

Openresty实战应用

一、快速上手LUA

Lua是 OpenResty 中使用的编程语言，掌握它的基本语法还是很有必要的。

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”。从作者所在的国家来看，NGINX 诞生于俄罗斯，Lua 诞生于巴西，OpenResty 诞生于中国，这三门同样精巧的开源技术都出自金砖国家，而不是欧美，也是挺有趣的一件事。

回到Lua语言上。事实上，Lua 在设计之初，就把自己定位为一个简单、轻量、可嵌入的胶水语言，没有走大而全的路线。虽然你平常工作中可能没有直接编写 Lua 代码，但 Lua 的使用其实非常广泛。很多的网游，比如魔兽世界，都会采用 Lua 来编写插件；而键值数据库 Redis 则是内置了 Lua 来控制逻辑。

另一方面，虽然 Lua 自身的库比较简单，但它可以方便地调用 C 库，大量成熟的 C 代码都可以为其所用。比如在 OpenResty 中，很多时候都需要你调用 NGINX 和 OpenSSL 的 C 函数，而这都得益于 Lua 和 LuaJIT 这种方便调用 C 库的能力。

1、lua是什么？

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

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

2、Lua 和 LuaJIT 的区别

Lua 非常高效，它运行得比许多其它脚本(如 Perl、Python、Ruby)都快，这点在第三方的独立测评中得到了证实。

尽管如此，仍然会有人不满足，他们总觉得“嗯，还不够快!”。LuaJIT 就是一个为了再榨出一些速度的尝试，它利用即时编译 (Just-in Time) 技术把 Lua 代码编译成本地机器码后交由 CPU 直接执行。

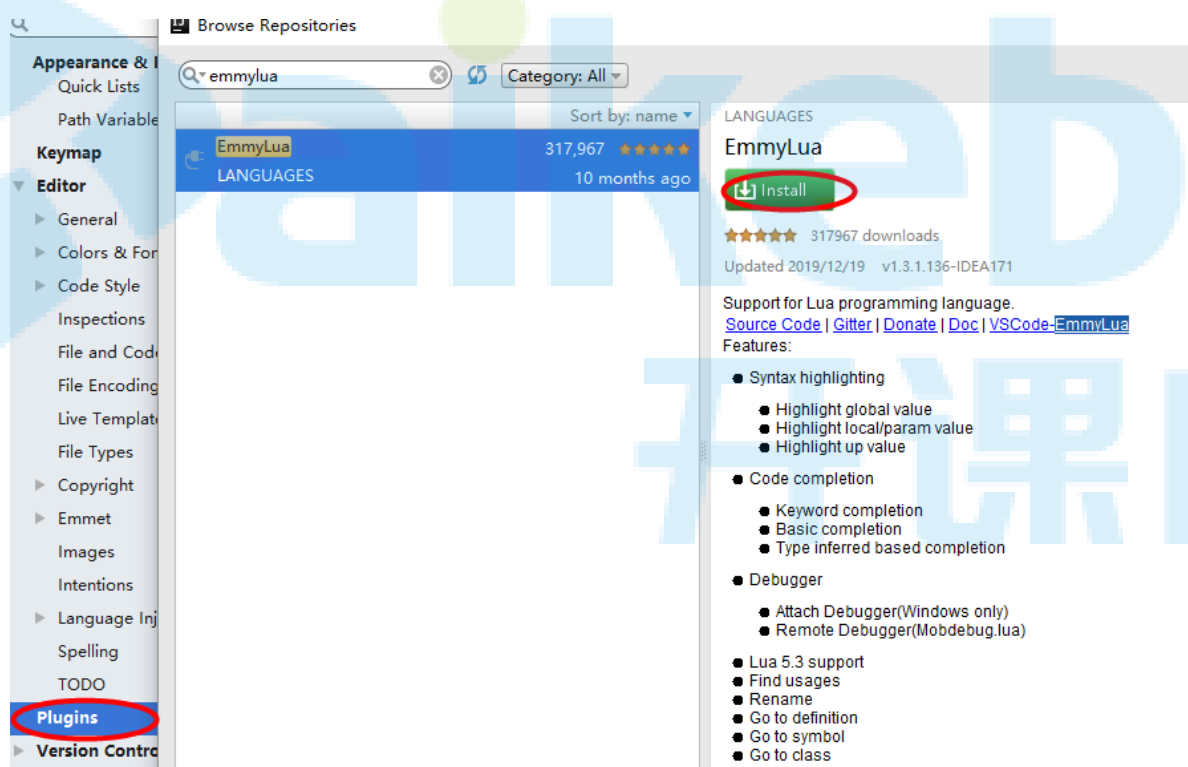
LuaJIT 2 的测评报告表明，在数值运算、循环与函数调用、协程切换、字符串操作等许多方面它的加速效果都很显著。凭借着 FFI 特性，LuaJIT 2 在那些需要频繁地调用外部 C/C++ 代码的场景，也要比标准 Lua 解释器快很多。目前 LuaJIT 2 已经支持包括 i386、x86_64、ARM、PowerPC 以及 MIPS 等多种不同的体系结构。

