2Lua快速入门

一、Lua概述

1.1 Lua是什么

> Lua 是一个小巧精妙的脚本语言，诞生于巴西的大学实验室，这个名字在葡萄牙语里的含义是“美丽的月亮”。Lua开发小组的目标是开发一种小巧、高效且能够很好地和C语言一起工作的编程语言。在脚本语言领域，Lua是最快、最高效的脚本语言之一，因为它有资格作为游戏开发的备选方案。

1.2 脚本语言

1.3 编程语言

- 机器语言：0和1,人类几乎没办法阅读、理解
- 汇编语言：由一些特定指令构成，学习成本比较高；在底层做一些驱动，体积小，运行效率高
- 高级语言：c、c++、java、php、python、javascript等

下面呢,我们着重看一下什么是编译类语言,什么是解释类语言:

- 编译类语言: 先通过高级语言把我们的程序写出来,然后再通过编译器编译成我们的目标机器语言,就是计算机所能认识的0或1
- 解释类语言: 由解释器完成。

接下来,我们比较一下这两种语言的优缺点:

- 编译类语言: 运行效率高
- 解释类语言: 更灵活; 缺点呢就是运行效率低。

而脚本语言呢?

摘自百度百科上一句话:

> 一个脚本通常是解释执行而非编译。脚本语言通常都有简单、易学、易用的特性，目的就是希望能让程序员快速完成程序的编写工作。

所以我理解的是: 脚本语言是一种解释型语言,例如Python、lua、javascript等等,它不象c\c++等可以编译成二进制代码,以可执行文件的形式存在，脚本语言不需要编译，可以直接用，由解释器来负责解释。

二、Lua特性

> Lua是一门嵌入式的脚本语言，如果你Lua当成开发独立应用程序时使用的语言，那可能要让你失望了。

- 可移植
- 良好的嵌入式
- 尺寸非常小
- 效率很高

能穿过针孔的语言,"小而精" 就是我最大的特点!

三、Lua 应用场景

> 如今我已是游戏领域使用最广泛的脚本语言之一

其他领域

- web应用：Open2Resty使用我Lua扩展Nginx服务器的功能，使用者仅需要编写Lua代码就能轻松完成业务逻辑。值得一提的是，这个项目的作者是中国人章亦春。

- Redis原子性： Redis服务提供Lua脚本。
- Adobe Photoshop Lightroom 也是使用Lua编写的插件。

游戏领域

- C\C++语言实现的服务器引擎内核，其中包括最核心的功能，比如网络收发、数据库查询、游戏主逻辑循环等。以下将这一层简称为引擎层。
- 向引擎层注册一个Lua主逻辑脚本，当接收到用户数据时，将数据包放到Lua脚本中进行处理，主逻辑脚本主要是一个大的函数表，可以根据接收到的协议包的类型，调用相关的函数进行处理。以下将这一层简称为脚本层。
- 引擎层向脚本层提供很多API，能方便地调用引擎层的操作，比如脚本层处理完逻辑之后调用引擎层的接口应答数据等。

