c = a + b
for k, v in ipairs( c ) do
    print( k, v )
end

```

采用元表中重写`__add`元方法，即可实现

```

a = { 1, 2, 3 }
b = { 4, 5, 6 }

mt = { }
function mt.__add( a, b )
    local n = #a
    local ret = { }
    for i = 1, n do
        ret[ i ] = a[ i ] + b[ i ]
    end
    return ret
end

setmetatable( a, mt )
setmetatable( b, mt )

c = a + b

for k, v in ipairs( c ) do
    print( k, v )
end

```

12.3.2. 关系类元方法

12.3.2.1. `__eq`

假设现在有如下，表 `a` 和表 `b`，我们知道表是引用类型，所有即便是表 `a` 和表 `b` 中的内容完全相同，亦不可用 `==` 比较。

```

a = {x = 1, y = 2, z = 3}
b = {x = 1, y = 2, z = 3}

if a == b then
    print("a == b")
else
    print("a ~= b")
end

```

若要实现两表的比较该如何呢？此有点类似于 C/C++ 中的逻辑运算符重载。只是 Lua 中采用元表中的元方法的方式实现。

```
a = { x = 1, y = 2, z = 3 }
```

```

b = { x = 1, y = 2, z = 3 }

if a == b then
    print( "a == b" )
else
    print( "a ~= b" )
end

mt = { }
function mt.__eq( a, b )
    if a.x == b.x and a.y == b.y and a.z == b.z then
        return true;
    else
        return false;
    end
end

setmetatable( a, mt )
setmetatable( b, mt )

if a == b then
    print( "a == b" )
else
    print( "a ~= b" )
end

```

12.4. 表相关的元方法

12.4.1. __index 读元方法

依据前面所学，当我们访问一个表中不存在的字段时，返回值是 nil。下面要学的是，当表中字段不存在时，会引发解析器，去寻找__index 元方法，如果元方法__index 不存在，则返回 nil。

如果存在，__index 的值可以有两种情况，一种是表，一种是方法。具体内容，详见下文。

12.4.1.1. __index 为表

这是 metatable 最常用的键。

当你通过键来访问 table 的时候，如果没有这个键值，那么 Lua 就会寻找该 table 的 metatable(假定有 metatable)中的__index 键。如果__index 的值为一个表，Lua 会在表格中查找相应的键。

```

other = {foo = 3}
t = {}
--t = setmetatable({}, {__index = other})
setmetatable(t, {__index = other})
print(t.foo)
print(t.bar)

```

12.4.1.2. __index 为表->应用

```

prototype = {x = 0, y = 0, width = 100, height = 100}
mt = {__index = prototype}

function new( o )
    setmetatable(o, mt)
    return o
end

w = new {x=11}

print(w.x)
print(w.y)
print(w.width)

```

12.4.1.3. __index 为函数

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。

```

mytable = {}

mt = {__index = function (t, k)      -- function (_, k)
       print(t, k)
       return true
   end}

setmetatable(mytable, mt)

print(mytable)
print(mytable.key)

```

__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 的值函数返回结果。

```

mytable = { key1 = "raw value" }
mt = { }
mt.__index = function( t, k )          -- 体现了 key 的存在
    if k == "key2" then
        return "meta value"
    else
        return nil
    end
end
setmetatable( mytable, mt )

print( mytable.key1, mytable.key2 )

```

```

mytable = setmetatable(
    {key1 = "raw value"},
    {
        __index = function(mytable, key)
            if key == "key2" then
                return "meta value"
            else
                return nil
            end
        end
    })
print(mytable.key1, mytable.key2)

```

12.4.1.4. __index 为函数->应用

```

prototype = {x = 0, y = 0, width = 100, height = 100}

mt = {__index = function ( _,k)
      return prototype[k]
    end}

function new( o )
    setmetatable(o,mt)
    return o
end

w = new {x=11}

print(w.x)
print(w.y)
print(w.width)

```

12.4.1.5. 小结

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。

12.4.2. __newindex 写元方法

依据前面所学，当往一个表中存在的字段赋值时，则为更新字段值，当往一个表中不存在的字段赋值时，则会产生新的字段并赋值。可称之为对本表的更新。

```

t = {x = 1}
t.x = 100
t.y = 200

```

```

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

```

`__index` 用于在本表中不存在字段的查询，而 `__newindex` 用于本表中不存在字段的更新，当对本表中一个不存在字段赋值时，若本表的元表中无 `__newindex` 字段，则在本表中更新。若有些字段，那么解析器就会调用它。

`__newindex` 字段的值同 `__index`，可以是表，也可以是函数。下面分别就两种情况来进行分析。

12.4.2.1. `__newindex` 为表

当你给表的一个缺少的索引赋值，解释器就会查找元表中 `__newindex` 元方法：如果存在则调用这个函数而不进行对本表赋值操作。

`__newindex` 元方法如果是一张表，Lua 对这张表做索引赋值操作，新的 k-v 会出现在表中。

```

mytable = {key = 999}
print(mytable.key)

mytable.newkey = 111
print(mytable.newkey)

```

```

t = {}
mt = {__newindex = t}
mytable = {key = 999}
setmetatable(mytable, mt)

print(mytable.key)

mytable.newkey = 111
print(mytable.newkey)      --nil
print(t.newkey)          --111

```

```

t = {}
mt = {__newindex = t, __index = t}
mytable = {key = 999}
setmetatable(mytable, mt)

print(mytable.key)

mytable.newkey = 111
print(mytable.newkey)      -- 111

print(t.newkey)

```

以上实例中表设置了元方法 `__newindex`，在对新索引键(newkey)赋值时(mytable.newkey = 111)，会调用元方法，而不进行赋值。而如果对已存在的索引键(key)，则会进行赋值，而不调用元方法 `__newindex`。

12.4.2.2. `__newindex` 为函数

若 `__newindex` 元方法，如果是一个函数，则将 **table**、**key**、以及 **value** 作为参数传入函数中。

```
mt = {__newindex = function (t,k,v)
      print(t,k,v)
    end}
mytable = {key = 111}
setmetatable(mytable,mt)

print(mytable)
mytable.newkey = 999
```

```
t = {}
mt = {__newindex = function (_,k,v)
      t[k] = v
    end}
mytable = {key = 111}
setmetatable(mytable,mt)

print(mytable)
mytable.newkey = 999

print(t.newkey)

print(mytable.newkey) -- nil
```

```
t = {}
mt = {__newindex = function (_,k,v)
      t[k] = v
    end,__index = t}
mytable = {key = 111}
setmetatable(mytable,mt)

print(mytable)
mytable.newkey = 999

print(t.newkey)

print(mytable.newkey) -- 999
```

12.4.3. raw method

12.4.3.1. rawget (table, index)

Gets the real value of table[index], without invoking the __index metamethod。table must be a table; index may be any value。

```
mytable = {x = 0}
setmetatable(mytable,{__index = {y = 999}})

print(mytable.x)
print(mytable.y)
print(rawget(mytable,"x"))
print(rawget(mytable,"y"))
```

12.4.3.2. rawset (table, index, value)

Sets the real value of table[index] to value, without invoking the __newindex metamethod。table must be a table, index any value different from nil and NaN, and value any Lua value。

```
t = {}
mytable = {x =0}
setmetatable(mytable,{__newindex =t})

mytable.y = 999
print(t.y)

rawset(mytable,"z",99)
print(t.z)
print(mytable.z)
```

12.4.4. 应用

12.4.4.1. 默认值表

12.4.4.2. 只读表

12.5. 其它元方法

12.5.1. __call 元方法

Lua 中的 __call 元方法，像 c++ 中 [仿函数](#)一样，将对象当函数使用。C++ 中曾举例 cout<<pow(10)<<endl; 注意 pow 此时是个对象，而不是一般的函数。

```
#include <iostream>
using namespace std;
class Pow
{
public:
```

```

int operator()(int x){
    return x * x;
}
};

int main()
{
    Pow pow;
    cout<<pow(2)<<endl;
    return 0;
}

```

当 Lua 尝试调用一个非函数的值表的时候会触发这个事件（即 func 不是一个函数）。查找 func 的元方法，如果找得到，就调用这个元方法，func 作为第一个参数传入，原来调用的参数 (args) 后依次排在后面。

```

pow = {}
setmetatable(pow, {__call = function (t, x) --function (_, x)
    return x * x
end})
print(pow(10))

```

以下实例演示了计算表中元素的和*系数：

```

t = {}
setmetatable(t, {__call = function (t, a, b, factor)
    --t.a = 1;
    --t.b = 2;
    --t.factor = factor
    return (a + b)*factor
end})

print(t(1, 2, 0.1))      --> 0.3

--print(t.a)            --> 1
--print(t.b)            --> 2
--print(t.factor)        --> 0.1

```

12.5.2. __tostring 元方法

在 C++ 中若要输出基本类型，cout 可直接输出，若要输出结构类型，则需要重载 operator<< 运算符。

Lua 中若输出基类型，print 即可，或要输出表，则只会打印表的地址。__tostring 元方法用于修改表的输出行为，__tostring 接受一个参数，即为本表，以下实例我们自定义了表的输出内容：

```

mytable = setmetatable({ 10, 20, 30 }, {
    __tostring = function( mytable )

```

```

sum = 0
for k, v in pairs( mytable ) do
    sum = sum + v
end
return "表所有元素的和为 " .. sum
end
})
print( mytable ) --表所有元素的和为 60

```

12. 5. 3. Other

注：__ 代表两条下划线。

<u>add</u>	对应的运算符 '+'
<u>sub</u>	对应的运算符 '-'
<u>mul</u>	对应的运算符 '*'
<u>div</u>	对应的运算符 '/'
<u>mod</u>	对应的运算符 '%'
<u>unm</u>	对应的运算符 '-'
<u>concat</u>	对应的运算符 '...'
<u>eq</u>	对应的运算符 '=='
<u>lt</u>	对应的运算符 '<'
<u>le</u>	对应的运算符 '<='

13. OO 面向对象

13.1. 类与对象

对象是真实世界的实体，对象与实体是一一对应的，也就是说现实世界的每一个实体都是对象，它是一个具体的概念。

类是具体某些共同特征的实体的集合，它一种抽象的概念。

在常规的语言中，类的定义和对象的定义，是不同的。类的定义是新类型的定义，而对象的的定义，是变量的定义。

而 **lua 中类也是对象，对象也是对象。**

13.1.1. 表对象

所谓对象，即属性和方法。下面示例了，对象的创建方法。以及对象数据成员的访问和函数成员的访问。

```
-- _width = 99

shape = {
    _width = 100,
    _height = 100,
    _getWidth= function ()
        return _width
    end
}

print(shape._width)
print(shape._height)

print(shape._getWidth()) --本局的，可以访问，本表内的不可以访问
```

把表当成一种对象，表中数据成员，和函数成员，均可以访问，有一个很大的局限性问题就在于，表成员函数无法访问表中的数据成员。

C++中的类的成员函数，之所以能访问，类中的数据成员，是因为有一个指向类成员对象的 this 指针。

this 的最大意义，不是代表某一个对象，而是代表，任意调用该函数的对象。

```
Shape = {
    width = 10,
    height = 100,
    -- area = function (self)
        --     return self.width * self.height
    -- end
}
-- Shape.area = function(self)
--     return self.width * self.height
-- end
```

```
--a ,shape = shape ,nil --为什么不直接用 Shape.width Shape.height
Shape.area = function(this)
    return this.width * this.height
end

print(Shape.width)
print(Shape.height)
print(Shape.area(Shape))
```

13.1.2. self 与 this

上例中，对象是不可以直接访问其数据成员的，每次要将对象作为参数传入才可，lua 中提供了语法糖来提供便利。

上例中，self 和 this 是一样的，但是使用语法糖后，只能采用 self 作为默认传入的参数。且语法糖的函数定义只能为 function Shape:area(), 而这种写法 Shape:area = function(), 则不可行，且仅限于在表外实现。

```
Shape = {
    width = 10,
    height = 100,
}
function Shape:area()
    -- return this.width * this.height
    return self.width * self.height
end

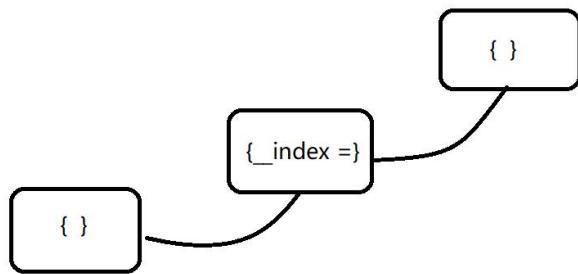
Shape:area = function()
    return self.width * self.height
end

print(Shape.width)
print(Shape.height)
-- print(Shape.area(Shape))
print(Shape:area())
```

13.1.3. metatable 自索引

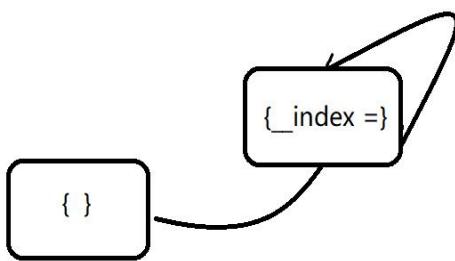
13.1.3.1. setmetatable

下图中的三张表，我们称之为本表，原表和__index 索引表。本表要访问索引表中的内容，必须经历，元表和元方法。



13.1.3.2. 自锁引

所谓的自索引，即将元表中的，`__index`方法指向，元表本身。可以省却一张额外的表。这种作法，在类继承中比较常用。



```

Father = {
    x = 1,y = 2
}

function Father:dis()
    print("in Father:dis", self)
    print(self.x ,self.y)
end

Father.__index = Father

Son = {
    a = 11,b = 99
}
setmetatable(Son,Father)

print(Son.a,Son.b)
print(Son.x,Son.y)
  
```

```
print("Father=", Father)
print("Son=", Son)
Son:dis()
```

```
11 99
1 2
Father= table: 00e488f0
Son=   table: 00e489b8
in Father:dis  table: 00e489b8
1 2
```

通过打印来看，当 Son 来调用 Father 中的 dis 的时候，Father 中的 self 是 son 而不是 Father，此时，self.x，仍然是找不到的。再次找元表，找__index 元方法，__index 自索引到 Father 本身，而 Father 中正好有 x 成员。

13.1.4. 对象

前面讲的 self 和自索引，分别是用来实现类成员函数中数据独立和继承的初步。下面，继续补充如何实现对象的独立，即个自独立的数据，然后调用数据独立的成员函数。

13.1.4.1. Account

lua 中并没有类这个概念，所谓的类也不过是一个对象，只是这个对象，提供了创建新对象的方法。

```
-- 账户余额初始为 0,
Account = { balance = 0 }
```

13.1.4.2. Account:new

以下，是常见的对象生成。即通过 new(非关键字) 函数实现多个 Account，即对象独立。大家思考一下，可以实现吗？

```
function Account:new (o)
    o = o or {}
    -- create object if user does not provide one
    setmetatable(o,self)
    self.__index = self
    return o
end
```

new 完成了 Account 的自索引，并将 Account 设置为 o 的元表，若返回 o 空对象，则 o 对象与 Account 无异。

```
Account.balance = 11
a = Account:new()
b = Account:new()
print("Account balance",Account.balance)      --11
print("a      balance",a.balance)      --11
```

```
print("b      balance", b.balance)           --11
```

如何产生差异呢，重点就在于传递的参数 o 上，如果传递的 o 中均有一个 balance 成员则会产生差异。

```
a = Account:new({balance = 99})
b = Account:new({balance = 199})
print("Account balance",Account.balance)
print("a      balance",a.balance)
print("b      balance",b.balance)

Account balance 0
a      balance 99
b      balance 199
```

13.1.4.3. other member

是不是一定要传参才能体现出差异呢，答案是不需要的。

对象 a 和 b 第一次访问 balance 时候，`self.balance` 会引发`_index` 行为，而 `self.balance` 被赋值，引发的时`_newindex` 行为，此时`_index` 是 Account 而`_newindex` 是空。故而 `self.balance` 访问了父类的 balance，而 `self.balance` 增加了 balance 字段。

继而后继所有的访问，均是访问的 self 所指对象中的元素。

```
function Account:deposit(v)
    self.balance = self.balance + v           --两个 balance 并不一样 写时复制
end
function Account:withdraw(v)
    self.balance = self.balance - v
end
function Account:withdraw(v)
    self.balance = self.balance - v
end
```

Subsequent accesses to b.balance will not invoke the index metamethod, because now b has its own balance field。

```
print("Account balance",Account.balance)
print("a      balance",a.balance)
print("b      balance",b.balance)

a:deposit(100)
print("Account balance",Account.balance)
print("a      balance",a.balance)
print("b      balance",b.balance)

b:deposit(199)
print("Account balance",Account.balance)
print("a      balance",a.balance)
print("b      balance",b.balance)
```

```
Account balance 0
a      balance 0
b      balance 0
Account balance 0
a      balance 100
b      balance 0
Account balance 0
a      balance 100
b      balance 199
```

当然，如果传入的 o 表中有 balance 字段，就更简洁明了了。self.balance 中即为传入对象的 balance 字段。

13.1.4.4. 对象差异性

对象的差异性来自，传入的参数和写时产生字段。

13.2. 继承

13.2.1. 父类

```
Account = {balance = 0}

function Account:new (o)
    o = o or {}
    setmetatable(o,self)
    self.__index = self
    return o
end

function Account:deposit (v)
    self.balance = self.balance + v
end

function Account:withdraw (v)
    -- 普通账户不可以透支。
    if v > self.balance then
        error("insufficient funds")
    end
    self.balance = self.balance - v
end

acc = Account:new()
acc:deposit(300)
print(acc.balance)

acc:withdraw(500)
```

13.2.2. 继承

13.2.2.1. 子类

```
CreditAccount = Account:new()
--[[ 新的实例也可以规定自己的限额。
"CreditAccount" 中没有"new()", 实际调用的是"Account.new()"。 ]]
```

13.2.2.2. 子类对象

```
credit = CreditAccount:new{limit = 2000.00} --({})

-- 为了让信用卡账户能够透支，需要重写"withdraw()"方法。
-- 在本对象内实现的方法，而不再去元表查询
```

13.2.2.3. 重写

```
function CreditAccount:withdraw (v)
    -- 信用卡账户在一定额度内可以透支。
    if v - self.balance >= self:getLimit() then
        error "insufficient funds"
    end
    self.balance = self.balance - v
end

function CreditAccount:getLimit ()
    return self.limit or 0
end
```

13.2.2.4. 应用

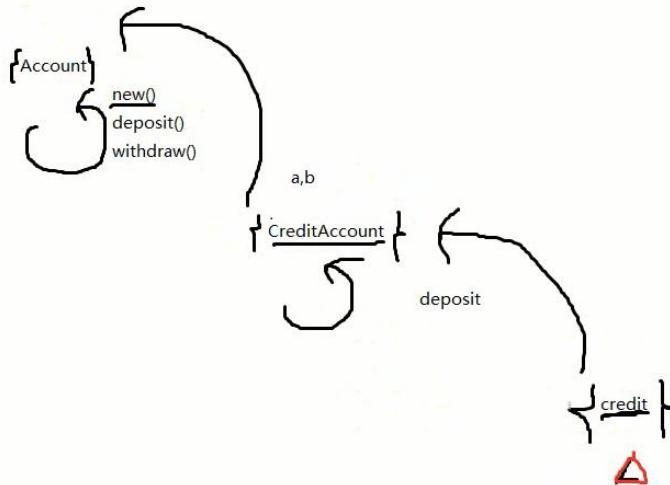
```
--[[ "credit" 中没有"deposit()", 
      "Credit Account" 中没有"deposit()", 实际调用的是"Account.deposit()".]]
credit:deposit(100.00)

-- 此时调用的是"CreditAccount:withdraw()".
credit:withdraw(200.00)

print(credit.balance) --> -100.0
```

13.2.2.5. 小结

读时，在本身中找，找不到到父类中找，写时，在本身中生成新字段。一旦有了新字段，则不需要再引发`_index`行为。



13.2.3. 成员私有化

使用两个"table"，其一存储私有成员，另一个存储公有成员和公有方法，两个"table"组成"Closure"，私有"table"作为公有"table"的"Closure"被访问，私有方法直接存储在"Closure"中。