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

3、编译器选择

1) 下载插件

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



详细帮助文档参考此地址（官方文档）：https://emmylua.github.io/zh_CN/

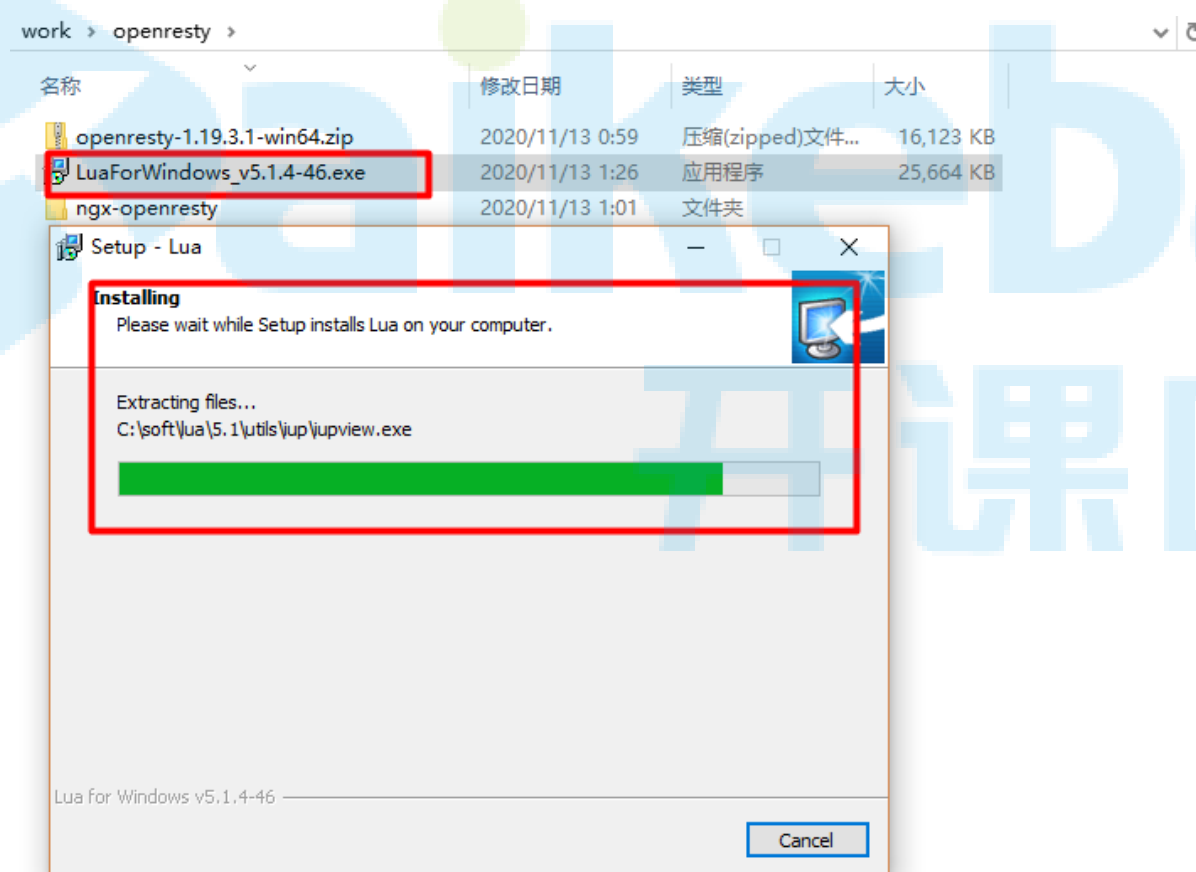
github地址：<https://github.com/EmmyLua/IntelliJ-EmmyLua>

功能简要说明

- 语法高亮
- 全局/local/参数 等类型高亮
- upvalue 高亮
- 查找引用(Find usages)
- 重命名(Rename(Shift+F6))
- 跳转到定义(Go to definition(Ctrl + Mouse))
- 快速跳转到文件(Navigate to file (Ctrl + Shift + N))