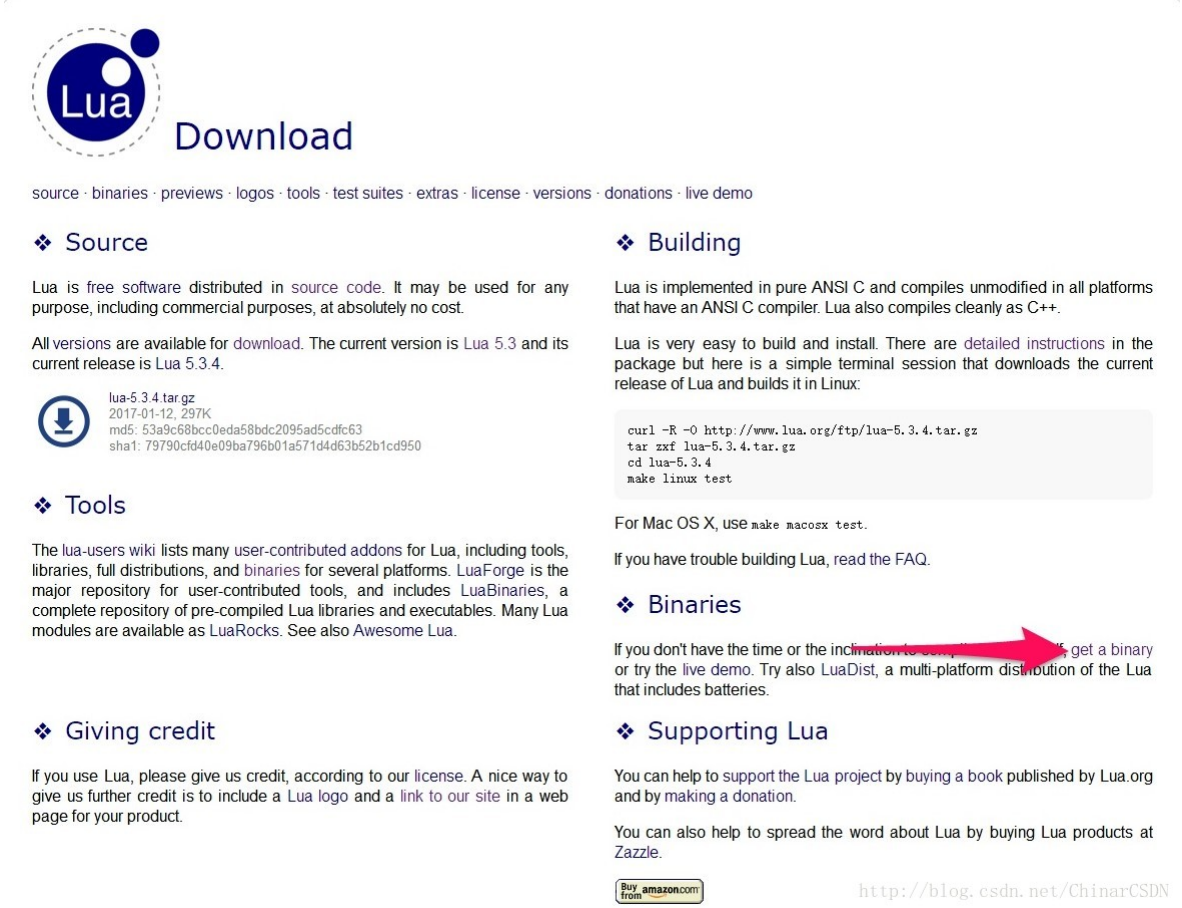
四、Lua环境安装

4.1 在 Windows 上搭建环境

4.1.1 进入Lua官网：http://www.lua.org——下载Lua

4.1.2 点击——下载/download

4.1.3 点击——获取一个二进制文件/get a binary



4.1.4 进入新界面，点击左侧的Download

4.1.5 选择自己需要的文件进行下载

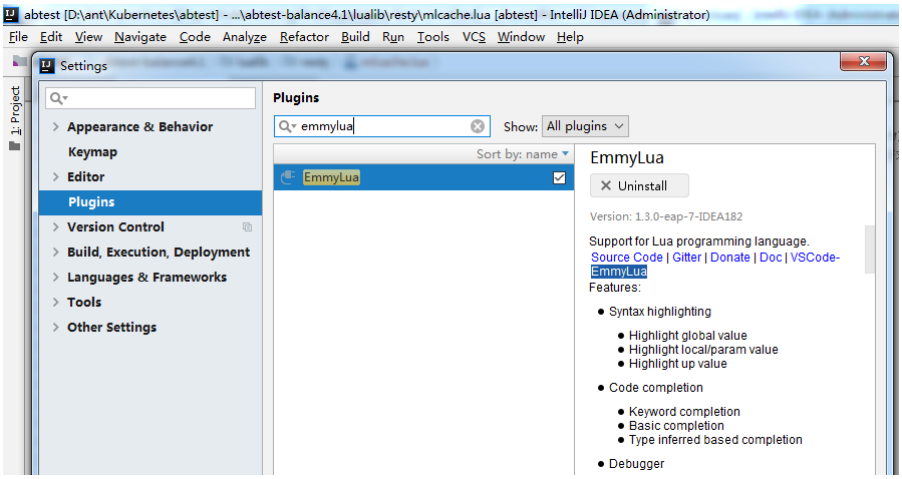
五、Lua编译器选择

5.1 下载idea并配置

idea是一个java语言非常受好评的编辑器,但是并不是只支持java.