```

function newAccount( initialBalance )
    -- 私有"table".
    local priv = {
        balance = initialBalance,
        count = 0
    }
    -- 私有方法,未导出到公有"table"中,外部无法访问.
    local addCount = function( v )
        priv.count = priv.count + v * 0.1 -- 消费的 10%作为积分.
    end

    local withdraw = function ( v )
        priv.balance = priv.balance - v
    end

    local deposit = function ( v )
        priv.balance = priv.balance + v
        addCount( v )
    end

    local getBalance = function ( )
        return priv.balance
    end

```

```

local getCount = function( )
    return priv.count
end
-- 公有"table".
return {
    withdraw = withdraw,
    deposit = deposit,
    getBalance = getBalance,
    getCount = getCount
}
end

acc = newAccount(1000)
print(acc.balance)      -- nil
print(acc.getBalance()) -- 1000

print(acc.getCount())   -- 0
acc.deposit(99)
print(acc.getBalance()) -- 1099
print(acc.getCount())   -- 9.9

```

13.3. cocos2dx-lua 之 class 函数

13.3.1. 环境搭建

13.3.2. 继承函数

```

function class(classname,super)
    local superType = type(super)
    local cls

    if superType ~= "function" and superType ~= "table" then
        superType = nil
        super = nil
    end
    if superType == "function" or (super and super.__ctype == 1) then
        -- inherited from native C++ Object
        cls = {}

        if superType == "table" then
            -- copy fields from super
            for k,v in pairs(super) do cls[k] = v end
            cls.__create = super.__create
            cls.super    = super
        else
            cls.__create = super
            cls.ctor    = function() end
        end
    else
        error("Unsupported super type: " .. superType)
    end
end

```

```

end

cls.__cname = classname
cls.__ctype = 1 --C++

function cls.new(...)
    local instance = cls.__create(...)
    -- copy fields from class to native object
    for k,v in pairs(cls) do instance[k] = v end
    instance.class = cls
    instance:ctor(...)
    return instance
end

else
    -- inherited from Lua Object
    if super then
        cls = {}
        setmetatable(cls,{__index = super})
        cls.super = super
    else
        cls = {ctor = function() end}
    end

    cls.__cname = classname
    cls.__ctype = 2 -- lua
    cls.__index = cls

    function cls.new(...)
        local instance = setmetatable({},cls)
        instance.class = cls
        instance:ctor(...) -- 调用父类或自实现
        return instance
    end
end

return cls
end

```

13.3.3. 实例分析

```

A = class("A")

function A:ctor()
    print("A:ctor")
end
function A:run()

```

```

for i=0,10 do
    print("====",i,"====")
end

a = A:new()
a:run()

```

13.4. 面向对象纵深要点

13.4.1. 函数 self

self保证了函数的独立性，只有逻辑没有数据，数据来自传入的调用者自身。

13.4.2. 对象 o = o or {}

对象提供的 new 中返回对象 o，保证的数据的独立性。

13.4.3. 机制 meta

lua 中的 metatable 和 metafunction，提供了继承机制。

self.balance = self.balance -v 。两个 self.balance 分别代表，在本对象添加新的成员和本对象内找不到会去父类查找。

还有对象函数成员自实现，会覆盖父类成员，以实现增加新功能。

13.5. 更简单的面向对象

面向对象的两大特性，一封装，二继承，封装已有表 table 来表示，继承，即复用，只要实现了复用，也就是实现了继承。如下采用了复制表和闭包的方式来实现复用。也是 lua 中常用的方式。

13.5.1. 复制表实现

13.5.1.1. 原理解析

要想复用，简单粗暴的方式就是去拷贝一分，对于拷贝来的，即为继承，再对其添加即为扩展。

13.5.1.2. People

```

local People= {}
People.sayHi = function (self)
    print("people say hi to ",self.name)
end

People.new = function (name)
    local self = clone(People)
    self.name = name
    return self
end

function clone(tab)

```

```
local ins = {}
for k,v in pairs(tab) do
    ins[k] = v
end
return ins
end

local p = People.new("zhangsan")
-- p.sayHi(p)
p:sayHi()
```

13.5.1.3. Man public People

```
local People= {}
People.getName = function (self)
    print("my Name is ",self.name)
end

People.new = function (name)
    local self = clone(People)
    self.name = name
    return self
end

function clone(tab)
    local ins = {}
    for k,v in pairs(tab) do
        ins[k] = v
    end
    return ins
end

local p = People.new("zhangsan")
-- p.sayHi(p)
p:getName()

function copy(dst, src)
    for k,v in pairs(src) do
        dst[k] = v
    end
end
local Man= {}

Man.new = function(name, age)
    local p = People.new(name)
    Man.age = age
end
```

```

copy(p,Man)
return p
end
Man.getAge = function(self)
    print("my age is "..self.age)
end

Man.getName = function(self)
    print("MY NAME IS "..self.age)
end
local m = Man.new("lisi",18)

-- m.getName(m)
-- m.getAge(m)

m:getName()
m:getAge()

```

13.5.2. 闭包实现

13.5.2.1. 闭包回顾

当一个函数内嵌套另一个函数的时候，内函数可以访问外部函数的局部变量。具体可解析为：1) 外部函数，通常可以称为工厂函数，2) 内部函数通常称为闭包，3) 可访问的局部变量称为上值。

如果，上值是一个表，内部函数是一个表的函数成员，会如何呢？

13.5.2.2. People

```

local function People(name)
    local self = {}
    local function init()
        self.name = name
    end
    self.getName = function ()
        print("my name is "..self.name)
    end
    init()
    return self
end

local p = People("zhangsan")
p.getName()

```

13.5.2.3. Man public People

```

local function People(name)
    local self = {}

```

```
local function init()
    self.name = name
end
self.getName = function ()
    print("my name is "..self.name)
end
init()
return self
end

local p = People("zhangsan")
p.getName()

local function Man(name, age)
    local self = People(name)
    local function init( )
        self.age = age
    end
    self.getAge = function()
        print("myage is "..self.age)
    end
    self.getName = function ()
        print("MY NAME IS "..self.name )
    end
    init()
    return self
end

local m = Man("lisi",18)
m.getName()
m.getAge()
```

14. Env 环境

14.1. _G

所谓 lua 的环境就是指的是_G 这张表，_G 也是一张普通的全局表。只是他存储了全局环境的值。

14.1.1. 打印_G

```
for k,v in pairs(_G) do
    print(k,v)
end
```

string	table: 000000001DAACAC0
xpcall	function: 000000001DAA9DB0
package	table: 000000001DAAC660
tostring	function: 000000001DAA9BD0
print	function: 000000001DAA9FC0
os	table: 000000001DAAC8E0
unpack	function: 000000001DAA9E10
require	function: 000000001DB01450
getfenv	function: 000000001DAA7190
setmetatable	function: 000000001DAAA050
next	function: 000000001DAAA230
assert	function: 000000001DAA7280
tonumber	function: 000000001DAAA0B0
io	table: 000000001DAA8420
rawequal	function: 000000001DAAA260
collectgarbage	function: 000000001DAA6F80
arg	table: 000000001DAACB60
load	function: 000000001DAAA2F0
module	function: 000000001DB01270
rawset	function: 000000001DAA9C30
class	function: 000000001DB026A0
java	table: 000000001DAAC840
math	table: 000000001DAACBB0
debug	table: 000000001DAAC890
pcall	function: 000000001DAA9C90
getmetatable	function: 000000001DAA73D0
table	table: 000000001DAA7780
type	function: 000000001DAAA200
coroutine	table: 000000001DAA7950
newproxy	function: 000000001DAA9380
select	function: 000000001DAAA110

```

_G                      table: 000000001DAA61E0
gcinfo                 function: 000000001DAA7340
rawget                 function: 000000001DAA9ED0
loadstring              function: 000000001DAA9EA0
pairs                  function: 000000001DAA94C0
ipairs                 function: 000000001DAA9A80
dofile                 function: 000000001DAA6D40
_VERSION                Lua 5.3
setfenv                function: 000000001DAAA350
error                  function: 000000001DAA70D0
loadfile               function: 000000001DAA7400

```

14.1.2. 使用 G

14.1.2.1. G.xxx

G 代表的时全局有名空间，类似于 C++ 中的全局无名命名空间 ::，假设有全局函数，也可以这样设用的，::func()。

```

_G.print("abc")
_G.print(_G.math.huge)
_G.print(_G.string.format("name = %5s age=%d org = %10s","lua",0,"nzhsoft"))

ct = {1,3,5,7,9,2,4,6,8,10}
_G.table.sort(_G.t)
_G.print(table.concat(_G.t,"-"))

_G.math.randomseed(_G.os.time())
for i=1,10 do
    print(_G.math.random(100))
end

```

14.1.2.2. ldt 中调试 G

14.2. 沙盒 sandbox

14.2.1. 改变环境(lua5.1)

函数的上下文环境可以通过 setfenv(f, table) 函数改变，其中 table 是新的环境表，f 表示需要被改变环境的函数。如果 f 是数字，则将其视为堆栈层级(Stack Level)，从而指明函数(1 为当前函数，2 为上一级函数)：

```

a = 3          -- 全局变量 a
setfenv(1,{}) -- 将当前函数的环境表改为空表
print(a)       -- 出错，因为当前环境表中 print 已经不存在了

```

不仅是 a 不存在，连 print 都一块儿不存在了。如果需要引用以前的 print 则需要在新的环境表中放入线索：

```

function func()
    a = 3
    setfenv(1,{g = _G})           --setfenv(1,{print = _G.print})

```

```

g.print(a)                      --nil
g.print(g.a)                    -- 3
end
func()
print(a)                        --3

```

14. 2. 2. 沙盒(lua5.1)

于是，出于安全或者改变一些内置函数行为的目的，需要在执行 Lua 代码时改变其环境时便可以使用 `setfenv` 函数。仅将你认为安全的函数或者[新的实现](#)加入新环境表中：

```

local env = {
    print = _G.print,
    os = _G.os,
    math = _G.math
}                                     -- 沙盒环境表，按需要添入允许的函数

function run_sandbox(code)
    local func,message = loadstring(code)
    if not func then return nil,message end -- 传入代码本身错误
    setfenv(func,env)
    return pcall(func)
end

chunk= "print(\"abc\") print(os.time()) print(math.pi)"
run_sandbox(chunk)

```

14. 2. 3. __ENV 改写

详见 [14.3.2](#)

14. 3. __ENV 变量(lua5.3)

14. 3. 1. __ENV

Lua 5.2 中所有对全局变量 `var` 的访问都会在语法上翻译为 `__ENV.var`。而 `__ENV` 本身被认为是处于当前块外的一个局部变量。(于是只要你自己定义一个名为 `__ENV` 的变量，就自动成为了其后代码所处的「环境」(environment)。另有一个「全局环境」(global environment)的概念，指初始的 `_G` 表。)

```

print(__ENV)
print(_G)
--table: 00751340
--table: 00751340
print(type(__ENV))                  --> table
for k,v in pairs(__ENV) do print(k, v) end -- 打印当前运行环境中所有全局变量。

```

```
a = 10      print(__ENV["a"]) --> 10
```

```
a = 1                                -- create a global variable
_ENV = {}                            -- change current environment to a new empty table
print(a)                             --> attempt to call a nil value (global 'print')
```

```
a = 1                                -- create a global variable -- "a"存储在"_G"中, "_G.a"
_ENV = {_G = _G}                      -- 新的环境变量中"_G"域存储原先的"_G"
_G.print(a) --> nil                  -- "a"在"_G"中, 不在当前的环境变量中
_G.print(_G.a) --> 1
```

这里看出两个指向的是同一个 table。那么这两个是什么关系呢？

Lua 官方说明文档中：

When Lua loads a chunk the default value for its `_ENV` upvalue is the global environment (see `load`)。

Therefore, by default, free names in Lua code refer to entries in the global environment (and, therefore, they are also called global variables).

Moreover all standard libraries are loaded in the global environment and some functions there operate on that environment.

You can use `load` (or `loadfile`) to load a chunk with a different environment. (In C, you have to load the chunk and then change the value of its first upvalue.)

一个 chunk 就是 lua 的一个解释单元，可以存储在文件或者字符串中。对于每一个 chunk，都有一个叫 `_ENV` 的 upvalue，此时 `_ENV` 的初值就是 `_G`。在 chunk 内的函数，都会有这个 upvalue 值。修改当前的 chunk 的 `_ENV`，也就修改了 `_G`，那么在该代码块中加入的非 local 变量，可以直接通过名称在其他 chunk 中访问到（当然该 chunk 的 `_ENV` 也得是 `_G`）。

```
a = 3
function get_echo()
    local _ENV = {print = print, a = 2}
    return function()
        print(a)
    end
end

get_echo()()
```

14.3.2. `_ENV` 改写 5.1 的案例

```
local env = {
    print = _G.print,
    os = _G.os,
    math = _G.math,
    pcall = _G.pcall
```

```
} -- 沙盒环境表，按需要添入允许的函数

function run_sandbox(code)
    local func, message = load(code)           -- 5.3 中没有 loadstring
    if not func then return nil, message end   -- 传入代码本身错误
    local _ENV = env
    return pcall(func)
end

chunk = "print(\"abc\") print(os.time()) print(math.pi)"
run_sandbox(chunk)
```

15. Require 模块

15.1. 引入

15.1.1. 定义包

现有 mymath.lua，代码如下：

```
mymath = {}  
function mymath.add(a, b)  
    return(a + b)  
end  
function mymath.sub(a, b)  
    return(a - b)  
end  
function mymath.mul(a, b)  
    return(a * b)  
end  
function mymath.div(a, b)  
    return(a / b)  
end  
return mymath
```

15.1.2. 引用包(调试)

main.lua 中引用 mymath.lua

```
require "mymath"  
print(mymath.add(4, 5))  
  
my= require ("mymath")  
print(my.add(4, 5))
```

15.1.3. 注意事项

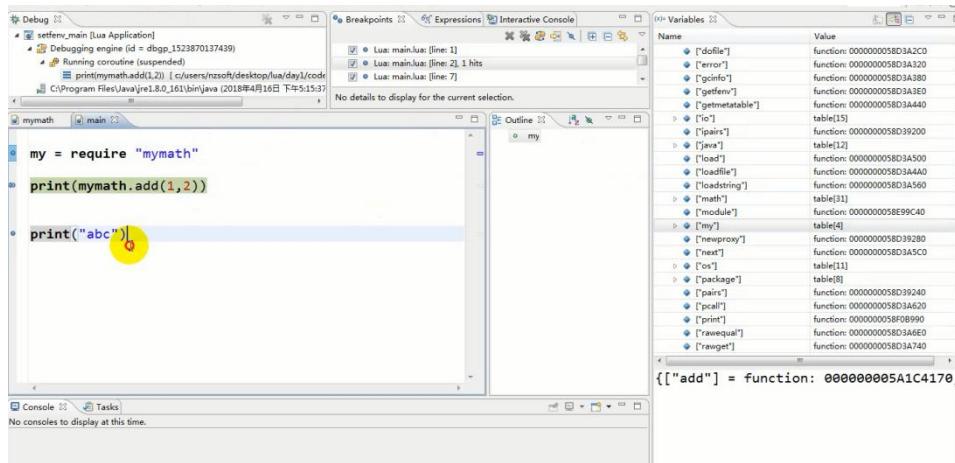
定义的包，如果是全局的，require 以后为全局的，即可直接使用本包，也可以在 require 返回后，赋值给全局或本地变量。

15.1.3.1. require 覆盖

定义的包中，尽量使用 **local** mymath，如果全局中有 mymath 则会造成覆盖。如果引发覆盖而不报错，则后果不堪设想。local 会尽可能少的引用全局变量。

```
mymath = {x = 10, y = 20}  
local res = require("mymath")  
print(res)  
print(mymath.x)
```

15.1.3.2. ldt 中的 local 查看



15.1.4. 加载次数

多次打印，返回的表地址是一样的，说明只加载一次。

```
print(require("mymath"))
print(require("mymath"))
```

```
print(require("mymath"))
```

15.2. require

15.2.1. 定义/原理

为了方便代码管理，通常会把 lua 代码分成不同的模块，然后在通过 require 函数把它们加载进来。现在看看 lua 的 require 的处理流程。

假设有模块叫，xxx.lua，实际上 require "xxx" 后，会将 xxx 中的全局函数和数据放到表 _G 中，所以也就能访问了。

多次执行 require "xxx"，xxx.lua 只会被加载一次，可以多次打印其返值得到相同的 table 地址。

15.2.2. 加载流程

15.2.2.1. package.loaded

加载一个模块。这个函数首先查找 package.loaded 表，检测 modname 是否被加载过。如果被加载过，require 返回 package.loaded[modname] 中保存的值。

```
for k, v in pairs(package.loaded) do
    print (k, v)
end

string  table: 00fd8710
coroutine  table: 00fd7858
os  table: 00fd8558
bit32  table: 00fd83a0
```

```
package table: 00fd75b0
math    table: 00fd8468
utf8    table: 00fd8418
debug   table: 00fd8508
table   table: 00fd7808
_G     table: 00fd2dd0
io     table: 00fd7880
```

15.2.2.2. package.preload

然后 require 查找 package.preload[modname]，预加载的库。

```
for k, v in pairs(package.preload) do
    print (k, v)
end
```

15.2.2.3. package.path

然后搜索 package.path，加载 lua 文件

```
print(package.path)

C:\lua\lua\?.lua;C:\lua\lua\?\init.lua;C:\lua\?.lua;C:\lua\?\init.lua;C:\lua\..
\share\lua\5.3\?.lua;C:\lua\..\share\lua\5.3\?\init.lua;.\?.lua;.\?\init.lua
```

15.2.2.4. package.cpath

然后搜索 package.cpath，加载 c 库。

```
print(package.cpath)

C:\lua\?.dll;C:\lua\..\lib\lua\5.3\?.dll;C:\lua\loadall.dll;.\?.dll
```

15.2.3. require 搜索路径

打印 package.path 和 package.cpath 如下：

假现在 require("xx")，如下给出的是找不到的情况，若是从上到下执行顺序中，找到了模块则会返回，下面的不再执行。

```
lua53: C:\Users\Guilin\Desktop\aa.lua:291: module 'xx' not found:
no field package.preload['xx']
no file 'C:\lua\lua\xx.lua'
no file 'C:\lua\lua\xx\init.lua'
no file 'C:\lua\xx.lua'
no file 'C:\lua\xx\init.lua'
no file 'C:\lua\..\share\lua\5.3\xx.lua'
no file 'C:\lua\..\share\lua\5.3\xx\init.lua'
no file '.\xx.lua'
no file '.\xx\init.lua'
no file 'C:\lua\xx.dll'
no file 'C:\lua\..\lib\lua\5.3\xx.dll'
no file 'C:\lua\loadall.dll'
no file '.\xx.dll'
```

可以看到，上图中的 xx 即是 path 和 cpath 中的? 部分。也就是说，将 require 的参数，填入到，? 部分，组成完整的搜索路径。

15.3. 实现原理

15.3.1. 测试

15.3.1.1. mymath 为空

```
local res = require("mymath")
print(res)      --true
```

15.3.1.2. return something

mymath.lua 中有啥返回啥，没有啥，返回 true

```
print("mymath")
-- return "abc"
-- return 1
mymath = {}
return mymath
```

返回后，可以打印

```
local res = require("mymath")
print(res)
print(package.loaded["mymath"])
```

15.3.2. 代码