- 快速跳转到符号(Navigate to symbol(Ctrl + Alt + Shift + N))
- 快速跳转到类(Navigate to class(Ctrl + N))
- 格式化
- 自定义高亮颜色(Color settings page)
- 代码完成提示
- 基于注解的代码完成提示(--@class ---@type 等等)
- 大纲/快速大纲(Structure view)
- 注释/反注释(Comment in/out)
- 本地附加调试(目前只支持Windows32/64位程序)
- 远程调试(基于mobdebug.lua, 适用所有平台)
- 代码模板
- Postfix completion templates
- Code intentions
- Code inspections
- Region folding
- 文档(Quick Documentation(Ctrl + Q))
- 支持lua5.3语法

2) 下载luaforwindows环境



4、Lua环境

我们不用专门去安装标准 Lua 5.1 之类的环境，因为 OpenResty 已经不再支持标准 Lua，而只支持 LuaJIT。

这里我介绍的 Lua 语法，也是和 LuaJIT 兼容的部分，而不是基于最新的 Lua 5.3，这一点需要你特别注意。

在 OpenResty 的安装目录下，你可以找到 LuaJIT 的目录和可执行文件。我这里是 Mac 环境，使用 brew 安装 OpenResty，所以你本地的路径很可能和下面的不同：

```
# lua环境
ll /usr/local/Cellar/openresty/1.13.6.2/luajit/bin/luajit
lrwxr-xr-x 1 ming admin 18B 4 2 14:54
/usr/local/Cellar/openresty/1.13.6.2/luajit/bin/luajit -> lua
```

你也可以在系统的可执行文件目录中找到它：

```
# luajit
which luajit
/usr/local/bin/luajit
```

并查看 LuaJIT 的版本号：

```
luajit -v
LuaJIT 2.1.0-beta2 -- Copyright (C) 2005-2017 Mike Pall. http://luajit.org/
```

5、hello world

查清楚这些信息后，你可以新建一个 1.lua 文件，并用 luajit 来运行其中的 hello world 代码：

```
cat 1.lua
print("hello world")
luajit 1.lua
hello world
```

当然，你还可以使用 resty 来直接运行，要知道，它最终也是用 LuaJIT 来执行的：

```
resty -e 'print("hello world")'
hello world
```

上述两种运行 hello world 的方式都是可行的。不顾对我来说，我更喜欢 resty 这种方式，因为后面很多 OpenResty 的代码，也都是通过 resty 来运行的。

二、Lua 基础数据类型

函数 type 能够返回一个值或一个变量所属的类型。

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

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
```

值得一提的是，OpenResty 的 Lua 接口还提供了一种特殊的空值，即 `ngx.null`，用来表示不同于 nil 的“空值”。

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

如下：

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

-- do something
person = nil
-- do something

print(person.name)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

返回的结果如下：

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

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

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

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

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
```

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
```

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

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

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

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。

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

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

print(str1)      -->output:hello world
print(str2)      -->output:hello lua
print(str3)      -->output:"add\nname",'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 $O(n)$ 。

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。

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

6、lua正则

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

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

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

- `string.find` 的基本应用是在目标串内搜索匹配指定的模式的串。函数如果找到匹配的串，就返回它的开始索引和结束索引，否则返回 `nil`。`find` 函数第三个参数是可选的：标示目标串中搜索的起始位置，例如当我们想实现一个迭代器时，可以传进上一次调用时的结束索引，如果返回了一个 `nil` 值的话，说明查找结束了。

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

- `string.gmatch` 我们也可以使用返回迭代器的方式。

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

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

7、虚变量

当一个方法返回多个值时，有些返回值有时候用不到，要是声明很多变量来一一接收，显然不太合适（不是不能）。Lua 提供了一个虚变量(dummy variable)的概念，按照惯例以一个下划线（"_"）来命名，用它来表示丢弃不需要的数值，仅仅起到占位的作用。

看一段示例代码：

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

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

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

local _,finish = string.find("hello", "he") --采用虚变量（即下划线），接收起
--始下标值，然后丢弃，finish接收
--结束下标值
print ( finish ) --输出 2
print ( _ ) --输出 1, `_` 只是一个普通变量，我们习惯
上不会读取它的值
```

代码倒数第二行，定义了一个用 `local` 修饰的 虚变量（即 单个下划线）。使用这个虚变量接收 `string.find()` 第一个返回值，忽略不用，直接使用第二个返回值。