5.2 安装emmylua插件

安装完成后打开File->Settings->Plugins在其中输入emmylua点击右边的install安装并重启idea



5.3 创建Lua项目

六、Lua 基本语法

6.1 注释

单行注释

两个减号是单行注释:

```
--
```

多行注释

```
--[[
 多行注释
 多行注释
 --]]
```

6.2 标示符

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

最好不要使用下划线加大写字母的标示符，因为Lua的保留字也是这样的。

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

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

6.3 关键词

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

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

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

6.4 全局变量

在默认情况下，变量总是认为是全局的。

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

```
> print(b)
nil
> b=10
> print(b)
10
>
```

如果你想删除一个全局变量，只需要将变量赋值为nil。

```
b = nil
print(b)      --> nil
```

七、Lua 数据类型

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

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

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

函数 type 能够返回一个值或一个变量所属的类型。

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

7.1 nil（空）

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

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

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

7.2 boolean（布尔）

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

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

if b then
    print("b")        -->output:b
```

```
else
    print("not b")    --这个没有执行
end

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

7.3 number（数字）

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

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

7.4 string（字符串）

Lua 中有三种方式表示字符串:

1、使用一对匹配的单引号。例：'hello'。

2、使用一对匹配的双引号。例："abclua"。

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

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

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

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

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

7.5 table (表)

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

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

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

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

7.6 function (函数)

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

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

local a = foo     --把函数赋给变量

print(a())

--output:
in the function
30
```

7.7 thread（线程）

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

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

7.8 userdata（自定义类型）

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

八、Lua 表达式

8.1 算术运算符

Lua 的算术运算符如下表所示：

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

```
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
```

8.2 关系运算符

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

> 示例代码：test2.lua

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

注意：Lua 语言中“不等于”运算符的写法为：~=

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

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

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

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

8.3 逻辑运算符

逻辑运算符	说明
and	逻辑与
or	逻辑或
not	逻辑非

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

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

> 示例代码：test3.lua

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

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

8.4 字符串连接

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

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

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

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

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

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

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

8.5 优先级

Lua 操作符的优先级如下表所示(从高到低)：

优先级
^
not　#　-
*　/　%
+　-
..
<　>　<=　>=　==　~=
and
or

> 示例：

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

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

九、Lua变量

> 变量在使用前，需要在代码中进行声明，即创建该变量。

编译程序执行代码之前编译器需要知道如何给语句变量开辟存储区，用于存储变量的值。

Lua 变量有三种类型：全局变量、局部变量、表中的域。

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

局部变量的作用域为从声明位置开始到所在语句块结束。

变量的默认值均为 nil。

```
-- test.lua 文件脚本
a = 5               -- 全局变量
local b = 5         -- 局部变量

function joke()
    c = 5           -- 全局变量
    local d = 6     -- 局部变量
end

joke()
print(c,d)          --> 5 nil

do
    local a = 6     -- 局部变量
    b = 6           -- 对局部变量重新赋值
    print(a,b);     --> 6 6
end

print(a,b)      --> 5 6
```

执行以上实例输出结果为：

```
$ lua test.lua
5    nil
6    6
5    6
```

十、Lua 循环

> 很多情况下我们需要做一些有规律性的重复操作，因此在程序中就需要重复执行某些语句。

> 一组被重复执行的语句称之为循环体，能否继续重复，决定循环的终止条件。

> 循环结构是在一定条件下反复执行某段程序的流程结构，被反复执行的程序被称为循环体。

> 循环语句是由循环体及循环的终止条件两部分组成的。

10.1 Lua 语言提供了以下几种循环处理方式：

循环类型	描述
while	在条件为 true 时，让程序重复地执行某些语句。执行语句前会先检查条件是否为 true。
for	重复执行指定语句，重复次数可在 for 语句中控制。
repeat...until	重复执行循环，直到 指定的条件为真时为止

10.2 循环控制语句

循环控制语句用于控制程序的流程， 以实现程序的各种结构方式。

Lua 支持以下循环控制语句：

控制语句	描述
break 语句	退出当前循环或语句，并开始脚本执行紧接着的语句。
goto 语句	将程序的控制点转移到一个标签处。

10.3 无限循环

在循环体中如果条件永远为 true 循环语句就会永远执行下去，以下以 while 循环为例：

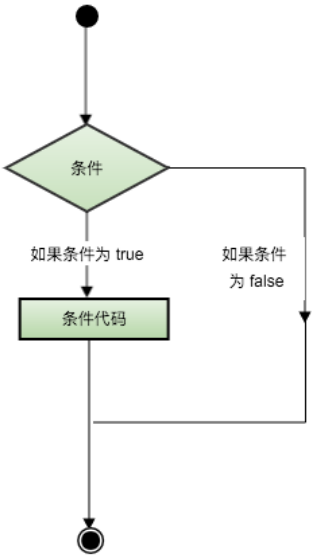
实例