```
function require(name)
    if not package.loaded[name] then
        local loader = findloader(name)
        if loader == nil then
            error("unable to load module" .. name)
        end

        package.loaded[name] = true
        local res = loader(name)
        if res ~= nil then
            package.loaded[name] = res
        end
    end
    return package.loaded[name]
end
```

16. Coroutine 协程

16.1. 概述

16.1.1. 官方协程

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。

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。

A coroutine is similar to a thread (in the sense of multithreading): it is a line of execution, with its own stack, its own local variables, and its own instruction pointer; it shares global variables and mostly anything else with other coroutines.

The main difference between threads and coroutines is that a multithreaded program runs several threads in parallel, while coroutines are collaborative: at any given time, a program with coroutines is running only one of its coroutines, and this running coroutine suspends its execution only when it explicitly requests to be suspended

16.1.2. 协程注解

协程本质上是在一个线程里面，因此不管协程数量多少，它们都是串行运行的，也就是说不存在同一时刻，属于同一个线程的不同协程同时在运行。因此它本身避免了所有多线程编程可能导致的同步问题。

协程的行为有点像函数调用，它和函数调用的不同在于，对于函数调用来说，假如 A 函数调用 B 函数，则必须等待 B 函数执行完毕之后程序运行流程才会重新走回 A，但是对于协程来说，如果在协程 A 中切到协程 B，协程 B 可以选择某个点重新回到 A 的执行流，同时允许在某个时刻重新从 A 回到 B 之前回到 A 的那个点，这在函数中是不可能实现的。因为函数只能一走到底。

用 knuth 的话来说：子程序就是协程的一种特例。即函数 A 调用了函数 B，B 完成后返回 A，而无交互的版本。

16.2. 接口 Api

S.N.	Method & Purpose
1.	coroutine.create (f) 根据一个函数 f 创建一个协同程序，参数为一个函数，返回 <code>thread</code> 类型的协程句柄。

2.	coroutine.status (co) 查看协同状态： 1>suspended (挂起， 协同刚创建完成时或者 yield 之后) 2>running (运行) 3>dead (函数走完后的状态， 这时候不能再重新 resume)
3.	coroutine.yield (...) 使正在运行的协同挂起， 可以传入参数
4.	coroutine.resume (co [, val1, ...]) 使协同从挂起变为运行： 1>激活 coroutine， 也就是让协程函数开始运行； 2>唤醒 yield， 使挂起的协同接着上次的地方继续运行。该函数可以传入参数
5.	coroutine.wrap (f)

函数名	函数参数	函数返回值	函数作用
coroutine.create(f)	接受单个参数，这个参数是coroutine的主函数	然后返回它的控制器，(一个对象为thread)的对象	create函数创建一个新的coroutine，定义了携程内的任务流程。 以面向对象的角度，可以看成是coroutine类创建了一个对象co
coroutine.resume(co, [, val1, ...])	第一个参数：coroutine.create的返回值，即一个thread对象。 第二个参数：coroutine中执行需要的参数，是一个变长参数，可传任意多个。	如果程序没有任何运行错误的话，那么会返回true，之后的返回值是前一个调用coroutine.yield中传入的参数 如果有任何错误的话，就会返回false，加上错误信息。	当你第一次调用coroutine的resume方法时，coroutine从主函数的第一行开始执行，之后在coroutine开始运行后，它会一直运行到自身终止或者是coroutine的下一个yield函数。
coroutine.yield(...)	传入变长参数	返回在前一个调用coroutine.resume()中传入的参数值	挂起当前执行的协程，这个协程不能正在使一个C函数，一个元表或者一个迭代器。
coroutine.running()	空	返回当前正在的协程 如果它被主线程调用的话，会返回nil	
coroutine.status()	空	返回当前协程的状态：有running,suspended,normal,dead	

16.2.1. 协程状态

协同有三个状态：挂起态(suspended)、运行态(running)、停止态(dead)。当我们创建协同程序成功时，其为挂起态，即此时协同程序并未运行。我们可用 coroutine.status 函数，检查协同的状态：

```
co = coroutine.create(
    function()
```

```

print("hello nzhsoft")
print(coroutine.status(co))

end)
print(type(co)) -- thread
print(coroutine.status(co))
print(coroutine.resume(co))
print(coroutine.status(co))

```

以上的协程，像是一个复杂的函数，直到 `yield` 出现。

```

co = coroutine.create(function ()
    for i = 1, 5 do
        print("co", i)
        coroutine.yield()
    end
end)

print(coroutine.status(co))
coroutine.resume(co)
print(coroutine.status(co))
print("=====")
print(coroutine.resume(co))
print("=====")
print(coroutine.resume(co))
print("=====")
print(coroutine.resume(co))
print("=====")
print(coroutine.resume(co))
print("=====")
print(coroutine.resume(co)) -- prints nothing
print("=====")
print(coroutine.resume(co))

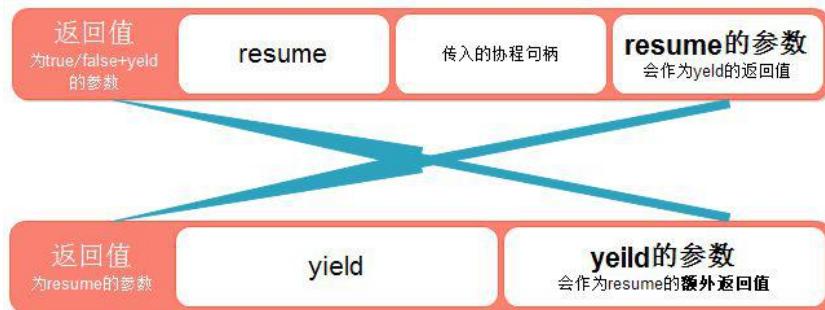
```

16.2.2. 传参示例

Lua 中协同的强大能力，还在于通过 `resume-yield` 来 **交换数据**。

Lua 的协同称为不对称协同 (**asymmetric coroutines**)，指“挂起一个正在执行的协同函数”与“使一个被挂起的协同程序再次执行的函数”是不同的，有些语言提供对称协同 (**symmetric coroutines**)，即使用同一个函数负责“执行与挂起间的状态切换”。

非协同协程，正是能过两个函数来实现的，`coroutine.resume` 和 `coroutine.yield`。协同的同时传递相关的数据。**即 resume 和 yield 均是阻塞型的函数**。



Lua 中协同的强大能力，还在于通过 resume-yield 来交换数据：

- 1>resume 把参数传给程序（相当于函数的参数调用）；
- 2>数据由 yield 传递给 resume；
- 3>resume 的参数传递给 yield；
- 4>协同代码结束时的返回值，也会传给 resume

协同中的参数传递形势很灵活，一定要注意区分，在启动 coroutine 的时候，resume 的参数是传给主程序的；在唤醒 yield 的时候，参数是传递给 yield 的。

16.2.2.1. 引例

The first resume, which has no corresponding yield waiting for it, passes its extra arguments to the coroutine main function:

```
co = coroutine.create(function (a, b)
    print("co", a, b)
end)
print(coroutine.resume(co, 10, 20))      --10  20
```

use return, We seldom use all these facilities in the same coroutine, but all of them have their uses.

```
co = coroutine.create(function (a, b)
    print("co", a, b)
    return a + b, a - b
end)

print(coroutine.resume(co, 10, 20))      --30 -10
```

A call to coroutine.resume returns, after the true that signals no errors, any arguments passed to the corresponding yield:

```
co = coroutine.create(function (a, b)
    print("co", a, b)
    coroutine.yield( a+b, a-b)
end)

print(coroutine.resume(co, 10, 20))      --30 -10
```

Symmetrically, coroutine.yield returns any extra arguments passed to the corresponding resume。

```
co = coroutine.create(function (a, b)
    print("berofe co yield ",a, b)
```

```

a,b = coroutine.yield(a, b)
print("after co yield ", a, b)
end)

print("main coroutine",coroutine.resume(co,10,20))
print("main coroutine",coroutine.resume(co,1,2))

```

`yield` 的目的，就是为了让协程停下来，并且传值给 `resume` 作为返回。而 `resume` 再次开启协程，让 `yield` 返回，并获取 `resume` 传入参数。使新的协程再次有了新的状态值。

16.2.2.2. 综合

跑如下两个综合案例，分析，`resume-yield` 的相互传参关系。

```

co = coroutine.create(function (a)
    local r = coroutine.yield(a + 1)
        -- yield()返回 a+1 给调用它的 resume()函数，即 2
    print("r=" ..r)
        -- r 的值是第 2 次 resume()传进来的，100
end)
status, r = coroutine.resume(co, 1)
print(status,r)
-- resume()返回两个值，一个是自身的状态 true,一个 is yield 的返回值 2
coroutine.resume(co, 100)           --resume()返回 true

```

```

function yieldReturn(arg) return arg end

co_yieldtest = coroutine.create(
    function()
        print("启动协程状态"..coroutine.status(co_yieldtest))
        print("--")
        coroutine.yield()
        coroutine.yield(1)
        coroutine.yield(print("第 3 次调用"))
        coroutine.yield(yieldReturn("第 4 次调用"))
        return 2
    end
)

print("启动前协程状态"..coroutine.status(co_yieldtest))
print("--")

for i = 1, 6 do
    print("第"..i.."次调用协程:", coroutine.resume(co_yieldtest))
    print("当前协程状态"..coroutine.status(co_yieldtest))
    print("--")
end

```

16.2.3. 练习(双 while 交互)

以下程序输出什么？

分析：c1 启动，拉起了 c2，c1 阻塞在 resume 上，传递给 c2 的参数，100，300 无参数接收。

c2 启动后，yield 出让线程，c2 阻塞，c1 唤醒，resume 返回，res=true v1=1 v2=3 并打印。

c1 第一轮循环结束后，继续，第二轮循环，再次 resume，c1 阻塞，c2 唤醒，yield 返回，此时收到 c1 传来的参数 v1=100，v2=300，打印。

c2 第一轮循环结束后，继续，第二轮循环，再次 yield 出让线程，c2 阻塞，c1 唤醒，resume 返回，res=true v1=1 v2=3 并打印。

如此往复

```
function Sleep(n)
    local start = os.clock()
    while os.clock() - start <= n do end
end

function f1()
    while true do
        local res, v1, v2 = coroutine.resume(c2, 100, 300)
        Sleep(1)
        print("co c1 f1", v1, v2)
    end
end
c1 = coroutine.create(f1)

function f2()
    while true do
        local v1, v2 = coroutine.yield(1, 3)
        Sleep(3)
        print("co,c2,f2", v1, v2)
    end
end
c2 = coroutine.create(f2)

coroutine.resume(c1)
```

16.3. 应用

16.3.1. 管道-生产者与消费者

最具代表性的例子是用来解决生产者-消费者问题。假定有一个函数不断地生产数据，另一个函数不断的处理这些数据。

16.3.1.1. 思路

消费者，发起消费行为，但是没有数据，`resume`一个生产者线来，等待数据到来。此时生产者，开始产生数据，数据生产出来以后，`yield`出让线程阻塞，并回传数据，消费者收到数据后，打印。

消费者，再次`resume`发起消费，逻辑同上。也是双`while`结构。

16.3.1.2. 实现

```

function productor()
    local i = 0
    while true do
        i = i + 1
        send(i)          -- 将生产的物品发送给消费者
    end
end

function consumer()
    while true do
        local i = receive()  -- 从生产者那里得到物品
        print(i)
    end
end

function receive()
    local status, value = coroutine.resume(co)
    return value
end

function send(x)
    coroutine.yield(x)      -- x 表示需要发送的值，值返回以后，就挂起该协同程序
end

-- 创建协程 获得句柄
co= coroutine.create(productor)
-- 消费驱动
consumer()

```

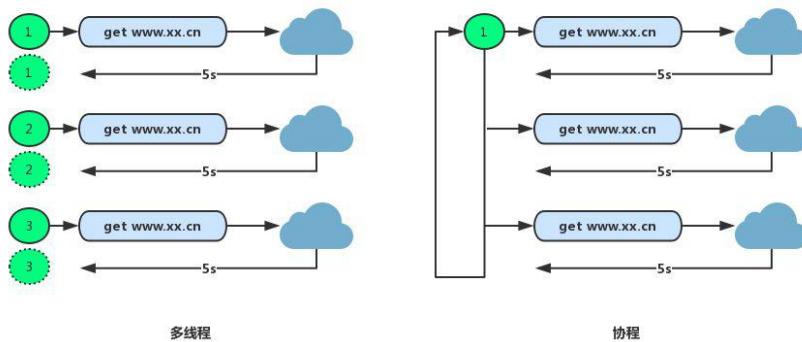
16.3.1.3. 生产消驱动

Therefore, we have what we call a consumer-driven design. Another way to write the program is to use a producer-driven design, where the consumer is the coroutine. Although the details seem reversed, the overall idea of both designs is the same.

16.3.2. 非抢占式线程

Lua 中的协同是一协作的多线程，每一个协同等同于一个线程，`yield-resume` 可以实现在线程中切换。然而与真正的多线程不同的是，协同是非抢占式的。

当一个协同正在运行时，不能在外部终止他。只能通过显示的调用 `yield` 挂起他的执行。对于某些应用来说这个不存在问题，但有些应用对此是不能忍受的。



非抢占式调用的程序是容易编写的。不需要考虑同步带来的 bugs，因为程序中的所有线程间的同步都是显示的。你仅仅需要在协同代码超出临界区时调用 `yield` 即可。

以 lua5.3 为例，使用 luasocket 示例非抢占式线程。

16.3.2.1. lua socket

```
# apt-get install luarocks
# luarocks install luasocket

root@nzhsoft:/usr/local/lib/luarocks/rocks/luasocket/3.0rc1-2/doc# lua
Lua 5.3.5 Copyright (C) 1994-2018 Lua.org, PUC-Rio
> require "socket"
table: 0x5558363be0e0
>

Lua 5.3.5 Copyright (C) 1994-2018 Lua.org, PUC-Rio
> print(package.cpath)
/usr/local/lib/lua/5.3/??.so;/usr/local/lib/lua/5.3/loadall.so;./??.so
```

或者：

```
sudo git clone https://github.com/diegonehab/luasocket
cd luasocket
sudo make
sudo make install-both
```

16.3.2.2. 测试示例

```
local socket = require "socket"

function download (host, file)
    local c = assert(socket.connect(host, 80))
    local count = 0                                -- counts number of bytes read
    local request = string.format(
        "GET %s HTTP/1.0\r\nhost: %s\r\n\r\n", file, host)
    c:send(request)
    while true do
        local s, status = receive(c)
        count = count + #s
    end
end
```

```

        if status == "closed" then break end
    end
    c:close()
    print(file, count)
end

function receive (connection)
    connection:settimeout(0)                                -- do not block
    local s, status, partial = connection:receive(2^10)
    if status == "timeout" then
        coroutine.yield(connection)
    end
    return s or partial, status
end

tasks = {}                                              -- list of all live tasks

function get (host, file)
    -- create coroutine for a task
    local co = coroutine.wrap(function ()
        download(host, file)
    end)                                                 -- insert it in the list
    table.insert(tasks, co)
end

```

dispatch 程序

```

function dispatch ()
    local i = 1
    while true do
        if tasks[i] == nil then                         -- no other tasks?
            if tasks[1] == nil then                      -- list is empty?
                break                                     -- break the loop
            end
            i = 1                                       -- else restart the loop
        end
        local res = tasks[i]()                          -- run a task
        if not res then                               -- task finished?
            table.remove(tasks, i)
        else
            i = i + 1                                 -- go to next task
        end
    end
end

```

select 版本的 dispatch

```
function dispatch ()
    local i = 1
    local timedout = {}
    while true do
        if tasks[i] == nil then -- no other tasks?
            if tasks[1] == nil then -- list is empty?
                break -- break the loop
            end
            i = 1 -- else restart the loop
            timedout = {}
        end
        local res = tasks[i]()
        if not res then -- run a task
            table.remove(tasks, i)
        else -- task finished?
            i = i + 1
            timedout[#timedout + 1] = res
            if #timedout == #tasks then -- all tasks blocked?
                socket.select(timedout) -- wait
            end
        end
    end
end
```

16.3.2.3. 结果对比

1>开始下载

```
get("www.lua.org", "/ftp/lua-5.3.2.tar.gz")
get("www.lua.org", "/ftp/lua-5.3.1.tar.gz")
get("www.lua.org", "/ftp/lua-5.3.0.tar.gz")
get("www.lua.org", "/ftp/lua-5.2.4.tar.gz")
get("www.lua.org", "/ftp/lua-5.2.3.tar.gz")
dispatch() -- main loop
```

2>运行结果

```
/ftp/lua-5.2.3.tar.gz251390
/ftp/lua-5.3.2.tar.gz288430
/ftp/lua-5.3.1.tar.gz282596
/ftp/lua-5.2.4.tar.gz252846
/ftp/lua-5.3.0.tar.gz278240
[Finished in 10.3s]
```

select dispatch 版本

```
/ftp/lua-5.2.3.tar.gz251390
/ftp/lua-5.3.2.tar.gz288430
/ftp/lua-5.2.4.tar.gz252846
/ftp/lua-5.3.1.tar.gz282596
/ftp/lua-5.3.0.tar.gz278240
[Finished in 3.0s]
```

16.4. lua 包 package 的安装

16.4.1. luarocks

lua 的第三方库，统一由 luarocks 来管理。官方地址：<https://luarocks.org/> 第三方库非常的多也非常的简单易用。

The screenshot shows the Luarocks website homepage. At the top, there's a search bar and navigation links for 'Install · Docs · Log In · Register'. Below the header, there's a large image of a blue cube with spheres, followed by text explaining what Luarocks is: 'Luarocks is the package manager for Lua modules. It allows you to create and install Lua modules as self-contained packages called rocks. You can download and install Luarocks on Unix and Windows.' A 'Get started' link is also present. Below this, there are two sections: 'Recent Modules' and 'Most Downloaded'. The 'Recent Modules' section lists several modules like 'logger', 'chars', 'HC', 'suit', and 'chess-fen'. The 'Most Downloaded' section lists modules like 'lua-cjson', 'lua-geolp', 'LuaFileSystem', 'LuaSocket', and 'LPeg', each with its download count and a brief description.

16.4.2. 下载及安装

```
$ wget https://luarocks.org/releases/luarocks-2.4.4.tar.gz
$ tar zxfp luarocks-2.4.4.tar.gz
$ cd luarocks-2.4.4
$ ./configure; sudo make bootstrap
```

```
$ apt-get install luarocks
$ sudo luarocks install luasocket
$ lua
Lua 5.3.4 Copyright (C) 1994-2017 Lua.org, PUC-Rio
> require "socket"
```


17. 标准库

17.1. 数学库

17.1.1. 列表

S.N.	函数与功能
1	<code>math.abs(x)</code> 返回 x 的绝对值.
2	<code>math.acos(x)</code> 返回 x 的反余弦值(弧度).
3	<code>math.asin(x)</code> 返回 x 的反正弦值(弧度).
4	<code>math.atan(x)</code> 返回 x 的反正切值(弧度).
5	<code>math.atan2(y, x)</code> 返回 y/x 的反正切值, 使用两个参数的符号查找象限(x 为 0 时也能正确的处理).
6	<code>math.ceil(x)</code> 返回大于或等于 x 的最小整数.
7	<code>math.cos(x)</code> 返回 x 的余弦值(x 以弧度为单位).
8	<code>math.cosh(x)</code> 返回 x 的双曲余弦值.
9	<code>math.deg(x)</code> 返回 x 的角度值(x 为弧度).
10	<code>math.exp(x)</code> 返回 e 的 x 次幂.
11	<code>math.floor(x)</code> 返回小于或等于 x 的最大整数.
12	<code>math.fmod(x, y)</code> 返回 $x\%y$.
13	<code>math.frexp(x)</code> 返回两个值 m, e , 满足 $x = m \cdot 2^e$. 其中, e 是整数, m 的绝对值属于区间 $[0.5, 1)$.
14	<code>math.huge</code> 最大值, 不小于任何其它数值.
15	<code>math.ldexp(m, e)</code> 返回 $m \cdot 2^e$ (e 为整数).
16	<code>math.log(x)</code> 计算自然对数.
17	<code>math.log10(x)</code> 计算以 10 为底的对数.
18	<code>math.max(x, ...)</code> 返回输入参数的最大值.
19	<code>math.min(x, ...)</code> 返回输入参数的最小值.
20	<code>math.modf(x)</code> 返回 x 的整数部分与小数部分.
21	<code>math.pi</code> 数值 PI.
22	<code>math.pow(x, y)</code> 等价于 x^y .
23	<code>math.rad(x)</code> 返回角度 x 的弧度值.
24	<code>math.random([m, [n]])</code> 该函数直接调用 ANSI C 的伪随机生成函数. 1>无参数时, 生成 $[0, 1)$ 区间的均匀分布的随机值;

		2>只传入参数 m 时，函数生成一个位于区间 [1, m] 的均匀分布伪随机值； 3>同时传入参数 m, n 时，生成位于区间 [m, n] 的均匀分布伪随机值。
25	math.randomseed(x)	初始化伪随机数生成器种子值。
26	math.sin(x)	返回 x 的正弦值。
27	math.sinh(x)	返回 x 的双曲正弦值。
28	math.sqrt(x)	返回 x 的平方根。
29	math.tan(x)	返回 x 的正切值。
30	math.tanh(x)	返回 x 的双曲正切值。
31	math.round()	四舍五入

17.1.2. 测试