虚变量不仅仅可以被用在返回值，还可以用在迭代等。

在for循环中的使用：

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

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

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

执行结果：

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

part data:
1
3
5
```

当有多个返回值需要忽略时,可以重复使用同一个虚变量:

多个占位:

```
-- test.lua 文件
function foo()
    return 1, 2, 3, 4
end

local _, _, bar = foo();    -- 我们只需要第三个
print(bar)
```

执行结果：

```
# luajit test.lua
3
```

8、点号与冒号操作符的区别

看下面示例代码：

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

执行结果:

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

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

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

示例代码:

```
obj = { x = 20 }

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

等价于

```
obj = { x = 20 }

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

9、function (函数)

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

示例

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

local a = foo    --把函数赋给变量

print(a())

--output:
in the function
30
```

有名函数的定义本质上是匿名函数对变量的赋值。为说明这一点, 考虑

```
function foo()
end
```

等价于

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

类似地,

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

等价于

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

Lua 里面的函数必须放在调用的代码之前, 下面的代码是一个常见的错误:

```
-- test.lua 文件  
local i = 100  
i = add_one(i)  
  
function add_one(i)  
    return i + 1  
end
```

我们将得到如下错误:

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

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

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

等价于

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

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

三、lua表达式

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

示例代码：test1.lua

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

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

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

```
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：

```
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。

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

```
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`。

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。

5、优先级

Lua 操作符的优先级如下表所示(从高到低)：

优先级
\wedge
not # -
* / %
+ -
..
< > <= >= == ~=
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，就应显式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。

四、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。

1、控制结构 if-else

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

1.1、单个 if 分支 型

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

运行输出：x is a positive number

1.2、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
```

运行输出：x is a positive number

1.3、多个分支 if-elseif-else 型

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

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

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

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

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

2、while 型控制结构

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

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

```
while 表达式 do
    --body
end
```

示例代码，求 1 + 2 + 3 + 4 + 5 的结果

```
x = 1
sum = 0

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

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

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

```

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

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

```

3、repeat 控制结构

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

以下代码将会形成死循环：

```

x = 10
repeat
    print(x)
until false

```

该代码将导致死循环，因为until的条件一直为假，循环不会结束

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

4、for 控制结构

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

4.1、for 数字型

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

数字型 for 的语法如下：

```

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

```

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

示例

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

-- output:

```
1
2
3
4
5
```

...

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

-- output:

```
1
3
5
7
9
```

以下是这种循环的一个典型示例：

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

-- output:

...

如果不想给循环设置上限的话，可以使用常量 `math.huge`：

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

4.2、for 泛型

泛型 for 循环通过一个迭代器（iterator）函数来遍历所有值：

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
local revDays = {}
for k, v in pairs(days) do
    revDays[v] = k
end
```

```
-- print value
for k,v in pairs(revDays) do
    print("k:", k, " v:", v)
end
```

-- output:

```
k: Tuesday    v: 2
k: Monday     v: 1
k: Sunday     v: 7
k: Thursday   v: 4
k: Friday     v: 5
k: Wednesday  v: 3
k: Saturday   v: 6
```

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

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

5、break, return 和 goto

5.1、break

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

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

在实际应用中, `break` 经常用于嵌套循环中。

5.2、return

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

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

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

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

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

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

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

5.3、goto

LuaJIT 一开始对标的是 Lua 5.1，但渐渐地也开始加入部分 Lua 5.2 甚至 Lua 5.3 的有用特性。`goto` 就是其中一个不得不提的例子。

有了 `goto`，我们可以实现 `continue` 的功能：

```
for i=1, 3 do
    if i <= 2 then
        print(i, "yes continue")
        goto continue
    end

    print(i, " no continue")

    ::continue::
    print([[i'm end]])
end
```

输出结果：

```
$ luajit test.lua
1  yes continue
i'm end
2  yes continue
i'm end
3   no continue
i'm end
```

在 [GotoStatement](#) 这个页面上，你能看到更多用 `goto` 玩转控制流的脑洞。

`goto` 的另外一项用途，就是简化错误处理的流程。有些时候你会发现，直接 `goto` 到函数末尾统一的错误处理过程，是更为清晰的写法。

```
local function process(input)
    print("the input is", input)
    if input < 2 then
        goto failed
    end
    -- 更多处理流程和 goto err

    print("processing...")
    do return end
    ::failed::
    print("handle error with input", input)
end

process(1)
process(3)
```