```
while( true )
do
   print("循环将永远执行下去")
end
```

十一、Lua 流程控制

Lua 编程语言流程控制语句通过程序设定一个或多个条件语句来设定。在条件为 true 时执行指定程序代码，在条件为 false 时执行其他指定代码。

以下是典型的流程控制流程图：



语句	描述
if 语句	**if 语句** 由一个布尔表达式作为条件判断，其后紧跟其他语句组成。
if...else 语句	**if 语句** 可以与 **else 语句**搭配使用, 在 if 条件表达式为 false 时执行 else 语句代码。

十二、Lua 函数

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

示例代码：

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

使用函数的好处：

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

12.1 函数定义

Lua 使用关键字 *function* 定义函数，语法如下：

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

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

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

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

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

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

> 示例代码：

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

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

如果参数列表为空，必须使用 `()` 表明是函数调用。

> 示例代码：

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

func()                  --函数调用，圆扩号不能省

--> output:
no parameter
```

在定义函数要注意几点：

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

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

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

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

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

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

12.2 函数的参数

按值传递

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

> 示例代码：

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

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

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

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

> 示例代码：

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

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

local x = 1
```

```lua
local y = 2
local z = 3

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

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

变长参数

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

> 示例代码：

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

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

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

-->output
1 2

1 2 3 4
```

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

具名参数

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

> 示例代码：

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

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

-->output
```

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

按引用传递

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

> 示例代码：

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

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

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

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

12.3 函数返回值

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

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

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

返回多个值时，值之间用“,”隔开。

> 示例代码：定义一个函数，实现两个变量交换值

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

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

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

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

> 示例代码：

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

x = init()
print(x)

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

--output
1
1 lua nil
```

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

> 示例代码：

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

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

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

函数调用的实参列表也是一个列表表达式。考虑下面的例子：

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

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

如果你确保只取函数返回值的第一个值，可以使用括号运算符，例如

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

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

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

十三、Lua 字符串

字符串或串(String)是由数字、字母、下划线组成的一串字符。

Lua 语言中字符串可以使用以下三种方式来表示：

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

以上三种方式的字符串实例如下：

```
string1 = "Lua"
print("\"字符串 1 是\"",string1)
string2 = 'runoob.com'
print("字符串 2 是",string2)

string3 = [["Lua 教程"]]
print("字符串 3 是",string3)
```

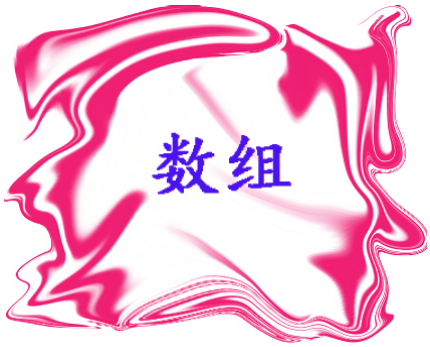
转义字符用于表示不能直接显示的字符，比如后退键，回车键，等。如在字符串转换双引号可以使用 """。

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

13.1 字符串操作