```
print(math.pi);
print(string.format("%.1f", math.sin(math.pi/2)))
print(string.format("%.1f", math.cos(math.pi/2)))
print(string.format("%.1f", math.tan(math.pi/2)))
print(string.format("%.1f", math.cosh(math.pi/2)))
print(math.deg(math.pi))
```

```
---- Floor
print("Floor of 10.5055 is ", math.floor(10.5055))
---- Ceil
print("Ceil of 10.5055 is ", math.ceil(10.5055))
---- Square root
print("Square root of 16 is ", math.sqrt(16))
---- Power
print("10 power 2 is ", math.pow(10, 2))
---- Absolute
print("Absolute value of -10 is ", math.abs(-10))
----Random
math.randomseed(os.time())
print("Random number between 0 and 1 is ", math.random())
----Random between 1 to 100
print("Random number between 1 and 100 is ", math.random(1, 100))
----Max
print("Maximum in the input array is ", math.max(1, 100, 101, 99, 999))
----Min
print("Minimum in the input array is ", math.min(1, 100, 101, 99, 999))
```

17. 2. string 库

17. 3. os 库

17. 3. 1. 列表

函数	功能
<code>os.clock()</code>	以秒为单位返回程序运行所用 CPU 时间的近似值.
<code>os.date([format[, time]])</code>	返回时间字符串或包含时间的表，时间按指定格式格式化.
<code>os.difftime(t2, t1)</code>	返回从 t1 时刻至 t2 时刻经历的时间.在 POSIX, windows, 及其它某些系统中，该值就是 t2-t1.
<code>os.execute([command])</code>	该函数等价于 ANSI C 中的 system 函数.传递的参数 command 由操作系统的 shell 执行.如果命令成功结束，则返回的第一个值为 true，否则为 nil.
<code>os.exit([code[, close]])</code>	调用 ANSI C 的 exit 函数，结束程序.如果 code 为 true，则返回状态为 EXIT_SUCCESS；若 code 为 false，则返回状态为 EXIT_FAILURE.如果 code 为数值，则返回状态也就为该数值.
<code>os.getenv(varname)</code>	返回进程的环境变量 varname 的值，如果此环境变量没有定义则返回 nil.
<code>os.remove(filename)</code>	删除文件(或 POSIX 系统中的空目录).如果函数失败，则返回 nil 以及描述错误的字符串与错误代码.
<code>os.rename(oldname, newname)</code>	重命名文件或目录.如果函数失败，则返回 nil 以及描述错误的字符串与错误代码.
<code>os.setlocale(locale[, category])</code>	设置程序当前的地区(locale)，locale 是一个与操作系统相关的字符串.category 是一个可选的字符串，描述设置更改的范围，包括：all, collate, ctype, monetary, numeric, time.默认为 all.函数返回新地区的名称，如果函数调用失败则返回 nil.
<code>os.time([table])</code>	无参数时，返回当前时间；传入参数时，则返回指定参数表示的日期和时间.传入的参数必须包含以下的域：年、月、日、时(默认 12)、分(默认 0)、秒(默认 0)、isdst(默认 nil) 四个域是可选的.
<code>os.tmpname()</code>	返回一个可作为临时文件名的字符串.这个临时文件必须显式地打开，使用结束时也必须显式地删

除。

17.3.2. 测试

```
-- 格式化日期
print("The date is ", os.date("%m/%d/%Y"))
-- 日期与时间
print("The date and time is ", os.date())
-- 时间
print("The OS time is ", os.time())
-- 等待一段时间
for i=1, 1000000 do
end
-- Lua 启动的时长
print("Lua started before ", os.clock())

print(os.getenv("PATH"))
```

```
function createDir (dirname)
    os.execute("mkdir" .. dirname)
end
createDir("mydir")
```

17.4. io 库

17.4.1. 简介

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

17.4.2. 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。I/O 库将当前输入文件初始化为进程标准输入(stdin)，将当前输出文件初始化为进程标准输出。在执行 io.read() 操作时，就会从标准输入中读取一行。

用函数 io.input 和 io.output 可以改变这两个当前文件。io.input(filename) 调用会以只读模式打开指定的文件，并将其设定为当前输入文件；除非再次调用 io.input，否则所有的输入都来源于这个文件；在输出方面，io.output 也可以完成类似的工作。

函数名	功能描述
-----	------

<code>io.lines([filename])</code>	打开指定的文件 <code>filename</code> 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 <code>nil</code> ，并自动关闭文件 若不带参数时 <code>io.lines()</code> <=> <code>io</code> 。 <code>input():lines();</code> 读取默认输入设备的内容，但结束时不关闭文件 如： <pre>for line in io.lines("main.lua") do print(line) end</pre>
<code>io.input ([file])</code>	
<code>io.output ([file])</code>	相当于 <code>io.input</code> ，但操作在默认输出文件上
<code>io.read (...)</code>	相当于 <code>io.input():read</code>
<code>io.write (...)</code>	相当于 <code>io.output():write</code>

`io.write`, `io.read` 是一对。默认情况下，他们从 `stdin` 读输入，输出到 `stdout`。
另有两个函数可以改变这一默认行为：`io.input("xx")`, `io.output("yy")` 他们改变输入为某个 `xx` 文件，输出到 `yy` 文件。

5.1	5.3	解释
"*all"	"a"	从当前位置读取整个文件，若为文件尾，则返回空字符串
"*line"	"l"	[默认]读取下一行的内容，若为文件尾，则返回 <code>nil</code>
	"L"	reads the next line (keeping the newline)
"*number"	"n"	读取指定字节数的字符，如果 <code>number</code> 为 0 则返回空字符串，若为文件尾，则返回 <code>nil</code> ；
<num>	num	读取 <code>num</code> 个字符到串

17.4.2.1. 常见测试用例

<0>--默认输入是键盘，输出是屏幕

```
io.write(io.read())          --默认输入是键盘，输出是屏幕
io.write("<", "nzhsoft", ">")
```

<1>读取一个文本的内容，并为其加上行号：

a>判读为 nil

```
io.input("C:\\\\Users\\\\Guilin\\\\Desktop\\\\luapro\\\\test\\\\aa.txt")
io.output("C:\\\\Users\\\\Guilin\\\\Desktop\\\\luapro\\\\test\\\\aaa.txt")
for count= 1, math.huge do
    line = io.read("*line")
    if line == nil then break end
    io.write(string.format("%2d ", count), line, "\\n");
end
```

b >io.lines() 迭代器版本

```
local count = 0
for line in io.lines() do
    count = count + 1
    io.write(string.format("%2d ", count), line, "\\n");
end
```

<2> 读取替换回写

```
t = io.read("*all")           -- read the whole file
t = string.gsub(t, "bad", "good") -- do the job
io.write(t) -- write the file
```

<3> 排序一个文件

```
local lines = {}
-- read the lines in table 'lines'
for line in io.lines() do
    lines[#lines + 1] = line
end
-- sort
table.sort(lines)
-- write all the lines
for _, l in ipairs(lines) do
    io.write(l, "\\n")
end
```

17.4.2.2. io.write 与 print 的不同

print 输出的数据，只能到 stdout，而 io.write 而可以指定文件，print 输出的数据，间隔是 tab，而 io.write 输出的数据，无间隔，print 是以'\\n'作为结束符的，而 io.write 是没有结不符的。

```
print("a", "b", "c");
print("a", "b", "c");

io.output(stdout);
io.write("a", "\\t", "b", "\\t", "c", "\\n");
```

17.4.3. 完全模式

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])
```

mod

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，同时也可创建新文件。
"r+"	读写模式打开已存在的文件。
"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。

17.5. debug 库

17.5.1. 列表

方法	描述
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。
getfenv(object)	返回对象的环境。
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。
getinfo (optional thread, function or stack level, 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。
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。

<code>getmetatable(value)</code>	返回指定对象的元表，如果不存在则返回 nil。
<code>getregistry()</code>	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。
<code>getupvalue (func function, upvalue index)</code>	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。
<code>setfenv (function or thread or userdata, environment table)</code>	将指定的对象的环境设置为 table，即改变对象的作用域。
<code>sethook (optional thread, hook function, hook mask string with "c" and/or "r" and/or "l", optional instruction count)</code>	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。
<code>setlocal (optional thread, stack level, local index, value)</code>	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。
<code>setmetatable(value, metatable)</code>	为指定的对象设置元表，元表可以为 nil。
<code>setupvalue (function, upvalue index, value)</code>	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。
<code>traceback (optional thread, optional message string, optional level argument)</code>	用 traceback 构建扩展错误消息

17.5.2. 测试

18. 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。

18.1. 理论基础

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。

18.2. 接口介绍

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；

●`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。

18.3. 实战操作

```
mytable = {"apple", "orange", "banana"}  
print(collectgarbage("count"))  
mytable = nil  
print(collectgarbage("count"))  
print(collectgarbage("collect"))  
print(collectgarbage("count"))
```

19. Lua 与 C/C++交互栈

当我们想在 **lua** 和 **C** 之间交换数据时，会面临两个问题，第一个问题是动态类型和静态类型体系之间的不匹配，第二个问题是自动内存管理和手动内存管理之间不匹配。

lua 和 **C** 之间的通信的主要组件，是无处不在的虚似栈，几乎所有的 **Api** 都是在操作这个栈中的值，**lua** 与 **C** 之间的所有数据交换都是通过这个栈完成的。此外还可以利用栈保存中间结果。

19.1. 交互机制

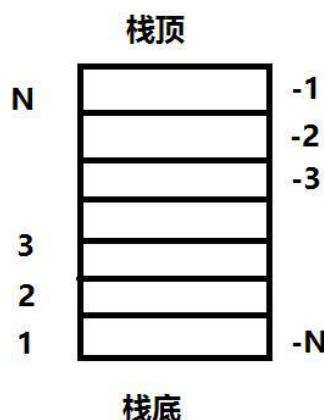
19.1.1. 交互原理

Lua 使用一个虚拟栈来和 **C** 互传值。栈上的每个元素都是一个 **Lua** 值 (**nil**, 数字, 字符串, 等等)。

19.1.2. 交互栈

所有针对栈的 **API** 操作都不严格遵循栈的操作规则(**FILO**)。而是可以用一个索引来指向栈上的任何元素：正的索引指的是栈上的绝对位置(从 1 开始)；负的索引则指从栈顶开始的偏移量。

展开来说，如果堆栈有 **n** 个元素，那么索引 1 表示第一个元素(也就是最先被压栈的元素)而索引 **n** 则指最后一个元素；索引 -1 也是指最后一个元素(即栈顶的元素)，索引 -**n** 是指第一个元素。



19.2. API

19.2.1. push functions (C -> stack)

push 系列函数是要入栈的。

```
void lua_pushnil    (lua_State * L);
void lua_pushboolean (lua_State * L, int bool);
```

```
void lua_pushnumber (lua_State * L, lua_Number n);
void lua_pushinteger (lua_State * L, lua_Integer n);
void lua_pushlstring (lua_State * L, const char * s, size_t len);
void lua_pushstring (lua_State * L, const char * s);
```

lua_push nil for the constant nil,
 lua_push boolean for Booleans (integers, in C),
 lua_push number for doubles,
 lua_push integer for integers,
 lua_push lstring for arbitrary strings (a pointer to char plus a length),
 lua_push string for zero-terminated strings:

19.2.2. set functions (stack -> Lua)

set 系列函数是要出栈的。

```
void (lua_setglobal) (lua_State * L, const char * name);
void (lua_settable) (lua_State * L, int idx);
void (lua_setfield) (lua_State * L, int idx, const char * k);
void (lua_seti) (lua_State * L, int idx, lua_Integer n);
void (lua_rawset) (lua_State * L, int idx);
void (lua_rawseti) (lua_State * L, int idx, lua_Integer n);
void (lua_rawsetp) (lua_State * L, int idx, const void * p);
int (lua_setmetatable) (lua_State * L, int objindex);
void (lua_setuservalue) (lua_State * L, int idx);
```

19.2.3. get functions (Lua -> stack)

get 系列函数是要入栈的。

```
int (lua_getglobal) (lua_State *L, const char *name);
int (lua_gettable) (lua_State *L, int idx);
int (lua_getfield) (lua_State *L, int idx, const char *k);
int (lua_geti) (lua_State *L, int idx, lua_Integer n);
int (lua_rawget) (lua_State *L, int idx);
int (lua_rawgeti) (lua_State *L, int idx, lua_Integer n);
int (lua_rawgetp) (lua_State *L, int idx, const void *p);

void (lua_createtable) (lua_State *L, int narr, int nrec);
void *(lua_newuserdata) (lua_State *L, size_t sz);
int (lua_getmetatable) (lua_State *L, int objindex);
int (lua_getuservalue) (lua_State *L, int idx);
```

19.2.4. access functions (stack -> C)不出栈

_is 和 _to 系列，不会出栈。是否出栈的问题，由我们学习的栈管理 api 来完成操作。也就是 16.2.5 要讲的栈管理 api。

lua_is* does not check whether the value has that specific type, but whether the value can be converted to that type。

```

int          (lua_isnumber) (lua_State *L, int idx);
int          (lua_isstring) (lua_State *L, int idx);
int          (lua_iscfunction) (lua_State *L, int idx);
int          (lua_isinteger) (lua_State *L, int idx);
int          (lua_isuserdata) (lua_State *L, int idx);
int          (lua_type) (lua_State *L, int idx);
const char  *(lua_typename) (lua_State *L, int tp);

lua_Number   (lua_tonumberx) (lua_State *L, int idx, int *isnum);
lua_Integer  (lua_tointegerx) (lua_State *L, int idx, int *isnum);
int          (lua_toboolean) (lua_State *L, int idx);
const char  *(lua_tolstring) (lua_State *L, int idx, size_t *len);
size_t        (lua_rawlen) (lua_State *L, int idx);
lua_CFunction (lua_tofunction) (lua_State *L, int idx);
void         * (lua_touserdata) (lua_State *L, int idx);
lua_State    *(lua_tothread) (lua_State *L, int idx);
const void   *(lua_topointer) (lua_State *L, int idx);

lua_check_*

```

```
lua_Number luaL_checknumber (lua_State *L, int arg);
```

检查函数的第 `arg` 个参数是否是一个 数字，并返回这个数字。

```
const char *luaL_checkstring (lua_State *L, int arg);
```

检查函数的第 `arg` 个参数是否是一个 字符串并返回这个字符串。这个函数使用 `lua_tolstring` 来获取结果。所以该函数有可能引发的转换都同样有效。

19.2.5. 栈管理

19.2.5.1. 常用接口

```

/* int lua_gettop(lua_State *L)
 * 返回栈顶元素的索引。
 * 因为栈中元素的索引是从 1 开始编号的，所以函数的返回值相当于栈中元素的个数。
 * 返回值为 0 表示栈为空。
 */

```

```

/* void lua_settop(lua_State *L, int index)
 * 设置栈顶为索引"index"指向处。
 * 如果"index"比"lua_gettop()"的值大，那么多出的新元素将被赋值为"nil"。
 */
Lua_settop(L, 0); // 清空栈。

```

```

/* void lua_pushvalue (lua_State *L, int index);
 * 将索引"index"处元素，压到栈顶。（把栈上给定索引处的元素作一个副本压栈）
 */

```

```
/* void lua_replace(lua_State *L, int index)
 * 将栈顶元素移动到索引"index"处。(栈顶出栈, 覆盖了索引"index"处的元素)
 */
```

```
/* void lua_rotate (lua_State *L, int index, int n);
 * 把从 idx 开始到栈顶的元素 轮转 n 个位置。对于 n 为正数时, 轮转方向是向栈顶的; 当 n
 * 为负数时, 向栈底方向轮转 -n 个位置。n 的绝对值不可以比参于轮转的切片长度大。
 */
```

19.2.5.2. 常用宏

```
#define lua_pop(L,n)           lua_settop(L, -(n) - 1)
```

```
#define lua_remove(L,idx)      (lua_rotate(L, (idx), -1), lua_pop(L, 1))
/* void lua_remove(lua_State *L, int index)
 * 移除 栈中索引"index"处的元素, 该元素之上的所有元素下移。
 * 从给定有效索引处移除一个元素, 把这个索引之上的所有元素移下来填补上这个空隙
 */
```

```
#define lua_insert(L,idx)      lua_rotate(L, (idx), 1)
/* void lua_insert(lua_State *L, int index)
 * 将栈顶元素移动到索引"index"处, 索引"index"(含)之上的所有元素上移。
 */
```

为防止进栈溢出, 有如下函数可用

```
int lua_checkstack (lua_State *L, int sz);
void luaL_checkstack (lua_State *L, int sz, const char *msg);
```

19.3. 测试

19.3.1. 打印栈

19.3.1.1. lua 基本数据类型

```
#define LUA_TBOOLEAN          1
#define LUA_TLIGHTUSERDATA     2
#define LUA_TNUMBER            3
#define LUA_TSTRING             4
#define LUA_TTABLE              5
#define LUA_TFUNCTION           6
#define LUA_TUSERDATA           7
#define LUA_TTHREAD              8
```

19.3.1.2. stackDump

```
static void stackDump(lua_State* L){
```

```

static int count = 0;
printf("begin dump lua stack %d\n", count);
int i = 0;
int top = lua_gettop(L);
for (i = 1; i <= top; ++i) // for (i = top; i >= 1; --i)
{
    int t = lua_type(L, i);
    switch (t)
    {
        case LUA_TSTRING:
            printf("%s ", lua_tostring(L, i));
            break;
        case LUA_TBOOLEAN:
            printf(lua_toboolean(L, i) ? "true " : "false ");
            break;
        case LUA_TNUMBER:
            printf("%g ", lua_tonumber(L, i));
            break;
        default:
            printf("%s ", lua_typename(L, t));
            break;
    }
}
printf("\n end dump lua stack %d \n", count++);
}

```

19.3.2. 测试

```

#include "lua.h"
#include "lualib.h"
#include "lauxlib.h"

int main()
{
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);
    stackDump(L);
    lua_pushinteger(L, 1);
    lua_pushinteger(L, 2);
    lua_pushinteger(L, 3);
    lua_pushinteger(L, 4);
    stackDump(L);

//    int n = lua_gettop(L);
//    printf("stack size = %d\n", n);

//    lua_settop(L, 2);
}

```

```
// stackDump(L);

// lua_pop(L, 2);
// lua_pop(L, -1);
// stackDump(L);

// lua_pushvalue(L, 1);
//// lua_pushvalue(L, 100);
// stackDump(L);

// lua_remove(L, 200);
// stackDump(L);
lua_insert(L, 1);
stackDump(L);

// lua_replace(L, 2);
// stackDump(L);
luaL_dofile(L, "xx.lua");
lua_close(L);
}
```

19.3.3. test

以下操作，对于虚拟栈有何影响？尝试去分析

```
lua_settop(L, -1);           /* set top to its current value */
lua_insert(L, -1);          /* move top element to the top */
lua_replace(L, -1);         /* replace top element by the top element */
lua_rotate(L, x, 0);        /* rotates by zero positions */
```

19.4. Lua API 及译文

19.4.1. 官方原文：

<https://www.lua.org/manual/5.3/#index>

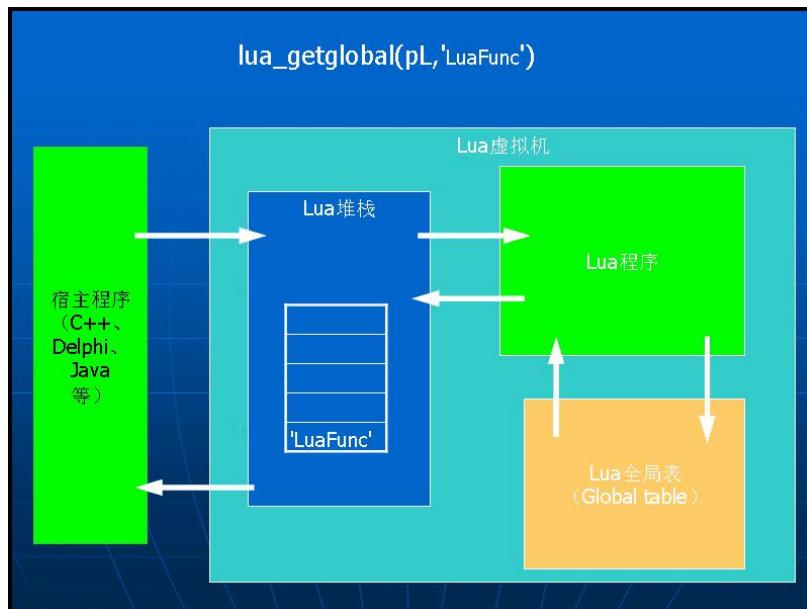
19.4.2. 云风译文：

<http://cloudwu.github.io/lua53doc/contents.html#index>

20. Lua 扩展 C 程序(C/C++->Lua)

Lua 是一种嵌入式语言(embedded language), 这意味着 lua 并不是一个独立运行的应用, 而是一个库, 它可以链接到其它的应用程序中, 将 lua 的功能融入到这些应用。

20.1. 宿主 / 栈 / lua



20.2. 访问 lua 全局变量

20.2.1. API

20.2.1.1. getglobal

```
int lua_getglobal (lua_State *L, const char *name);
```

指将 lua 全局作用域中变量名为 name 的成员压入虚拟栈中。 **Pushes** onto the stack the value of the global name. Returns the type of that value.