五、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。

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，找出 -12 和 20 中的最大值
print(m)                  --> output 20

```

如果参数列表为空，必须使用 `()` 表明是函数调用。

示例代码：

```

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

func()                 --函数调用，圆括号不能省

--> 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。

2、函数的参数

2.1、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
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
```

2.2、变长参数

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

示例代码：

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

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

    print(ans)
end

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

-->output
1 2

1 2 3 4
```

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

2.3、具名参数

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

示例代码：

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

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

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

2.4、按引用传递

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

示例代码：

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

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

change(rectangle)
print("after change:", "width = ", rectangle.width,
      " height =", rectangle.height)

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

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

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。

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

返回多个值时，值之间用“,”隔开。

示例代码：定义一个函数，实现两个变量交换值

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

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

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

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

示例代码：

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

x = init()
print(x)

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

--output
1
1 lua nil
```

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

示例代码：

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

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

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

函数调用的实参列表也是一个列表表达式。考虑下面的例子：

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

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

如果你确保只取函数返回值的第一个值，可以使用括号运算符，例如

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

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

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。

4、全动态函数调用

调用回调函数，并把一个数组参数作为回调函数的参数。

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

4.1、使用场景

如果你的实参 table 中确定没有 nil 空洞，则可以简化为

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

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

伪代码

```
add_task(end_time, callback, params)

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

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

4.2、小试牛刀

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

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

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

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

六、lua模块

1、lua模块方式

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

有东西，比如函数和变量。`require` 函数会返回 Lua 模块加载后的结果，即用于表示该 Lua 模块的 Lua 值。

require 函数：

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

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

把下面的代码保存在文件 my.lua 中

```
local _M = {}

local function get_name()
    return "Lucy"
end

function _M.greeting()
    print("hello " .. get_name())
end

return _M
```

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

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

注：对于需要导出给外部使用的公共模块，处于安全考虑，是要避免全局变量的出现。我们可以使用 `lj-releng` 或 `luacheck` 工具完成全局变量的检测。至于如何做，到后面再讲。

另一个要注意的是，由于在 LuaJIT 中，`require` 函数内不能进行上下文切换，所以不能够在模块的顶级上下文中调用 `cosocket` 一类的 API。否则会报 `attempt to yield across C-call boundary` 错误。

2、lua特别之处

2.1、lua下标从1开始

Lua 是我知道的唯一一个下标从 1 开始的编程语言。这一点，虽然对于非程序员背景的人来说更好理解，但却容易导致程序的 bug。

下面是一个例子：

```
resty -e 't={100}; ngx.say(t[0])'
```

你自然期望打印出 100，或者报错说下标 0 不存在。但结果出乎意料，什么都没有打印出来，也没有报错。既然如此，让我们加上 `type` 命令，来看下输出到底是什么：


```
resty -e 't={100};ngx.say(type(t[0]))'  
nil
```

原来是空值。事实上，在 OpenResty 中，对于空值的判断和处理也是一个容易让人迷惑的点，后面我们讲到 OpenResty 的时候再细聊。

2.2、拼接字符串

和大部分语言使用 + 不同，Lua 中使用两个点号来拼接字符串：

```
resty -e "ngx.say('hello' .. ', world')"  
hello, world
```

在实际的项目开发中，我们一般都会使用多种开发语言，而 Lua 这种不走寻常路的设计，总是会让开发者的思维，在字符串拼接的时候卡顿一下，也是让人哭笑不得。

2.3、只有table一种数据结构

不同于 Python 这种内置数据结构丰富的语言，Lua 中只有一种数据结构，那就是 table，它里面可以包括数组和哈希表：

```
local color = {first = "red", "blue", third = "green", "yellow"}  
print(color["first"]) --> output: red  
print(color[1]) --> output: blue  
print(color["third"]) --> output: green  
print(color[2]) --> output: yellow  
print(color[3]) --> output: nil
```

如果不显式地用 键值对 的方式赋值，table 就会默认用数字作为下标，从 1 开始。所以 color[1] 就是 blue。另外，想在 table 中获取到正确长度，也是一件不容易的事情，我们来看下面这些例子：

```
local t1 = { 1, 2, 3 }  
print("Test1 " .. table.getn(t1))  
local t2 = { 1, a = 2, 3 }  
print("Test2 " .. table.getn(t2))  
local t3 = { 1, nil }  
print("Test3 " .. table.getn(t3))  
local t4 = { 1, nil, 2 }  
print("Test4 " .. table.getn(t4))
```