```
string.upper(argument):字符串全部转为大写字母。
string.lower(argument):字符串全部转为小写字母。
string.gsub(mainString,findString,replaceString,num)在字符串中替换。
string.find (str, substr, [init, [end]])在一个指定的目标字符串中搜索指定的内容(第三个参数为索引),返回其具体位置。不存在则返回 nil。
string.reverse(arg)字符串反转
string.format(...)返回一个类似printf的格式化字符串
string.char(arg) 和 string.byte(arg[,int])
string.len(arg)计算字符串长度。
string.rep(string, n)返回字符串string的n个拷贝
.. 链接两个字符串
string.gmatch(str, pattern)回一个迭代器函数，每一次调用这个函数，返回一个在字符串 str 找到的下一个符合 pattern 描述的子串。如果参数 pattern 描述的字符串没有找到，迭代函数返回nil。
string.match()只寻找源字串str中的第一个配对. 参数init可选, 指定搜寻过程的起点, 默认为1。
```

十四、Lua 数组



> Lua并没有提供专门的数组对象来对数组进行操作，但是我们可以使用table来实现数组。

14.1 定义数组

不同于table表，初始化数组时不需要填写key，而数组始终使用数字作为其key：

```
1 arr = {1, "abc", 2, true}--定义数组
2 print(arr[1])--注意索引从 1 开始
```

我们要特别注意的就是Lua中数组的索引是从1开始的。

14.2 使用内置方法处理数组

Lua为我们提供了一些标准的方法来处理数组，我们来具体看看。

table.insert

强指定的值插入到指定的位置，如下：

```
1 arr = {}
2
3 for i = 1, 5 do
4   table.insert(arr,1,i)
5 end
6
7 for key, var in ipairs(arr) do
8     print(key, var)
9 end
```

我们看下输出：

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

我们每次都是把数据插入到第一个位置，所以以前的元素都会后移，故打印出来的值的结果就是从5到1。

table.maxn

获取数组最大的索引值，由于lua索引是从1开始的，所以最大的索引值就是数组元素的总数。

还有其他的操作方法大家可以参考帮助，最后需要特别注意的一点是，小心不要操作到不存在的索引，会导致运行卡死。

14.3 获取数组长度

在Lua中可以使用“#”号和table.maxn两种方法来获取数组的长度，我们看看他们之间的区别：

```
1 arr = {1,2,3,4,5,6}
2 print(#arr)--6
3 print(table.maxn(arr))--6
4
5 arr[9] = 9
6 print(#arr)--6
7 print(table.maxn(arr))--9
```

1. 都仅统计数字key的长度；
2. #号是表示从1递增到空项的长度；
3. table.maxn是表示所有数字key中最大的那个key的索引值；

十五、Lua 迭代器

> “迭代器”就是一种可以遍历一种集合中所有元素的机制。在Lua中迭代器以函数的形式表示，即没掉用一次函数，即可返回集合中的“下一个”元素。迭代器的实现可以借助于闭合函数实现，闭合函数能保持每次调用之间的一些状态。

Lua中内置得迭代函数

- pairs
- ipairs

——pairs与ipairs的区别
1.pairs既能遍历数组形式的表也能遍历键值对形式的表，ipairs只能遍历数组形式的表

```
--tab={key="a",key2="b"}
--for k,v in pairs(tab) do
--    print(v)
--end
--for k,v in ipairs(tab) do
--    print(v)
--end
```

2.pairs会遍历所有不为nil的元素（如果遇到nil则跳过当前元素继续遍历下一个），ipairs从索引1开始遍历遇到nil则停止遍历

```
--tab={1,nil,3}
--for k,v in pairs(tab) do
--    print(v)
--end
--for k,v in ipairs(tab) do
--    print(v)
--end
```

十六、Lua table(表)

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

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

Lua table 是不固定大小的，你可以根据自己需要进行扩容。

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

16.1 table(表)的构造

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

```
-- 初始化表
mytable = {}

-- 指定值
mytable[1]= "Lua"

-- 移除引用
mytable = nil
-- lua 垃圾回收会释放内存
```

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

以下实例演示了以上的描述情况:

```
-- 简单的 table
mytable = {}
print("mytable 的类型是 ",type(mytable))

mytable[1]= "Lua"
mytable["wow"] = "修改前"
print("mytable 索引为 1 的元素是 ", mytable[1])
print("mytable 索引为 wow 的元素是 ", mytable["wow"])