20.2.2. 测试

20.2.2.1. xx.lua

```
price = 69999
teacher = "wgl"
org = "nzhsoft"
```

20.2.2.2. main.c

```
#include "lua.h"
#include "lualib.h"
```

```
#include "lauxlib.h" --auxiliary
int main()
{
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);
    luaL_dofile(L, "xx.lua");

    lua_getglobal(L, "price");
    int price = lua_tointeger(L, -1);

    lua_getglobal(L, "teacher");
    char *teacher = lua_tostring(L, -1);

    printf("price = %d teacher %s\n", price, teacher);

    lua_pop(L, 2);
    return 0;
}
```

20.3. 访问 lua 全局表字段

20.3.1. api

20.3.1.1. gettable

```
void lua_gettable (lua_State *L, int index);
```

把 t[k] 值压入堆栈，这里的 t 是指有效索引 index 指向的值，而 k 则是栈顶放的值。这个函数会弹出堆栈上的 key，把结果放在栈上原来 key 的相同位置。

Pushes onto the stack the value t[k], where t is the value at the given index and k is the value at the top of the stack。

This function pops the key from the stack, pushing the resulting value in its place

获取 table 元素：

- ① 将元素的 key 压入到栈中，用 lua_gettable(Lua_state, index)
- ② 对于字符串索引，可以用 lua_getfield(Lua_state, index, key) 来直接获取，

20.3.1.2. lua_getfield

```
int lua_getfield (lua_State *L, int index, const char *k);
```

Pushes onto the stack the value t[k], where t is the value at the given index. Returns the type of the pushed value.

如:lua_getfield(stack, -1, "loaded"); 等价于 lua_pushstring(L, "loaded") lua_gettable(L, -2);

20.3.2. 测试

20.3.2.1. xx.lua

```
nhsoft = {price = 69999, teacher = "wgl"}
```

20.3.2.2. main.c

```
#include "lua.h"
#include "lualib.h"
#include "lauxlib.h"
#include "math.h"

int main()
{
    lua_State * L = luaL_newstate();
    luaL_openlibs(L);
    luaL_dofile(L, "xx.lua");
    lua_getglobal(L, "nzhsoft");           // -1
//    lua_pushstring(L, "price");          // -1 -2
//    lua_gettable(L, -2);
    lua_getfield(L, -1, "price");         // 将 k=price 替换成了 v = 69999
    int price = lua_tonumber(L, -1);

    printf("%d\n", price);

    lua_settop(L, 1);                   // -1
//    lua_pushstring(L, "teacher");      // -1 -2
//    lua_gettable(L, -2);
    lua_getfield(L, -1, "teacher");
    char * teacher = lua_tostring(L, -1);
    printf("%s\n", teacher);
    return 0;
}
```

20.3.3. cheatsheet

```
lua_gettable
lua_getglobal(L, "mytable") <== push mytable
lua_pushnumber(L, 1)           <== push key 1
lua_gettable(L, -2)           <== pop key 1, push mytable[1]

lua_settable
lua_getglobal(L, "mytable") <== push mytable
lua_pushnumber(L, 1)           <== push key 1
lua_pushstring(L, "abc")       <== push value "abc"
lua_settable(L, -3)           <== mytable[1] = "abc", pop key & value

lua_rawget:
用法同 lua_gettable, 但更快(因为当 key 不存在时不用访问元方法 __index)

lua_rawset:
用法同 lua_settable, 但更快(因为当 key 不存在时不用访问元方法 __newindex)

lua_rawgeti 必须为数值键
lua_getglobal(L, "mytable") <== push mytable
```

```

lua_rawget(L, -1, 1)      <== push mytable[1], 作用同下面两行调用
--lua_pushnumber(L, 1)    <== push key 1
--lua_rawget(L,-2)        <== pop key 1, push mytable[1]

lua_rawseti 必须为数值键
lua_getglobal(L, "mytable") <== push mytable
lua_pushstring(L, "abc")   <== push value "abc"
lua_rawseti(L, -2, 1)     <== mytable[1] = "abc", pop value "abc"

lua_getfield 必须为字符串键
lua_getglobal(L, "mytable") <== push mytable
lua_getfield(L, -1, "x")   <== push mytable["x"], 作用同下面两行调用
--lua_pushstring(L, "x")   <== push key "x"
--lua_gettable(L,-2)       <== pop key "x", push mytable["x"]

lua_setfield 必须为字符串键
lua_getglobal(L, "mytable") <== push mytable
lua_pushstring(L, "abc")   <== push value "abc"
lua_setfield(L, -2, "x")   <== mytable["x"] = "abc", pop value "abc"

```

20.4. 访问 lua 全局函数

20.4.1. Api

20.4.1.1. luaL_dofile

```
int luaL_dofile (lua_State *L, const char *filename);
```

加载并运行文件, luaL_dofile 的本质是个宏定义,(luaL_loadfile(L, filename) || luaL_pcall(L, 0, LUA_MULTRET, 0))。It returns false if there are no errors or true in case of errors。

luaL_dofile 函数执行这个 lua 脚本源码时会有两个阶段, 第一个是将脚本加载进内存, 分词解析并生成字节码并将其整个包裹为 main chunk 放于 lua stack 栈顶, 第二是调用 luaL_pcall 执行这个 chunk。

20.4.1.2. luaL_loadfile

luaL_loadfile 这个函数, 调用 load 函数, 加载文件并编译文件为 lua 的 chunk, 然后将其推到栈顶。

```
int luaL_loadfile (lua_State *L, const char *filename);
```

Loads a file as a Lua chunk。This function uses luaL_load to load the chunk in the file named filename。If filename is NULL, then it loads from the standard input。

```
int luaL_load(lua_State *L,
              lua_Reader reader,
              void *data,
              const char *chunkname,
              const char *mode);
```

加载一段 Lua 代码块, 但不运行它。如果没有错误, luaL_load 把一个编译好的代码块作为一个 Lua 函数压到栈顶。否则, 压入错误消息。

Loads a Lua chunk without running it。If there are no errors, luaL_load pushes the **compiled chunk** as a Lua function on top of the stack。Otherwise, it pushes an error message。

20.4.1.3. lua_pcall

```
int lua_pcall (lua_State *L, int nargs, int nresults, int msgh);
```

Calls a function in protected mode。安全模式下的，`lua_call`。通常，最后一个参数填零。If `msgh` is 0, then the error object returned on the stack is exactly the original error object。

```
void lua_call (lua_State *L, int nargs, int nresults);
```

要想调用一个 `lua` 函数，流程如下，第一，函数必须被压栈，第二，依次序压入参数，调用 `lua_call`，`nargs` 是入栈参数的个数。第三，函数调用后，函数及其参数要出栈，第四，返回值入栈，入栈顺序，先入栈者底，后入栈者顶，`nresults` 填入返回值的个数。

To call a function you must use the following protocol: first, the function to be called is pushed onto the stack; then, the arguments to the function are pushed in direct order; that is, the first argument is pushed first. Finally you call `lua_call`; `nargs` is the number of arguments that you pushed onto the stack.

All arguments and the function value are popped from the stack when the function is called。The function results are pushed onto the stack when the function returns。The function results are pushed onto the stack in direct order (the first result is pushed first), so that after the call the last result is on the top of the stack。

lua code

```
a = f("how", t.x, 14)
```

C code

```
lua_getglobal(L, "f");           /* function to be called */
lua_pushliteral(L, "how");       /* 1st argument */
lua_getglobal(L, "t");           /* table to be indexed */
lua_getfield(L, -1, "x");        /* push result of t.x (2nd arg) */
lua_remove(L, -2);              /* remove 't' from the stack */
lua_pushinteger(L, 14);          /* 3rd argument */
lua_call(L, 3, 1);               /* call 'f' with 3 arguments and 1 result */
lua_setglobal(L, "a");           /* set global 'a' */
```

20.4.2. 调用 `lua` 函数-无参无返回

20.4.2.1. xx.lua

```
print "in xx.lua"
function func()
    print("func test")
end
```

20.4.2.2. main.c

```
#include "lua.h"
#include "lualib.h"
#include "lauxlib.h"

int main()
```

```
{
    lua_State *L = luaL_newstate();
    luaL_openlibs(L); //若无此，print 函数不可用。
    luaL_dofile(L, "xx.lua");

    lua_getglobal(L, "func");
    lua_pcall(L, 0, 0, 0);
    return 0;
}
```

20.4.3. 调用 lua 函数-有参无返回

参数的压栈顺序，即函数参数从左往右的顺序。

20.4.3.1. xx.lua

```
print "in xx.lua"
function func( i, f, str)
    print("func test")
    print(i)
    print(f)
    print(str)
end
```

20.4.3.2. main.c

```
int main()
{
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);
    luaL_dofile(L, "xx.lua");

    lua_getglobal(L, "func");
    lua_pushinteger(L, 100);
    lua_pushnumber(L, 3.14);
    lua_pushstring(L, "nzhsoft");

    lua_pcall(L, 3, 0, 0);
    return 0;
}
```

20.4.4. 调用 lua 函数-有参有返回

lua 返回值，按从左往右的顺序，依次入栈。

20.4.4.1. xx.lua

```
print "in xx.lua"
function func ( i, f, str)
    print("func test")
```

```

print(i)
print(f)
print(str)
return 100,200,300
end

```

20.4.4.2. main.c

```

#include "lua.h"
#include "lualib.h"
#include "lauxlib.h"

int main()
{
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);
    luaL_dofile(L, "xx.lua");

    lua_getglobal(L, "func");
    lua_pushinteger(L, 100);
    lua_pushnumber(L, 3.14);
    lua_pushstring(L, "nzhsoft");

    lua_pcall(L, 3, 3, 0);

    int i = lua_tointeger(L, -1);
    printf("i = %d\n", i);
    i = lua_tointeger(L, -2);
    printf("i = %d\n", i);
    i = lua_tointeger(L, -3);
    printf("i = %d\n", i);
    return 0;
}

```

20.4.5. 调用 lua 表字段函数

20.4.5.1. xx.lua

```

print "in xx.lua"

tab = {
    func = function (a,b,c)
        print(a,b,c)
        return a*10,b*10,c*10
    end
}

```

20.4.5.2. main.c

```

int main()

```

```
{
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    luaL_dofile(L, "xx.lua");

    luaL_getglobal(L, "tab");
    luaL_getfield(L, -1, "func");
    luaL_remove(L, -2);

    luaL_pushinteger(L, 1);
    luaL_pushinteger(L, 2);
    luaL_pushinteger(L, 3);
    luaL_pcall(L, 3, 3, 0);

    for(int i=0; i<3; i++)
    {
        int t = luaL_tointeger(L, 3-i);
        printf("%-8d", t);
    }
    luaL_pop(L, 3);

    luaL_close(L);
}
```

20.4.6. 自制 lua 编译器

20.4.6.1. luaL_dosstring

int luaL_dosstring (lua_State *L, const char *str); Loads and runs the given string。 It is defined as the following macro:

```
(luaL_loadstring(L, str) || luaL_pcall(L, 0, LUA_MULTRET, 0))
```

It returns false if there are no errors or true in case of errors。

20.4.6.2. 自实现

```
#include <stdio.h>
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>

int main()
{
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);
```

```

char buf[1024];

while(fgets(buf, 1024, stdin))
{
//    luaL_dostring(L, buf);
    luaL_loadstring(L, buf);
    lua_pcall(L, 0, 0, 0);
}
lua_close(L);
return 0;
}

```

20.4.6.3. lua 编译器源码解析

```

int main (int argc, char **argv)
{
    int status, result;
    lua_State *L = luaL_newstate(); /* create state */
    if (L == NULL) {
        l_message(argv[0], "cannot create state: not enough memory");
        return EXIT_FAILURE;
    }
    luaL_pushcfunction(L, &pmain); /* to call 'pmain' in protected mode */
    luaL_pushinteger(L, argc); /* 1st argument */
    luaL_pushlightuserdata(L, argv); /* 2nd argument */
    status = lua_pcall(L, 2, 1, 0); /* do the call */
    result = lua_toboolean(L, -1); /* get result */
    report(L, status);
    lua_close(L);
    return (result && status == LUA_OK) ? EXIT_SUCCESS : EXIT_FAILURE;
}

```

```

static int pmain (lua_State *L) {
    int argc = (int)lua_tointeger(L, 1);
    char **argv = (char **)lua_touserdata(L, 2);
    int script;
    int args = collectargs(argv, &script);
    luaL_checkversion(L); /* check that interpreter has correct version */
    if (argc[0] && argc[0][0]) progname = argv[0];
    if (args == has_error) { /* bad arg? */
        print_usage(argv[script]); /* 'script' has index of bad arg. */
        return 0;
    }
    if (args & has_v) /* option '-v'? */
        print_version();
    if (args & has_E) { /* option '-E'? */

```

```

        lua_pushboolean(L, 1); /* signal for libraries to ignore env. vars. */
        lua_setfield(L, LUA_REGISTRYINDEX, "LUA_NOENV");
    }
    luaL_openlibs(L); /* open standard libraries */
    createargtable(L, argv, argc, script); /* create table 'arg' */
    if (!(args & has_E)) { /* no option '-E'? */
        if (handle_luainit(L) != LUA_OK) /* run LUA_INIT */
            return 0; /* error running LUA_INIT */
    }
    if (!runargs(L, argv, script)) /* execute arguments -e and -l */
        return 0; /* something failed */
    if (script < argc && /* execute main script (if there is one) */
        handle_script(L, argv + script) != LUA_OK)
        return 0;
    if (args & has_i) /* -i option? */
        doREPL(L); /* do read-eval-print loop */
    else if (script == argc && !(args & (has_e | has_v))) { /* no arguments? */
        if (lua_stdin_is_tty()) { /* running in interactive mode? */
            print_version();
            doREPL(L); /* do Read-Eval-Print Loop */
        }
        else dofile(L, NULL); /* executes stdin as a file */
    }
    lua_pushboolean(L, 1); /* signal no errors */
    return 1;
}

```

```

static void doREPL (lua_State *L) {
    int status;
    const char *oldprogname = progname;
    progname = NULL; /* no 'progname' on errors in interactive mode */
    while ((status = loadline(L)) != -1) {
        if (status == LUA_OK)
            status = docal1(L, 0, LUA_MULTRET);
        if (status == LUA_OK) l_print(L);
        else report(L, status);
    }
    lua_settop(L, 0); /* clear stack */
    lua_writeline();
    progname = oldprogname;
}

```

```

/*
** Prompt the user, read a line, and push it into the Lua stack.
*/
static int pushline (lua_State *L, int firstline) {
    char buffer[LUA_MAXINPUT];
    char *b = buffer;
    size_t l;
    const char *prmt = get_prompt(L, firstline);
    int readstatus = lua_readline(L, b, prmt);
    if (readstatus == 0)
        return 0; /* no input (prompt will be popped by caller) */
    lua_pop(L, 1); /* remove prompt */
    l = strlen(b);
    if (l > 0 && b[l-1] == '\n') /* line ends with newline? */
        b[--l] = '\0'; /* remove it */
    if (firstline && b[0] == '=') /* for compatibility with 5.2, ... */
        lua_pushfstring(L, "return %s", b + 1); /* change '=' to 'return' */
    else
        lua_pushlstring(L, b, l);
    lua_freeline(L, b);
    return 1;
}

```

```

#define lua_readline(L, b, p) \
    ((void)L, fputs(p, stdout), fflush(stdout), /* show prompt */ \
     fgets(b, LUA_MAXINPUT, stdin) != NULL) /* get line */

```

```

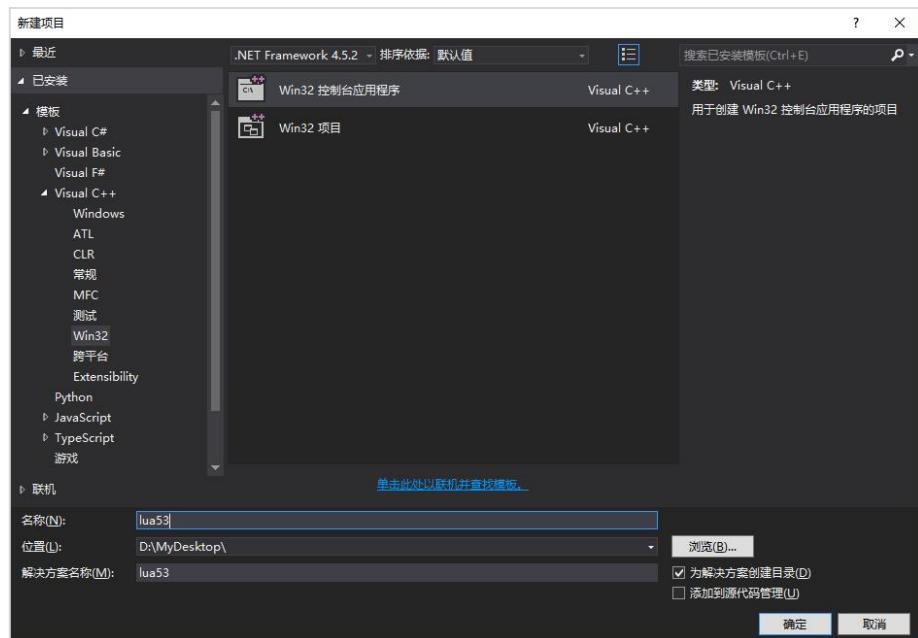
/*
** Interface to 'lua_pcall', which sets appropriate message function
** and C-signal handler. Used to run all chunks.
*/
static int docall (lua_State *L, int narg, int nres) {
    int status;
    int base = lua_gettop(L) - narg; /* function index */
    lua_pushcfunction(L, msghandler); /* push message handler */
    lua_insert(L, base); /* put it under function and args */
    globalL = L; /* to be available to 'laction' */
    signal(SIGINT, laction); /* set C-signal handler */
    status = lua_pcall(L, narg, nres, base);
    signal(SIGINT, SIG_DFL); /* reset C-signal handler */
    lua_remove(L, base); /* remove message handler from the stack */
    return status;
}