-- alternatetable和mytable的是指同一个 table
alternatetable = mytable

print("alternatetable 索引为 1 的元素是 ", alternatetable[1])
print("mytable 索引为 wow 的元素是 ", alternatetable["wow"])
```

```
alternatetable["wow"] = "修改后"

print("mytable 索引为 wow 的元素是 ", mytable["wow"])

-- 释放变量
alternatetable = nil
print("alternatetable 是 ", alternatetable)

-- mytable 仍然可以访问
print("mytable 索引为 wow 的元素是 ", mytable["wow"])

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

16.2 Table 操作

以下列出了 Table 操作常用的方法：

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

16.3 Table 连接

我们可以使用 concat() 输出一个列表中元素连接成的字符串:

实例

```
--fruits = {"banana","orange","apple"}
--print(table.concat(fruits))
--print(table.concat(fruits,","))
--print(table.concat(fruits,",",2,3))
```

16.4 插入和移除

以下实例演示了 table 的插入和移除操作:

```
-- 插入移除
--fruits = {"banana","orange","apple"}
---- 在末尾插入元素
--table.insert(fruits,"mango")
---- 在索引2 得位置添加水果
--table.insert(fruits,2,"grapes")
--
--table.remove(fruits)
--
--for k,v in pairs(fruits) do
--    print(v)
--end
--
```

16.5 Table 排序

> 以下实例演示了 sort() 方法的使用，用于对 Table 进行排序:

实例

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

16.6 Table 最大值

> table.maxn 在 Lua5.2 之后该方法已经不存在了，我们定义了 table_maxn 方法来实现。

以下实例演示了如何获取 table 中的最大值:

```
tab = {[1] = 2,[2] = 6,[4]=34,[26] = 5}

function table_maxn(t)
    local mn = nil
    for k,v in pairs(t) do
        if (mn == nil) then
            mn = v
        end
        if (mn < v) then
            mn = v
        end
    end
    return mn
```

```
end

print("table 最大值:",table_maxn(tab))
```

十七、Lua 模块与包

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

require 函数

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

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

```
-- 文件名为 module.lua
-- 定义一个名为 module 的模块
module = {}

-- 定义一个常量
module.constant = "这是一个常量"

-- 定义一个函数
function module.func1()
    io.write("这是一个公有函数!\n")
end

local function func2()
    print("这是一个私有函数!")
end

function module.func3()
    func2()
end

return module
```

十八、Lua 元表(Metatable)

> 在 Lua table 中我们可以访问对应的 key 来得到 value 值，但是却无法对两个 table 进行操作(比如相加)。因此 Lua 提供了元表(Metatable)，允许我们改变 table 的行为，每个行为关联了对应的元方法。

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

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

18.1 元表的元方法

函数	描述
__add	运算符 +
__sub	运算符 -
__mul	运算符 *
__ div	运算符 /
__mod	运算符 %
__unm	运算符 -（取反）
__concat	运算符 ..
__eq	运算符 ==
__lt	运算符 <
__le	运算符 <=
__call	当函数调用
__tostring	转化为字符串
__index	调用一个索引
__newindex	给一个索引赋值

由于那几个运算符使用类似，所以就不单独说明了，接下来说 **call，** tostring， **index，** newindex四个元方法。

除了操作符之外，如下元方法也可以被重载，下面会依次解释使用方法：

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

```
local mt = {}

mt.__add = function(t1,t2)
    local temp = {}
    for  _,v in pairs(t1) do
        table.insert(temp,v)
    end
    for  _,v in pairs(t2) do
```

```
        table.insert(temp,v)
    end
    return temp
end


local t1 = {1,2,3}
local t2 = {2}

-- 设置t1的 元素为mt
setmetatable(t1,mt)

local t3 = t1 + t2

for k,v in pairs(t3) do
    print(v)
end


-- 1. 查看t1 是否 有元表 如果有  则查看t1 的元表是否有--add 的方法 如果有就调用
-- 2. 查看t2 是否有元表 如果有  则查看t1 的元表是否有--add 的方法 如果有就调用
-- 3. 若都没有则会报错
```

tostring

```
local mt = {}
mt.__tostring = function(t)
    local s = "{"
    for i,v in ipairs(t)  do
        if (i > 1) then
            s = s..","   -- 字符串拼接
        end
        s  = s..v
    end

    s = s.."}"
    return s
end

t  = {1,2,3}

-- 将t的元素设置为mt
setmetatable(t,mt)
print(t)
--   {1,2,3,4}
```

_call

```
--local mt = {}
--mt.__call  = function(mytable,...)
--    -- 输出所有参数
--    for _,v in ipairs{...} do
--        print(v)
--    end
--end
--
--t = {}
--
--setmetatable(t,mt)
--
--t(1,2,3)
```

__index

```
local mt = {}
mt.__index = function(t,key)
    return "it is "..key
end

t = {1,2,3}
--print(t.key)
setmetatable(t,mt)

print(t.key)
```

十九、Lua 协程

19.1 为什么需要协程?

> 我们都知道多线程，当需要同时执行多项任务的时候，就会采用多线程并发执行。拿手机支付举例子，当收到付款信息的时候，需要查询数据库来判断余额是否充足，然后再进行付款。

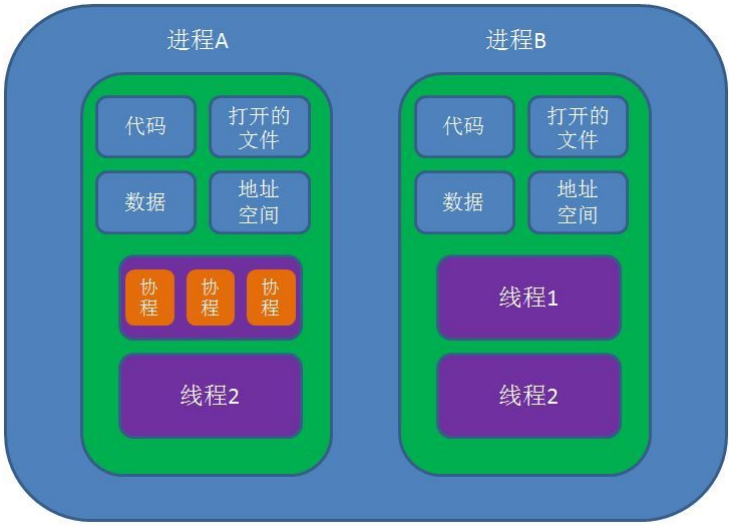
造成原因:

1. 系统线程会占用非常多的内存空间
2. 过多的线程切换会占用大量的系统时间

协程并没有增加线程数量，只是在线程的基础之上通过分时复用的方式运行多个协程

19.2 什么是协程

协程，英文Coroutines，是一种比线程更加轻量级的存在。正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。



最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行）。

这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。

19.3 Lua实现协程

基本语法

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

```
-- coroutine_test.lua 文件
co = coroutine.create(
    function(i)
        print(i);
    end
)

coroutine.resume(co, 1)   -- 1
print(coroutine.status(co))  -- dead

print("----------")

co = coroutine.wrap(
```

```
    function(i)
        print(i);
    end
)

co(1)

print("----------")

co2 = coroutine.create(
    function()
        for i=1,10 do
            print(i)
            if i == 3 then
                print(coroutine.status(co2))  --running
                print(coroutine.running()) --thread:XXXXXX
            end
            coroutine.yield()
        end
    end
)

coroutine.resume(co2) --1
coroutine.resume(co2) --2
coroutine.resume(co2) --3

print(coroutine.status(co2))   -- suspended
print(coroutine.running())

print("----------")
```

深入理解

```
function foo (a)
    print("foo 函数输出", a)
    return coroutine.yield(2 * a) -- 返回  2*a 的值
end

co = coroutine.create(function (a , b)
    print("第一次协同程序执行输出", a, b) -- co-body 1 10
    local r = foo(a + 1)

    print("第二次协同程序执行输出", r)
    local r, s = coroutine.yield(a + b, a - b)  -- a，b的值为第一次调用协同程序时传入

    print("第三次协同程序执行输出", r, s)
    return b, "结束协同程序"                    -- b的值为第二次调用协同程序时传入
end)

print("main", coroutine.resume(co, 1, 10)) -- true, 4
print("--分割线----")
print("main", coroutine.resume(co, "r")) -- true 11 -9
print("---分割线---")
print("main", coroutine.resume(co, "x", "y")) -- true 10 end
print("---分割线---")
print("main", coroutine.resume(co, "x", "y")) -- cannot resume dead coroutine
print("---分割线---")
```