```

20.5. 编译 lua5.3 静态库

20.5.1. VS 静态库工程

20.5.1.1. 新建 lua53 工程

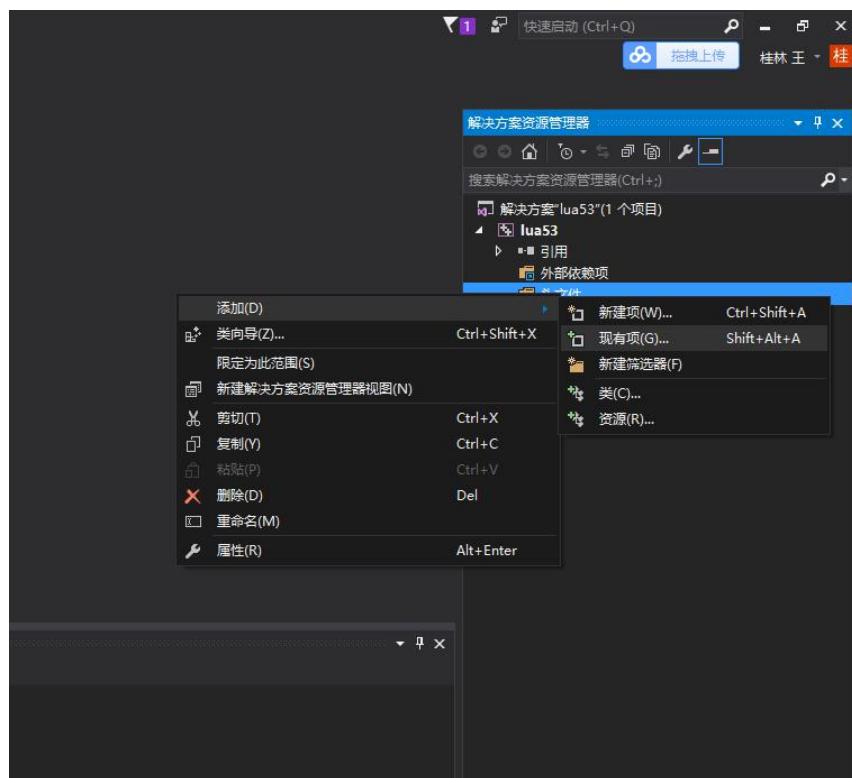


相关配置



然后点击完成

20.5.1.2. 添加.h 与.cpp 文件



名称	修改日期	类型	大小
Makefile	2015/5/27 19:10	文件	7 KB
lua.hpp	2004/12/23 8:53	hpp 文件	1 KB
lzio.h	2015/9/8 23:41	C++ Header file	2 KB
lvm.h	2016/12/22 21:08	C++ Header file	4 KB
lundump.h	2015/9/8 23:41	C++ Header file	1 KB
lualib.h	2017/1/13 1:14	C++ Header file	2 KB
luconf.h	2016/12/22 21:08	C++ Header file	21 KB
lua.h	2016/12/22 23:51	C++ Header file	15 KB
ltm.h	2016/2/27 3:20	C++ Header file	2 KB
itable.h	2016/12/22 21:08	C++ Header file	3 KB
lstring.h	2015/11/3 23:36	C++ Header file	2 KB
lstate.h	2016/12/22 21:08	C++ Header file	8 KB
lprefix.h	2014/12/30 0:54	C++ Header file	1 KB
lparser.h	2015/12/31 2:16	C++ Header file	5 KB
lopcodes.h	2016/7/20 1:12	C++ Header file	9 KB
lobject.h	2016/8/2 3:51	C++ Header file	15 KB
lmem.h	2014/12/20 1:26	C++ Header file	3 KB
llimits.h	2015/11/20 3:16	C++ Header file	8 KB
llex.h	2016/5/2 22:02	C++ Header file	3 KB
lgc.h	2015/12/21 21:02	C++ Header file	5 KB
lfunc.h	2015/1/13 23:49	C++ Header file	2 KB
ldo.h	2015/12/21 21:08	C++ Header file	3 KB
ldebug.h	2015/5/23 1:45	C++ Header file	2 KB
lctype.h	2011/7/15 20:50	C++ Header file	2 KB
lcode.h	2016/1/6 0:22	C++ Header file	4 KB
lauxlib.h	2016/12/6 22:54	C++ Header file	9 KB
lapi.h	2015/3/7 3:49	C++ Header file	1 KB
lzio.c	2015/9/8 23:41	C Source file	2 KB
lvm.c	2016/2/6 3:59	C Source file	44 KB
lutf8lib.c	2016/12/22 21:08	C Source file	7 KB
lundump.c	2015/11/3 0:09	C Source file	7 KB
luac.c	2015/3/12 9:58	C Source file	11 KB
luac.c	2017/1/13 1:14	C Source file	18 KB

名称	修改日期	类型	大小
lapi.c	2016/2/29 22:27	C Source file	31 KB
lauxlib.c	2016/12/21 2:37	C Source file	30 KB
lbaselib.c	2016/9/6 3:06	C Source file	14 KB
lbitlib.c	2015/11/12 3:08	C Source file	5 KB
lcode.c	2016/12/22 21:08	C Source file	34 KB
lcordolib.c	2016/4/12 3:19	C Source file	4 KB
lctype.c	2014/11/3 3:19	C Source file	3 KB
ldblib.c	2015/11/23 19:29	C Source file	13 KB
ldebug.c	2016/10/19 20:32	C Source file	20 KB
ldo.c	2016/12/13 23:52	C Source file	25 KB
ldump.c	2015/10/8 23:53	C Source file	5 KB
lfunc.c	2014/11/3 3:19	C Source file	4 KB
lgc.c	2016/12/22 21:08	C Source file	36 KB
linit.c	2016/12/5 4:17	C Source file	2 KB
liolib.c	2016/12/21 2:37	C Source file	20 KB
llex.c	2016/5/2 22:02	C Source file	16 KB
lmathlib.c	2016/12/22 21:08	C Source file	10 KB
lmem.c	2015/3/7 3:45	C Source file	3 KB
loadlib.c	2017/1/13 1:14	C Source file	24 KB
lobject.c	2016/12/22 21:08	C Source file	17 KB
lopcodes.c	2015/1/5 21:48	C Source file	4 KB
loslib.c	2016/7/19 1:58	C Source file	11 KB
lparser.c	2016/8/2 3:51	C Source file	46 KB
lstate.c	2015/11/13 20:16	C Source file	9 KB
lstring.c	2015/11/23 19:32	C Source file	7 KB
lstrlib.c	2016/12/22 21:08	C Source file	47 KB
itable.c	2016/11/7 20:38	C Source file	20 KB
ltablib.c	2016/2/26 3:41	C Source file	14 KB
ltm.c	2016/12/22 21:08	C Source file	5 KB
lua.c	2017/1/13 1:14	C Source file	18 KB
luac.c	2015/3/12 9:58	C Source file	11 KB
lundump.c	2015/11/3 0:09	C Source file	7 KB
lutf8lib.c	2016/12/22 21:08	C Source file	7 KB
lvm.c	2016/2/6 3:59	C Source file	44 KB
lzio.c	2015/9/8 23:41	C Source file	2 KB
lapi.h	2015/3/7 3:49	C++ Header file	1 KB
lauxlib.h	2016/12/6 22:54	C++ Header file	0 KB

20.5.1.3. 生成解决方案

两个版本分别生成



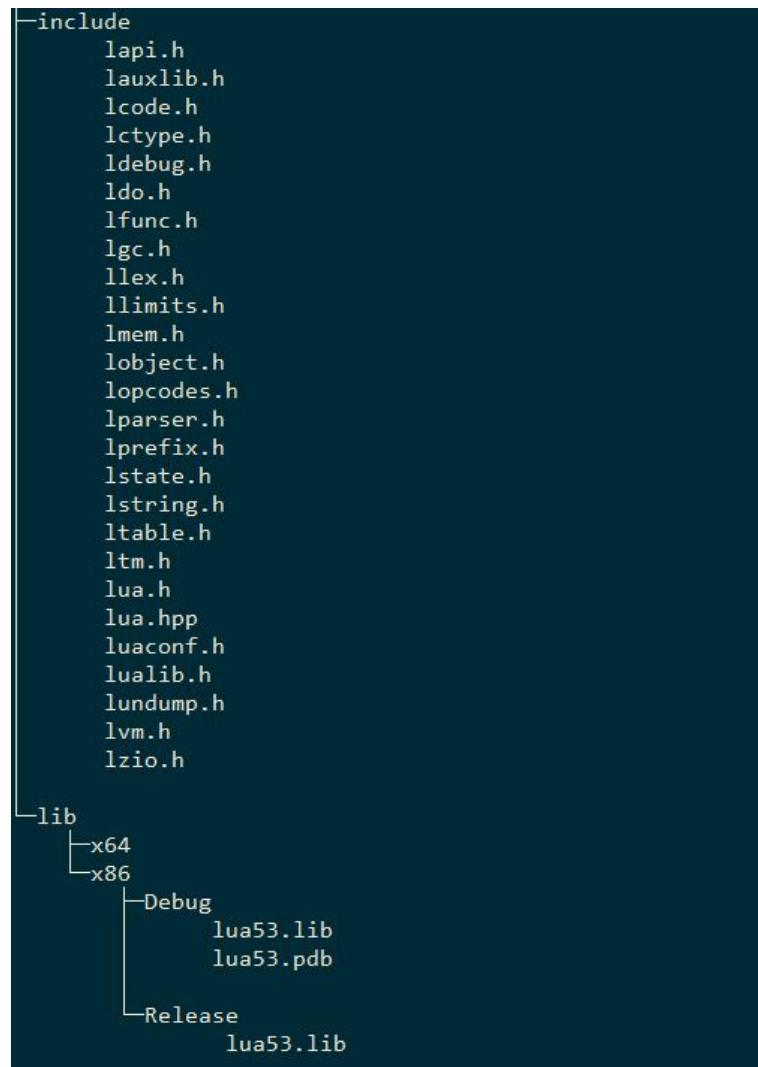
生成结果

lua53 > Debug				
	名称	修改日期	类型	大小
	lua53.lib	2018/10/13 7:58	360压缩	1,267 KB
	lua53.pdb	2018/10/13 7:55	Program Debug...	172 KB

lua53 > Release				
	名称	修改日期	类型	大小
	lua53.lib	2018/10/13 7:55	360压缩	2,208 KB

20.5.1.4. 组织库结构

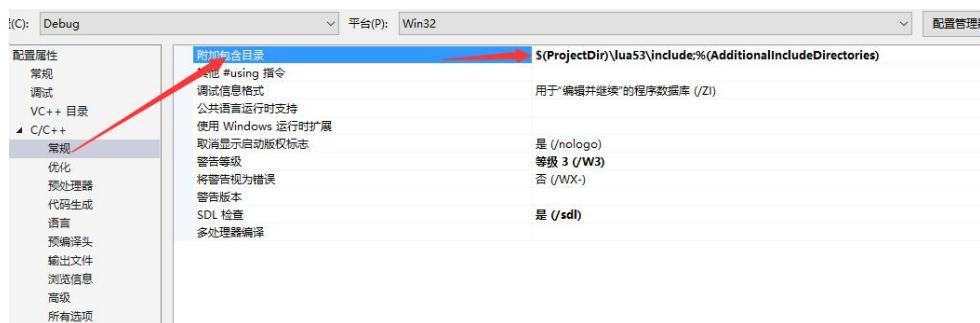
一个三方静态库应该包含，头文件与静态库，我们设置文件夹 lua53 目录结构如下：



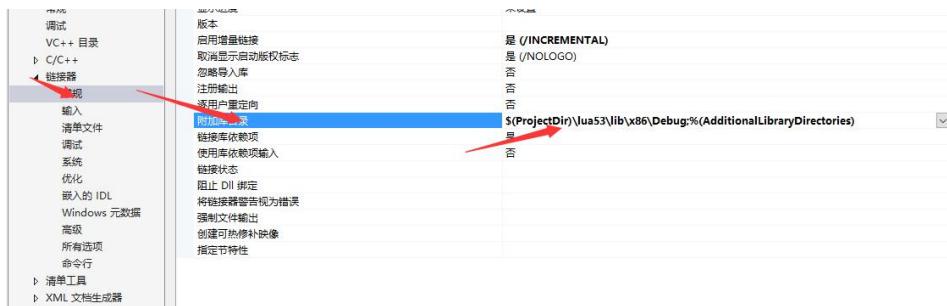
20.5.1.5. vs 工程应用

将 lua53 文件夹，拷贝至 32 位的工程路径中。

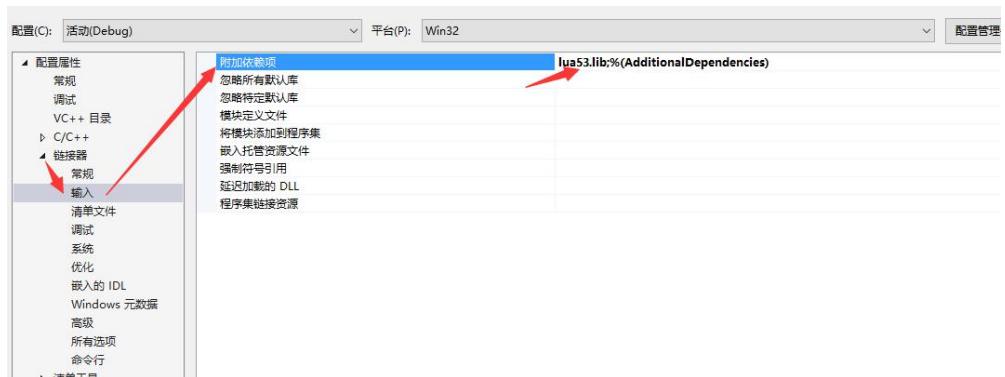
1, 添加头文件搜索目录：



2, 添加库目录：



3. 添加链接库名



4. 编译运行

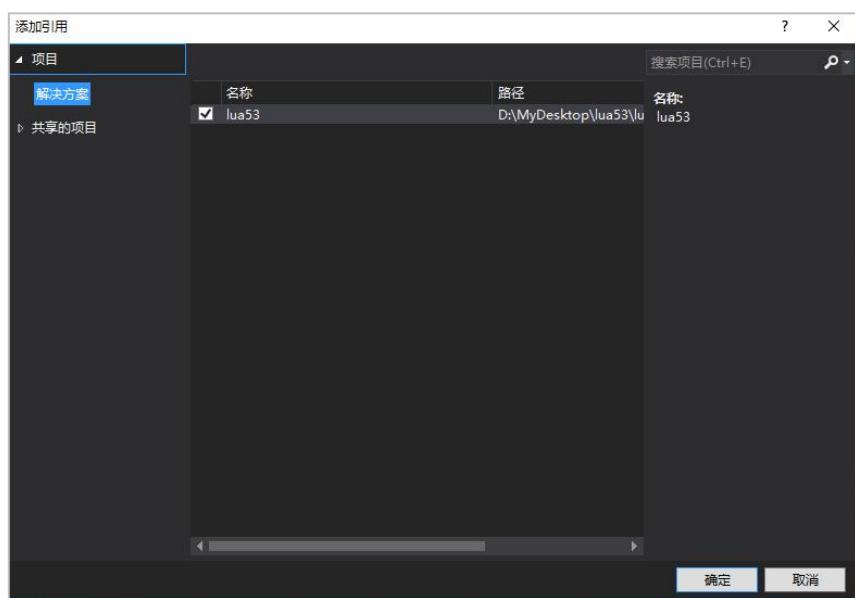
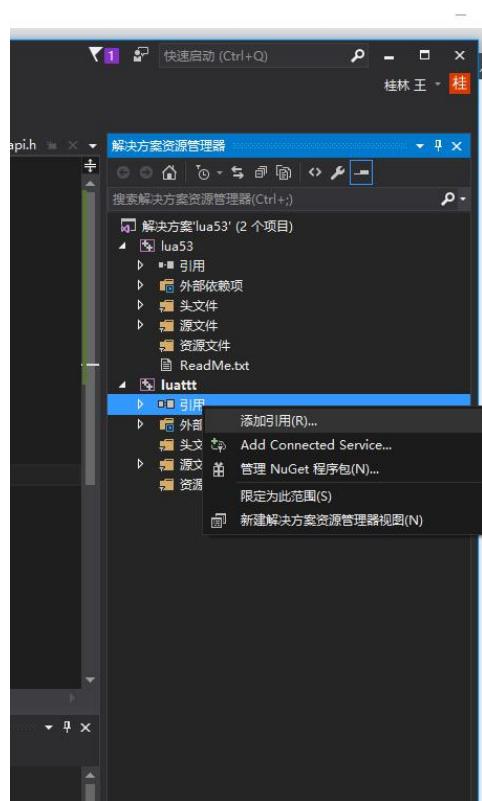
```
#include <iostream>
#include "lua.hpp"

using namespace std;

int main()
{
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);
    const char *buf = "print('lua: Hello World')";
    luaL_dostring(L, buf);
    lua_close(L);
   getc(stdin);
    return 0;
}
```