以上实例接下如下：

- 调用resume，将协同程序唤醒,resume操作成功返回true，否则返回false；
- 协同程序运行；
- 运行到yield语句；
- yield挂起协同程序，第一次resume返回；（注意：此处yield返回，参数是resume的参数）
- 第二次resume，再次唤醒协同程序；（注意：此处resume的参数中，除了第一个参数，剩下的参数将作为yield的参数）
- yield返回；
- 协同程序继续运行；
- 如果使用的协同程序继续运行完成后继续调用 resume方法则输出：cannot resume dead coroutine

resume和yield的配合强大之处在于，resume处于主程中，它将外部状态（数据）传入到协同程序内部；而yield则将内部的状态（数据）返回到主程中。

二十、Lua 文件io

> Lua I/O 库用于读取和处理文件。分为简单模式（和C一样）、完全模式。

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

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

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

20.1 简单模式

```
-- 以只读方式打开文件
file = io.open("test.lua", "r")

-- 设置默认输入文件为 test.lua
io.input(file)
```

```
-- 输出文件第一行
print(io.read())

-- 关闭打开的文件
io.close(file)

-- 以附加的方式打开只写文件
file = io.open("test.lua", "a")

-- 设置默认输出文件为 test.lua
io.output(file)

-- 在文件最后一行添加 Lua 注释
io.write("--  test.lua 文件末尾注释")

-- 关闭打开的文件
io.close(file)
```

20.2 完全模式

> 通常我们需要在同一时间处理多个文件。我们需要使用 file:function_name 来代替 io.function_name 方法。以下实例演示了如何同时处理同一个文件:

```
-- 以只读方式打开文件
file = io.open("test.lua", "r")

-- 输出文件第一行
print(file:read())

-- 关闭打开的文件
file:close()

-- 以附加的方式打开只写文件
file = io.open("test.lua", "a")

-- 在文件最后一行添加 Lua 注释
file:write("--test")

-- 关闭打开的文件
file:close()
```

二十一、 Lua 错误处理

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

任何程序语言中，都需要错误处理。错误类型有:

- 语法错误
- 运行错误

21.1 语法错误

语法错误通常是由于对程序的组件（如运算符、表达式）使用不当引起的。一个简单的实例如下：

```
-- test.lua 文件
a == 2
```

以上代码执行结果为：

```
lua: test.lua:2: syntax error near '=='
```

正如你所看到的，以上出现了语法错误，一个 "=" 号跟两个 "=" 号是有区别的。一个 "=" 是赋值表达式两个 "=" 是比较运算。

21.2 运行错误

运行错误是程序可以正常执行，但是会输出报错信息。如下实例由于参数输入错误，程序执行时报错：

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

add(10)
```

二十二、Lua 垃圾回收

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

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

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

print(collectgarbage("count"))

mytable = nil

print(collectgarbage("count"))

print(collectgarbage("collect"))

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

二十三、Lua 面向对象

23.1 Lua 面向对象

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

以下几种编程语言都支持面向对象编程：

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

23.2 面向对象特征

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

23.3 Lua 中面向对象

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

```
local _M = {}

local mt = { __index = _M }

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

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

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

return _M


local account = require("account")

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

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

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

> 在lua中，表拥有一个标识：self。self类似于this指针，大多数面向对象语言都隐藏了这个机制，在编码时不需要显示的声明这个参数，就可以在方法内使用this（例如C++和C#）。在lua中，提供了冒号操作符来隐藏这个参数，例如：

```
local t = {a = 1, b = 2}
function t:Add()
    return (self.a + self.b)
end

print(t:Add())
```

冒号的作用有两个：1. 对于方法定义来说，会增加一个额外的隐藏形参（self）； 2. 对于方法调用来说，会增加一个额外的实参（表自身）

冒号只是一种语法机制，提供的是便利性，并没有引入任何新的东西。使用冒号完成的事情，都可以使用点语法来完成。看下面的例子：