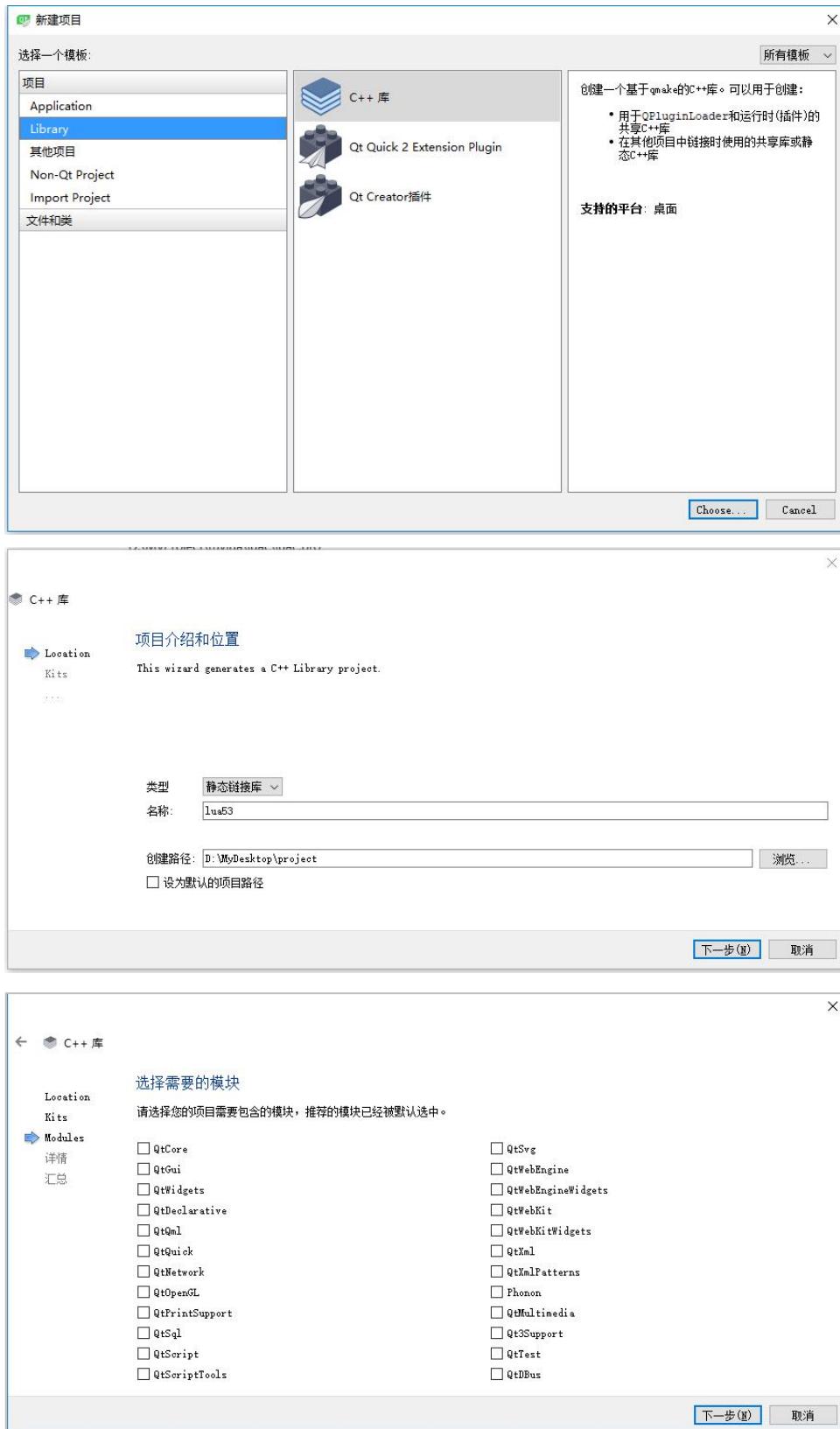
注：

或是完成第1步，添加头文件以后，添加同一解决方案下的lua库工程，



20.5.2. Qt 编译静态工程

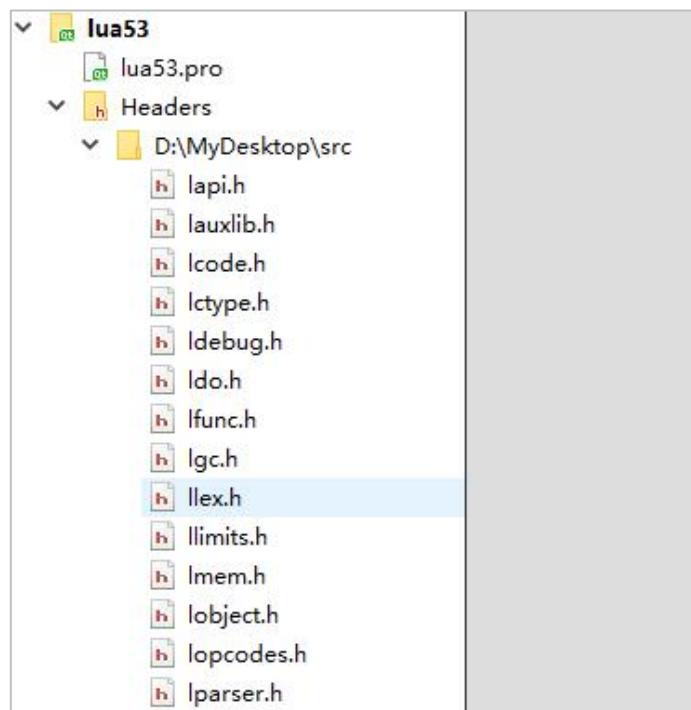
20.5.2.1. 新建 lua53 工程

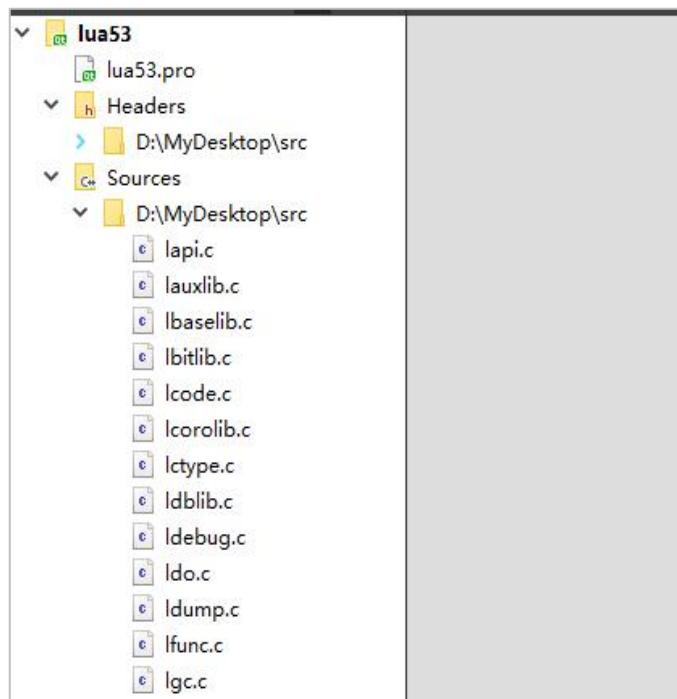




20.5.2.2. 添加.h 和.cpp

添加.h 和.cpp 的时候，将工程生成的文件删除





20.5.2.3. 编译生成

依据 linux 库的命名规则，库名要加前缀 lib, 工程命叫 lua53, 库名即为 liblua53.a, 后缀 a 表示静态库。

project > build-lua53-Desktop_Qt_5_11_1_MinGW_32bit-Debug > debug			
名称	修改日期	类型	大小
lapi.o	2018/10/13 10:54	O 文件	42 KB
lauxlib.o	2018/10/13 10:54	O 文件	31 KB
lbaselib.o	2018/10/13 10:54	O 文件	19 KB
lbitlib.o	2018/10/13 10:54	O 文件	3 KB
lcode.o	2018/10/13 10:54	O 文件	34 KB
lcorolib.o	2018/10/13 10:54	O 文件	8 KB
lctype.o	2018/10/13 10:54	O 文件	2 KB
ldblib.o	2018/10/13 10:54	O 文件	19 KB
ldebug.o	2018/10/13 10:54	O 文件	26 KB
ldo.o	2018/10/13 10:54	O 文件	25 KB
ldump.o	2018/10/13 10:54	O 文件	12 KB
lfunc.o	2018/10/13 10:54	O 文件	11 KB
lgc.o	2018/10/13 10:54	O 文件	30 KB
liblua53.a	2018/10/13 10:54	360压缩	638 KB
linit.o	2018/10/13 10:54	O 文件	4 KB

project > build-lua53-Desktop_Qt_5_11_1_MinGW_32bit-Release > release				
	名称	修改日期	类型	大小
	lapi.o	2018/10/13 10:57	O 文件	20 KB
	lauxlib.o	2018/10/13 10:57	O 文件	21 KB
	lbaselib.o	2018/10/13 10:57	O 文件	14 KB
	lbitlib.o	2018/10/13 10:57	O 文件	1 KB
	lcode.o	2018/10/13 10:57	O 文件	14 KB
	lcorolib.o	2018/10/13 10:57	O 文件	5 KB
	lctype.o	2018/10/13 10:57	O 文件	1 KB
	ldblolib.o	2018/10/13 10:57	O 文件	13 KB
	ldebug.o	2018/10/13 10:57	O 文件	10 KB
	ldo.o	2018/10/13 10:57	O 文件	9 KB
	ldump.o	2018/10/13 10:57	O 文件	4 KB
	lfunc.o	2018/10/13 10:57	O 文件	3 KB
	lgc.o	2018/10/13 10:57	O 文件	13 KB
	liblua53.a	2018/10/13 10:57	360压缩	316 KB

20.5.2.4. 组织库结构

同上

20.5.2.5. Qt 工程应用

注意我们的库名叫， liblua.a 但是在链接时候，保需要 -llua53 即可。

```

1 TEMPLATE = app
2 CONFIG += console c++11
3 CONFIG -= app_bundle
4 CONFIG -= qt
5
6 SOURCES += \
7     main.cpp
8
9 INCLUDEPATH += $$PWD/lua53/include
10
11 LIBS += -L$$PWD/lua53/lib/x86/Debug/ -llua53
12

```

```

#include <iostream>
#include "lua.hpp"

using namespace std;

int main()
{
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);
    const char *buf = "print('lua: Hello World')";
    luaL_dostring(L, buf);
    lua_close(L);
}

```

```
    return 0;  
}
```

21. C 提升 Lua 效率(lua->C/C++)

Lua 中可以调用 C 语言函数，并不意味着可以调用所有的 lua 函数。当 lua 调用 C 函数时，这个函数必须遵循某种规则来获取参数和返回结果。此外，lua 调用 C 函数时，必须注册该函数，即必须以一种恰当的方式为 lua 提供该 C 函数的地址。

lua 调用 C 函数时，也使用了一个与 C 语言调用 lua 函数时相同类型的栈，C 函数从栈中获取参数，并将结果压入栈中。

此处的栈，并不是一个全局的栈，每个函数都有其私有的局部栈(private local stack)。当 lua 调用 C 函数时，第一个参数，总是位于这个局部栈中索引为 1 的位置。

21.1. 引入

21.1.1. 从 lua 到 c

Qt 中单步调试 ipairs

21.1.2. ipairs 源码

```
/*
** Traversal function for 'ipairs'
*/
static int ipairsaux (lua_State *L) {
    lua_Integer i = luaL_checkinteger(L, 2) + 1;
    lua_pushinteger(L, i);
    return (lua_geti(L, 1, i) == LUA_TNIL) ? 1 : 2;
}

/*
** 'ipairs' function. Returns 'ipairsaux', given "table", 0.
** (The given "table" may not be a table.)
*/
static int luaB_ipairs (lua_State *L) {
#if defined(LUA_COMPAT_IPAIRS)
    return pairsmeta(L, "__ipairs", 1, ipairsaux);
#else
    luaL_checkany(L, 1);
    lua_pushcfunction(L, ipairsaux); /* iteration function */
    lua_pushvalue(L, 1); /* state */
    lua_pushinteger(L, 0); /* initial value */
    return 3;
}
```

```
#endif
```

21.1.3. C 即 Lua, Lua 即 C

21.2. 独立栈

无论何时 Lua 调用 C, 被调用的函数都得到一个新的栈, 这个栈独立于 C 函数本身的栈, 也独立于之前的 Lua 栈。它里面包含了 Lua 传递给 C 函数的所有参数, 而 C 函数则把要返回的结果放入这个栈以返回给调用者。

When Lua calls a C function, it uses the same kind of stack that C uses to call Lua. The C function gets its arguments from the stack and pushes the results on the stack.

An important point here is that the stack is not a global structure; each function has its own private local stack. When Lua calls a C function, the first argument will always be at index 1 of this local stack. Even when a C function calls Lua code that calls the same (or another) C function again, each of these invocations sees only its own private stack, with its first argument at index 1.

即通信的方式, 依然同前面讲 C 调用 lua 的栈, 但 lua 调用 C 用的栈, 是函数私有的本地栈 each function has its own private local stack, 每次调用均是全新的, 第一个入参的下标始科为 1。

21.3. Lua 访问 c 入栈值

21.3.1. API

21.3.1.1. setglobal

```
void lua_setglobal (lua_State *L, const char *name);
```

将虚拟栈中, 将栈顶元素弹出, 作为全局 lua 变量 name 的值。Pops a value from the stack and sets it as the new value of global name。

21.3.1.2. lua_newtable

```
void lua_newtable (lua_State *L);
```

产生一个空表, 并推入栈。Creates a new empty table and pushes it onto the stack。It is equivalent to lua_createtable(L, 0, 0)。

```
lua_newtable(L);
lua_setglobal(L, "nzhsoft");
```

21.3.1.3. settable

```
void lua_settable (lua_State *L, int index);
```

作一个等价于 $t[k] = v$ 的操作, 这里 t 是一个给定有效索引 index 处的值, v 指栈顶的值, 而 k 是栈顶之下那个值。

lua_settable 会把栈顶作为 value，栈顶的下一个作为 key 设置到 index 指向的 table，最后把这两个全部弹出栈，这时候 settable 完成。

Does the equivalent to $t[k] = v$, where t is the value at the given index, v is the value at the top of the stack, and k is the value just below the top。

This function **pops** both the key and the value from the stack。

```
//    lua_pushstring(L, "teacher");
//    lua_pushstring(L, "wgl");
//    lua_settable(L, -3);
```

21.3.1.4. **lua_setfield**

```
void lua_setfield (lua_State *L, int index, const char *k);
```

上式，等价于 $t[k] = v$, t 是栈上索引为 index 的表， v 是栈顶的值。函数结束，栈顶值 v 会被弹出。

Does the equivalent to $t[k] = v$, where t is the value at the given index and v is the value at the top of the stack。

This function pops the value from the stack。As in Lua, this function may trigger a metamethod for the "newindex" event

```
//    lua_pushstring(L, "teacher");
//    lua_pushstring(L, "wgl");
//    lua_settable(L, -3);

    lua_pushstring(L, "guilin");           //此两句，可顶上三句
    lua_setfield(L, -2, "teacher");
```

21.3.2. **lua** 访问 c 中入栈变量

21.3.2.1. **main.c**

```
#include "lua.h"
#include "lualib.h"
#include "lauxlib.h"

int main()
{
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    int price = 100;
    lua_pushnumber(L, price);
    int n = lua_gettop(L);

    printf("stack size = %d\n", n);
    lua_setglobal(L, "Price");
```

```

n = lua_gettop(L);
printf("stack size = %d\n", n);

char * teacher = "wgl";
lua_pushstring(L, teacher);
lua_setglobal(L, "Teacher");      --弹出栈顶元素，并给出一个名字，用于lua访问

 luaL_dofile(L, "xx.lua");
return 0;
}

```

21.3.2.2. xx.lua

```

print(Price)
print(Teacher)

```

21.3.3. lua 访问 c 中入栈的表

21.3.3.1. main.c

```

#include "lua.h"
#include "lualib.h"
#include "lauxlib.h"
int main()
{
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    lua_newtable(L);
    lua_pushstring(L, "price");
    lua_pushinteger(L, 100);
    lua_settable(L, -3);

    //    lua_pushstring(L, "teacher");
    //    lua_pushstring(L, "wgl");
    //    lua_settable(L, -3);

    lua_pushstring(L, "guilin");           //此两句，可顶上三句
    lua_setfield(L, -2, "teacher");

    lua_setglobal(L, "nzhsoft");

    luaL_dofile(L, "xx.lua");
    return 0;
}

```

21.3.3.2. xx.lua

```

print(nzhsoft.price)

```

```
print(nzhsoft.teacher)
```

21. 4. lua 调用 C 函数

21. 4. 1. API

21. 4. 1. 1. **lua_pushcfunction**

将函数压入栈，该函数接受一个 C 类型的函数指针，并将其以 lua function 形式入栈。在 lua 中调用该函数即会发生 c 函数的调用。

```
void lua_pushcfunction (lua_State *L, lua_CFunction f);
```

Pushes a C function onto the stack。This function receives a pointer to a C function and pushes onto the stack a Lua value of type function that, when called, invokes the corresponding C function。

Any function to be callable by Lua must follow the correct protocol to receive its parameters and return its results

21. 4. 1. 2. **lua_CFunction**

如下定义了一个函数指针，即为 lua 调用 C 函数的函数书写标准。为了更好的与 lua 通信，C 函数参数与返回值能过入栈的方式：即 C 函数获取 lua 参数通过栈，第一个参数是 index 为 1，最后一个参数的 index 在 lua_gettop(L)。返回值给 lua，只需将其依次入栈，并且返回返回值的个数。此时被 lua 调用的 C 函数也可以返回多个值。

```
typedef int (*lua_CFunction) (lua_State *L);
```

假设，在 lua 中调用一个函数，传入多个数值型数据，返回总和和平均值。lua 函数如：avg, sum = foo(1, 2, 3, 4, 5, 6)。

```
static int foo (lua_State *L)
{
    int n = lua_gettop(L);                                /* number of arguments */
    lua_Number sum = 0.0;
    int i;
    for (i = 1; i <= n; i++) {
        if (!lua_isnumber(L, i)) {
            lua_pushliteral(L, "incorrect argument");
            lua_error(L);
        }
        sum += lua_tonumber(L, i);
    }
    lua_pushnumber(L, sum/n);                            /* first result */
    lua_pushnumber(L, sum);                             /* second result */
    return 2;                                         /* number of results */
}
```

In order to communicate properly with Lua, a C function must use the following protocol, which defines the way parameters and results are passed:

a C function receives its arguments from Lua in its stack in direct order (the first argument is pushed first)。So, when the function starts, lua_gettop(L) returns the number of arguments

received by the function。The first argument (if any) is at index 1 and its last argument is at index lua_gettop(L)。

To return values to Lua, a C function just pushes them onto the stack, in direct order (the first result is pushed first), and returns the number of results.

Any other value in the stack below the results will be properly discarded by Lua。 Like a Lua function, a C function called by Lua can also return many results。

21.4.1.3. lua_register

lua_register 也是一个宏函数，可以简化书写。其本质等同于先 pushcfunction，然后再 setglobal。

```
void lua_register (lua_State *L, const char *name, lua_CFunction f);
```

Sets the C function f as the new value of global name。It is defined as a macro。

```
#define lua_register(L, n, f) (lua_pushcfunction(L, f), lua_setglobal(L, n))
```

21.4.2. 调用 C++ 函数-无参无返回

21.4.2.1. xx.lua

```
func()
```

21.4.2.2. main.c

```
#include "lua.h"
#include "lualib.h"
#include "lauxlib.h"

int func(lua_State *L)
{
    printf("c function\n");
    return 0;
}

int main()
{
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    lua_pushcfunction(L, &func);
    lua_setglobal(L, "func");

    luaL_dofile(L, "xx.lua");
    lua_close(L);
    return 0;
}
```

21.4.3. 调用 C++ 函数-有参无返回

此时函数参数的压栈顺序是从左往右压入栈中。

21.4.3.1. xx.lua

```
func(6999, "nzhsoft")
```

21.4.3.2. main.c

```
#include "lua.h"
#include "lualib.h"
#include "lauxlib.h"

int func(lua_State *L)
{
    printf("c function\n");
    char * str = lua_tostring(L, -1);
    int price = lua_tointeger(L, -2);
    printf("price = %d str = %s", price , str);

    return 0;
}

int main()
{
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    luaL_pushcfunction(L, &func);
    luaL_setglobal(L, "func");

    luaL_dofile(L, "xx.lua");
    luaL_close(L);
    return 0;
}
```

21.4.4. 调用 C++ 函数-有参有返回

返回值的压栈顺序，即为，lua 函数返回值的接受顺序。

21.4.4.1. xx.lua

```
price, str = func(6999, "nzhsoft")
print(price)
print(str)
```

21.4.4.2. main.c

```
#include "lua.h"
#include "lualib.h"
#include "lauxlib.h"
int func(lua_State *L)
```

```

{
    printf("c function\n");
    char * str = lua_tostring(L, -1);
    int price = lua_tointeger(L, -2);
    printf("price = %d str = %s\n", price , str);
    printf("stack size = %d\n", lua_gettop(L));

    lua_pushinteger(L, price*2);
    char buf[1024];
    sprintf(buf, "good C&C++ study %s", str);
    lua_pushstring(L, buf);
    printf("stack size = %d\n", lua_gettop(L));
    return 2;
}

int main()
{
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    //    lua_pushcfunction(L, &func);
    //    lua_setglobal(L, "func");
    luaL_register(L, "func", func);

    luaL_dofile(L, "xx.lua");
    lua_close(L);
    return 0;
}

```

21.4.5. 调用 C++ 函数返回 table

table 中可以有普通字段，也可以有函数字段。

21.4.5.1. xx.lua

```

print "in xx.lua"

t = func_return_table()
print(t.key)

t.foo()

```

21.4.5.2. main.c

```

int foo(lua_State *L)
{
    printf("tab.foo\n");
}

```

```

int func_return_table(lua_State *L)
{
    lua_newtable(L);           // 创建一个表格，放在栈顶
    lua_pushstring(L, "key");  // 压入 key
    lua_pushnumber(L, 66);     // 压入 value
    lua_settable(L, -3);      // 弹出 key, value，并设置到 table 里面去

    lua_pushstring(L, "foo");
    lua_pushcfunction(L, foo);
    lua_settable(L, -3);      // 不需要给 tab 名字
    return 1;
}

int main()
{
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    luaL_pushcfunction(L, func_return_table);
    luaL_setglobal(L, "func_return_table");

    luaL_dofile(L, "xx.lua");
    luaL_close(L);
}

```

21.5. C++函数批量注册/表化

21.5.1. API

21.5.1.1. luaL_Reg

`Luau_Reg` 定义如下：

```

typedef struct luaL_Reg
{
    const char *name;
    lua_CFunction func;
} luaL_Reg;

```

用于存放，函数指针和注册的函数名。常用于生成数组，数组的最后一个元素，一定是以`{NULL, NULL}`结尾。

Type for arrays of functions to be registered by `luaL_setfuncs`。 name is the function name and func is a pointer to the function。 Any array of `luaL_Reg` must end with a sentinel entry in which both name and func are NULL。

21.5.1.2. luaL_newlib

`luaL_newlib` 也是一个宏函数，简化了创建表和循环注册函数。

创建表入栈，并且注册 1 中的所有函数到栈上的表。等价于：`(luaL_newlibtable(L, 1), luaL_setfuncs(L, 1, 0))`

```
void luaL_newlib (lua_State *L, const luaL_Reg l[]);
```

创建一张新的表，并预分配足够保存下数组 l 内容的空间（但不填充）。这是给 luaL_setfuncs 一起用的。

它以宏形式实现，数组 l 必须是一个数组，而不能是一个指针。

```
void luaL_newlibtable (lua_State *L, const luaL_Reg l[]);
```

21.5.1.3. luaL_setfuncs

设置数组 l 中所有函数的到栈顶的表中，第三个参数通常为 0。Registers all functions in the array l (see luaL_Reg) into the table on the top of the stack (below optional upvalues, see next)。

这个函数，是一种循环注册的简化函数。

```
void luaL_setfuncs (lua_State *L, const luaL_Reg *l, int nup);
```

21.5.1.4. luaL_requiref

如果 modname 不在 package.loaded 中，则调用 openf 函数以 modname 为参数，使其为 package.loaded[modname]。其行为类似于 require，如果 glb 为 true，则存储模块到全局的 modname。

```
void luaL_requiref (lua_State *L,
                    const char *modname,
                    lua_CFunction openf,
                    int glb);
```

If modname is not already present in package.loaded , calls function openf with string modname as an argument and sets the call result in package.loaded[modname] , as if that function has been called through require 。

If glb is true, also stores the module into global modname。Leaves a copy of the module on the stack.

21.5.2. 循环批量绑定全局函数

```
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>

//要想注册进 lua,
//函数的定义为 typedef int (*lua_CFunction)(lua_State* L)
int printHello(lua_State * l)
{
    lua_pushstring(l, "hello lua");
    //返回值代表向栈内压入的元素个数
    return 1;
}

int foo(lua_State * l)
{
    //获得 Lua 传递过来的参数个数
```

```
int n = lua_gettop(l);
if(n != 0)
{
    //获得第一个参数
    int i = lua_tonumber(l, 1);
    //将传递过来的参数加一以后最为返回值传递回去
    lua_pushnumber(l, i+1);
    return 1;
}
return 0;
}

//相加
int add(lua_State * l)
{
    int n = lua_gettop(l);
    int sum = 0;
    for (int i=0;i<n;i++)
    {
        sum += lua_tonumber(l, i+1);
    }
    if(n!=0)
    {
        lua_pushnumber(l, sum);
        return 1;
    }
    return 0;
}
//把需要用到的函数都放到注册表中，统一进行注册
const luaL_Reg lib[]=
{
    {"printHello", printHello},
    {"foo", foo},
    {"add", add},
    {NULL, NULL}
};

int main(int argc, const char * argv[])
{
    //创建一个新的 Lua 环境
    lua_State * l= luaL_newstate();
    //打开需要的库
    luaL_openlibs(l);
    //统一注册 lua 中调用的函数
    const luaL_Reg* libf =lib;
    for (; libf->func; libf++)
```

```
{
    //把 foo 函数注册进 lua,
    //第二个参数代表 Lua 中要调用的函数名称,
    //第三个参数就是 c 层的函数名称
    luaL_register(l, libf->name, libf->func);
    //将栈顶清空
    lua_settop(l, 0);
}
//加载并且执行 lua 文件
luaL_dofile(l, "xx.lua");
//关闭
lua_close(l);
return 0;
}
```

21.5.3. 批量表化绑定全局函数

21.5.3.1. main.c

```
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>

//要想注册进 lua,
//函数的定义为 typedef int (*lua_CFunction)(lua_State* L)
int printHello(lua_State * L)
{
    lua_pushstring(L, "hello lua");
    //返回值代表向栈内压入的元素个数
    return 1;
}
int foo(lua_State * L)
{
    //获得 Lua 传递过来的参数个数
    int n = lua_gettop(L);
    if(n != 0)
    {
        //获得第一个参数
        int i = lua_tonumber(L, 1);
        //将传递过来的参数加一以后最为返回值传递回去
        lua_pushnumber(L, i+1);
        return 1;
    }
    return 0;
}
//相加
int add(lua_State * L)
{
    int n = lua_gettop(L);
```

```
int sum = 0;
for (int i=0;i<n;i++)
{
    sum += lua_tonumber(L, i+1);
}
if(n!=0)
{
    lua_pushnumber(L, sum);
    return 1;
}

return 0;
}
//把需要用到的函数都放到注册表中，统一进行注册
const luaL_Reg myLib[]=
{
    {"printHello", printHello},
    {"foo", foo},
    {"add", add},
    {NULL, NULL}
};
int luaopen_my(lua_State * L)
{
    //首先创建一个 table，然后把成员函数名做 key，成员函数作为 value 放入该 table 中
    luaL_newlib(L, myLib);
    return 1;
}
int main(int argc, const char * argv[])
{
    //创建一个新的 Lua 环境
    lua_State * L= luaL_newstate();
    //打开需要的库
    luaL_openlibs(L);
    //统一注册 lua 中调用的函数
    luaL_requiref(L, "my", luaopen_my, 1);
    //加载并且执行 lua 文件
    luaL_dofile(L, "xx.lua");
    //关闭
    lua_close(L);

    return 0;
}
```

21.5.3.2. lua.c

21. 6. C++ ->dll / so

21. 6. 1. 生成 so 流程

21. 6. 1. 1. xx.c

```
#include <stdio.h>
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>

//要想注册进 lua,
//函数的定义为 typedef int (*lua_CFunction)(lua_State* L)
int printHello(lua_State * L)
{
    lua_pushstring(L, "hello lua");
    //返回值代表向栈内压入的元素个数
    return 1;
}
int foo(lua_State * L)
{
    //获得 Lua 传递过来的参数个数
    int n = lua_gettop(L);
    if(n != 0)
    {
        //获得第一个参数
        int i = lua_tonumber(L, 1);
        //将传递过来的参数加一以后最为返回值传递回去
        lua_pushnumber(L, i+1);
        return 1;
    }
    return 0;
}
//相加
int add(lua_State * L)
{
    int n = lua_gettop(L);
    int sum = 0;
    for (int i=0;i<n;i++)
    {
        sum += lua_tonumber(L, i+1);
    }
    if(n!=0)
    {
        lua_pushnumber(L, sum);
        return 1;
    }
}

return 0;
```

```

}

//把需要用到的函数都放到注册表中，统一进行注册
const luaL_Reg xxLib[] =
{
    {"printHello", printHello},
    {"foo", foo},
    {"add", add},
    {NULL, NULL}
};

int luaopen_xx(lua_State * L)
{
    //首先创建一个 table，然后把成员函数名做 key，成员函数作为 value 放入该 table 中
    luaL_newlib(L, xxLib);
    return 1;
}

```

21.6.1.2. xx.lib

将 xx.c 生成 xx.lib 或 xx.so，并将其放置于 c 库的路径下，

```
#> gcc xx.c -fPIC -shared -Wall -o xx.so
```

21.6.1.3. require "xx"

然后在 lua 中采用 require 加载。

```
local xx= require "xx"
print(xx.add(1,2,3,4,5,6))
```

21.6.2. Skynet 典型模块

21.6.2.1. lua-mongo.c

```

LUAMOD_API int
luaopen_skynet_mongo_driver(lua_State *L) {
    luaL_checkversion(L);
    luaL_Reg l[] = {
        { "query", op_query },
        { "reply", op_reply },
        { "kill", op_kill },
        { "delete", op_delete },
        { "more", op_get_more },
        { "update", op_update },
        { "insert", op_insert },
        { "length", reply_length },
        { NULL, NULL },
    };
}
```

```
    luaL_newlib(L,1);
    return 1;
}
```

21.6.2.2. lua-crypt.c

```
LUAMOD_API int
luaopen_skynet_crypt(lua_State *L) {
    luaL_checkversion(L);
    static int init = 0;
    if (!init) {
        // Don't need call srand more than once.
        init = 1 ;
        srand((random() << 8) ^ (time(NULL) << 16) ^ getpid());
    }
    luaL_Reg l[] = {
        { "hashkey", lhashkey },
        { "randomkey", lrandomkey },
        { "desencode", ldesencode },
        { "desdecode", ldesdecode },
        { "hexencode", ltohex },
        { "hexdecode", lfromhex },
        { "hmac64", lhmac64 },
        { "hmac64_md5", lhmac64_md5 },
        { "dhexchange", ldhexchange },
        { "dhsecret", ldhsecret },
        { "base64encode", lb64encode },
        { "base64decode", lb64decode },
        { "sha1", lsha1 },
        { "hmac_sha1", lhmac_sha1 },
        { "hmac_hash", lhmac_hash },
        { "xor_str", lxor_str },
        { NULL, NULL },
    };
    luaL_newlib(L,1);
    return 1;
}
```

21.7. C 函数的其它操作

21.7.1. Array Manipulation

21.7.2. String Manipulation

21.7.3. Upvalues

22. Full Userdata

22. 1. Userdata

22. 1. 1. 定义

前面的章节，讲解了，如何将 c 写的函数用 lua 来调用，本章旨在如何将 c 中自定义的类型用 lua 来使用。C 中自定义的类型相对于 lua 而言就是 UserData。

lua 用 userdata 来表述 c 语言中的自定义结构。userdata 提供一段原生的内存区域，表在 lua 中没有任何预定义的操作，可以用于存储任何类型。

Full UserData 的内存管理，由 lua 虚拟机来统一管理。Light UserData 的内存管理则需要自己来管理。

22. 1. 2. 接口

22. 1. 2. 1. lua_newuserdata

函数 `lua_newuserdata` 函数分配一块指定大小的内存块，把内存块地址作为一个完全用户数据压栈，并返回这个地址。宿主程序可以随意使用这块内存。

```
void *lua_newuserdata (lua_State *L, size_t size);
```

22. 1. 2. 2. lua_touserdata

函数 `lua_touserdata`，如果给定索引处的值是一个 full userdata，函数返回其内存块的地址。如果值是一个 light userdata，那么就返回它表示的指针。否则，返回 NULL。

```
void *lua_touserdata (lua_State *L, int index);
```

22. 1. 2. 3. luaL_argcheck

检查 "cond" 是否为 "true"，如果为 "false" 则报错，并返回形如如下格式的错误，* "bad argument #arg to 'funcname' (extramsg)"

```
void luaL_argcheck (lua_State *L, int cond, int arg, const char *extramsg);
```

22. 2. 自实现 array

用 c 语言来实现一个数组的操作，用 lua 语言来调用。

22. 2. 1. code Lua

```
print "in xx.lua"

arr = array.new(100)
print(arr,type(arr))
print(array.size(arr))

for i = 1, 50 do
    array.set(arr, i, i+100)
end
print(array.get(arr, 10))
```

22.2.2. code C

数组声明为一个长度只是为了占位，因为 C 中不允许声明长度为 0 的数组，在实际程序中，我们会根据指定的大小来申请合适的空间。

```
typedef struct ARRAY
{
    int size;
    double values[1];
} ARRAY;
```

// "n" 为指定的大小。因为 "NumArray" 中已包含了一个元素的大小，所以需要减去。
sizeof(NumArray) + (n - 1) * sizeof(double)

```
//array.c
#include <stdio.h>
#include <string.h>
#include <lua.hpp>

typedef struct ARRAY
{
    int size;
    double values[1];
} ARRAY;

static int newArray(lua_State *L)
{
    // 检查待创建数组大小参数是否为整数。
    int n = luaL_checkinteger(L, 1);
    size_t nbytes = sizeof(ARRAY) + (n - 1)*sizeof(double);
    ARRAY *a = (ARRAY *)lua_newuserdata(L, nbytes);
    a->size = n;      // 设置数组的大小。
    return 1;          // 函数返回创建的"userdata"。
}

static int setArray(lua_State *L)
{
    // 获取传递的数组 ("userdata") 的地址。
    ARRAY *a = (ARRAY *)lua_touserdata(L, 1);
    // 检查传递的"key" 是否为整数。
    int index = luaL_checkinteger(L, 2);
    // 检查传递的"value" 是否为数值。
    double value = luaL_checknumber(L, 3);

    luaL_argcheck(L, a != NULL, 1, "'array' expected");
    luaL_argcheck(L, 1 <= index && index <= a->size, 2, "index out of range");

    a->values[index - 1] = value;      // 设置数组的值。
```

```
    return 0;
}

static int getArray(lua_State *L)
{
    ARRAY *a = (ARRAY *)lua_touserdata(L, 1);
    int index = luaL_checkinteger(L, 2);

    luaL_argcheck(L, a != NULL, 1, "'array' expected");
    luaL_argcheck(L, 1 <= index && index <= a->size, 2,
                  "index out of range");

    // 获取数组中指定的值并入栈。
    lua_pushnumber(L, a->values[index - 1]);

    return 1;      // 函数返回获取的值。
}

static int getSize(lua_State *L)
{
    ARRAY *a = (ARRAY *)lua_touserdata(L, 1);
    luaL_argcheck(L, a != NULL, 1, "'array' expected");
    lua_pushnumber(L, a->size);      // 获取数组的大小。

    return 1;      // 函数返回数组的大小。
}

static const struct luaL_Reg arraylib[] = {
    {"new", newArray},
    {"set", setArray},
    {"get", getArray},
    {"size", getSize},
    {NULL, NULL}
};

extern int luaopen_array(lua_State* L)
{
    luaL_newlib(L, arraylib);
    return 1;
}

int main()
{
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);
```

```

luaL_requiref(L, "array", luaopen_array, 1);

luaL_dofile(L, "xx.lua");
lua_close(L);

return 0;
}

```

22. 2. 3. Code C->dll/so

22. 2. 3. 1. so

```
gcc -fPIC -shared -Wall array.c -o array.so
```

22. 2. 3. 2. lua

```

local arr = require "array"

print(arr,type(arr))
local a = arr.new(100)
print (arr.size(a))

for i=1,100 do
    arr.set(a,i,i+100)
end

print(arr.get(a,10))

```

22. 3. 差异化 userdata

22. 3. 1. 无差异

上例中 a 是 array.new 的返回值是一个 userdata，且是其它函数的传入参数，不同的 userdata 就缺少了界限，给一个 userdata 增加 metatable 是为了区分不同 userdata 之用。

io.stdin 也是一个 userdata，假设上例中，用其作传入参数，但是会产生运行时错误。故对上例，继承改进。

```

local arr = array.new(100)
print(arr)

array.set(arr,1,10)
print(array.get(arr,1))

array.set(io.stdin, 1,10)

```

每次创建用户有数据时，用相应的元表进行标记，每当用户获取数据时，检查其是否有正确的元表，因为 Lua 代码不能更改"userdata"的"metatable"，因此，不能绕过检查。

22. 3. 2. metatable

22. 3. 2. 1. Api 解析

函数 luaL_newmetatable，会创建一张新表(即元表)，并将该表与注册表中的指定名称 tname 关联起来。

```
int luaL_newmetatable(lua_State *L, const char *tname);
```

函数 `luaL_getmetatable`，会从注册表中获取与 `tname` 关联的元表，压栈，如果没有 `tname` 对应的元表，则将 `nil` 压栈并返回假。

```
int luaL_getmetatable(lua_State *L, const char *tname);
```

`lua_setmetatable` 把栈顶一张表弹出栈，并将其设为给定索引处的值的元表。

```
void lua_setmetatable(lua_State *L, int index);
```

`luaL_checkudata`，会检查栈中指定位置上的对象是否是与指定名称的元表匹配的表用户数据。如果，该对象不是用户数据，或者该用户数据没有正确的元表，`checkudata` 就会引发错误。否则，返回这个用户数据的地址。

```
void *luaL_checkudata (lua_State *L, int arg, const char *tname);
```

22.3.2.2. 加入 metatable

```
#include <stdio.h>
#include <string.h>
#include <lua.hpp>

typedef struct ARRAY
{
    int size;
    double values[1];
} ARRAY;

#define checkArray(L) (ARRAY*)luaL_checkudata(L, 1, "nzhsoft.array")

static int newArray(lua_State *L)
{
    // 检查待创建数组大小参数是否为整数。
    int n = luaL_checkinteger(L, 1);
    size_t nbytes = sizeof(ARRAY) + (n - 1)*sizeof(double);
    ARRAY *a = (ARRAY *)lua_newuserdata(L, nbytes);
    a->size = n;      // 设置数组的大小。

    luaL_getmetatable(L, "nzhsoft.array");
    lua_setmetatable(L, -2);

    return 1;      // 函数返回创建的"userdata"。
}

static int setArray(lua_State *L)
{
    // 获取传递的数组 ("userdata") 的地址。
```

```
ARRAY *a = checkArray(L);
// 检查传递的"key"是否为整数。
int index = luaL_checkinteger(L, 2);
// 检查传递的"value"是否为数值。
double value = luaL_checknumber(L, 3);

a->values[index - 1] = value; // 设置数组的值。

return 0;
}

static int getArray(lua_State *L)
{
    ARRAY *a = checkArray(L);

    int index = luaL_checkinteger(L, 2);
    // 获取数组中指定的值并入栈。
    lua_pushnumber(L, a->values[index - 1]);

    return 1; // 函数返回获取的值。
}

static int getSize(lua_State *L)
{
    ARRAY *a = checkArray(L);
    lua_pushnumber(L, a->size); // 获取数组的大小。
    return 1; // 函数返回数组的大小。
}

static const struct luaL_Reg arraylib[] = {
    {"new", newArray},
    {"set", setArray},
    {"get", getArray},
    {"size", getSize},
    {NULL, NULL}
};

extern int luaopen_array(lua_State* L)
{
    luaL_newmetatable(L, "nzhsoft.array");
    luaL_newlib(L, arraylib);
    return 1;
}

int main()
{
    lua_State *L = luaL_newstate();
```

```

 luaL_openlibs(L);

luaL_requiref(L, "array", luaopen_array, 1);

//    luaL_dofile(L, "xx.lua");
//    luaL_loadfile(L, "xx.lua") ; lua_call(L, 0, LUA_MULTRET);
//    lua_close(L);

return 0;
}

```

22. 4. Object Oriented

22. 4. 1. main.c

```

static const struct luaL_Reg arraylib_m[] = {
    {"set", setArray},
    {"get", getArray},
    {"size", getSize},
    {NULL, NULL}
};

extern int luaopen_array(lua_State* L)
{
    luaL_newmetatable(L, "nzhsoft.array");

    lua_pushstring(L, "__index");
    lua_pushvalue(L, -2); // 复制一份"metatable"再次入栈。
    lua_settable(L, -3); // "metatable.__index = metatable"
    luaL_setfuncs(L, arraylib_m, 0);

    luaL_newlib(L, arraylib);
    return 1;
}

```

22. 4. 2. xx.lua

```

print "start xx.lua"

local arr = array.new(100)

print(arr:size())

for i=0,10 do
    arr:set(i,i*100)
end

```

```
print(arr:get(5))  
print "end xx.lua"
```

22.5. Class userdata

<http://www.cnblogs.com/chevin/p/5896858.html>

23. Light UserData

23. 1. full vs light

24. Lua 的状态与线程

我们不信任基于抢占式内存共享的多线程技术。在 HOPL 论文中，我们写道：“我们仍然认为，如果在连 `a=a+1` 都没有确定结果的语言中，无人可以写出正确的程序。

调用 `lua_newthread` 便可以在一个 lua 状态中创建其他的线程：

```
lua_State *lua_newthread(lua_State *L);
```

这个函数会返还一个 `lua_State` 指针，表示新建的线程。它还会将新线程作为一个类型为“`thread`”的值压入栈中。例如，在执行了以下语句后：

```
L1 = lua_newthread(L);
```

拥有了两个线程 `L1` 和 `L`，它们内部都引用了相同的 Lua 状态。每个线程都有其自己的栈。新线程 `L1` 以一个空栈开始运行，老线程 `L` 的栈顶就是这个新线程：

```
printf("%d\n", lua_gettop(L1));-->0  
print("%d\n", luaL_typename(L, -1));-->thread
```

除了主线程之外，其他线程和其他 lua 对象一样都是垃圾回收的对象。当新建一个线程时，线程会压入栈中，这样就能确保新线程不会成为垃圾。

25. LuaBridge

26. 消消乐 Skynet 服务

27. 综合练习

27.1. 写出运行结果

27.1.1. 表长度

```
test = {1, 2, 3, 4, 5, 6, 7, 8}
print(#test)
```

27.1.2. 表引用

```
a = {}
a["x"] = 10
b = a
print(b["x"])
b["x"] = 20
print(a["x"])
```

27.1.3. 面向对象

```
function base_type:new(x)
    local d = {}
    setmetatable(d, self)
    self.x = x
    return d
end

function base_type:ctor(x)
    print("base_type ctor")
    self.x = x
end

function base_type:print_x()
    print(self.x)
end

function base_type:hello()
    print("hello base_type")
end

a = base_type:new(10)
a:print_x()
a:hello()
```

27.1.4. 继承

```
Class = {x = 0, y = 0}
Class.__index = Class

function Class:new(x, y)
    local self = {}
    setmetatable(self, Class)
    self.x = x
    self.y = y
    return self
end

function Class:test()
    print(self.x, self.y)
end

function Class:plus()
    self.x = self.x + 1
    self.y = self.y + 1
end

Main = {z = 0}
setmetatable(Main, Class)
Main.__index = Main
function Main:new(x, y, z)
    local self = {}
    self = Class:new(x, y)
    setmetatable(self, Main)
    self.z = z
    return self
end

function Main:go()
    self.x = self.x + 10
end

function Main:test()
    print(self.x, self.y, self.z)
end

c = Main:new(20, 40, 100)
c:test()

d = Main:new(10, 50, 200)
```

```
d:go()  
d:plus()  
d:test()  
c:test()
```

27.2. 写程序

27.2.1. 对表进行排序并输出

```
network = {  
    {name = "grame",    IP = "202.26.12.32"},  
    {name = "aricial",  IP = "202.26.30.23"},  
    {name = "lua",       IP = "202.26.30.21"},  
}
```

27.2.2. 请写一个带有不定参数的 lua 函数 并输出所有的参数