使用 resty 运行的结果如下：

```
Test1 3  
Test2 2  
Test3 1  
Test4 1
```

你可以看到，除了第一个返回长度为 3 的测试案例外，后面的测试都是我们预期之外的结果。事实上，想在 Lua 中获取 table 长度，必须注意到，只有在 table 是 *序列* 的时候，才能返回正确的值。那什么是序列呢？首先序列是数组（array）的子集，也就是说，table 中的元素都可以用正整数下标访问，不存在键值对的情况。对应到上面的代码中，除了 t2 外，其他的 table 都是 array。其次，序列中不包含空洞（hole），即 nil。综合这两点来看，上面的 table 中，t1 是一个序列，而 t3 和 t4 是 array，却不是序列（sequence）。到这里，你可能还有一个疑问，为什么 t4 的长度会是 1 呢？其实这是因为，在遇到 nil 时，获取长度的逻辑就不继续往下运行，而是直接返回了。

2.4、默认全局变量

我想先强调一点，除非你相当确定，否则在 Lua 中声明变量时，前面都要加上 local：

```
local s = 'hello'
```

这是因为在 Lua 中，变量默认是全局的，会被放到名为 `_G` 的 table 中。不加 local 的变量会在全局表中查找，这是昂贵的操作。如果再加上一些变量名的拼写错误，就会造成难以定位的 bug。所以，在 OpenResty 编程中，我强烈建议你总是使用 local 来声明变量，即使在 require module 的时候也是一样：

```
-- Recommended
local xxx = require('xxx')
-- Avoid
require('xxx')
```

七、FFI

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

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

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

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

1、ffi 库词汇

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

2、ffi.* API

功能: Lua ffi 库的 API, 与 LuaJIT 不可分割。

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

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

3、ffi.cdef

语法: ffi.cdef(def)

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

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

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

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

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

接下来在 Linux 下生成动态链接库:

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

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

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

在 Lua 代码中要增加如下的行：

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

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

下面看一个完整例子：

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

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

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

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

完整的代码如下所示：

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

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

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

4、ffi.typeof

语法： `ctype = ffi.typeof(ct)`

功能： 创建一个 `ctype` 对象，会解析一个抽象的 C 类型定义。

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

5、ffi.new

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

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

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

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

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

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

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

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

6、ffi.fill

语法: `ffi.fill(dst, len [,c])`

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

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

7、ffi.cast

语法: `cdata = ffi.cast(ct, init)`

功能: 创建一个 scalar cdata 对象。

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

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

8、cdata 对象的垃圾回收

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

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

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

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

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

9、调用 C 函数

真的很用容易去调用一个外部 C 库函数, 示例代码:

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

以上操作步骤, 如下:

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

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

再来一个源自官方的示例代码:

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

local function compress(txt)
    local n = zlib.compressBound(#txt)
```

```

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

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

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

```

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

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

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

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

10、使用 C 数据结构

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

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

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

下面是一个使用 C 数据结构的实例

```

local ffi = require("ffi")

```



```

ffi.cdef[[
typedef struct { double x, y; } point_t;
]]

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

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

```

附表：Lua 与 C 语言语法对应关系

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

11、小心内存泄漏

所谓“能力越大，责任越大”，FFI 库在允许我们调用 C 函数的同时，也把内存管理的重担压到我们的肩上。还好 FFI 库提供了很好用的 `ffi.gc` 方法。该方法允许给 cdata 对象注册在 GC 时调用的回调，它能让你在 Lua 领域里完成 C 手工释放资源的事。

C++ 提倡用一种叫 RAII 的方式管理你的资源。简单地说，就是创建对象时获取，销毁对象时释放。我们可以在 LuaJIT 的 FFI 里借鉴同样的做法，在调用 `resource = ffi.C.xx_create` 等申请资源的函数之后，立即补上一行 `ffi.gc(resource, ...)` 来注册释放资源的函数。尽量避免尝试手动释放资源！即使不考虑 `error` 对执行路径的影响，在每个出口都补上一模一样的逻辑会够你受的（用 `goto` 也差不多，只是稍稍好一点）。

有些时候，`ffi.C.xx_create` 返回的不是具体的 cdata，而是整型的 handle。这会儿需要用 `ffi.metatype` 把 `ffi.gc` 包装一下：

```
local resource_type = ffi.metatype("struct {int handle;}", {
    __gc = free_resource
})

local function free_resource(handle)
    ...
end

resource = ffi.new(resource_type)
resource.handle = ffi.C.xx_create()
```

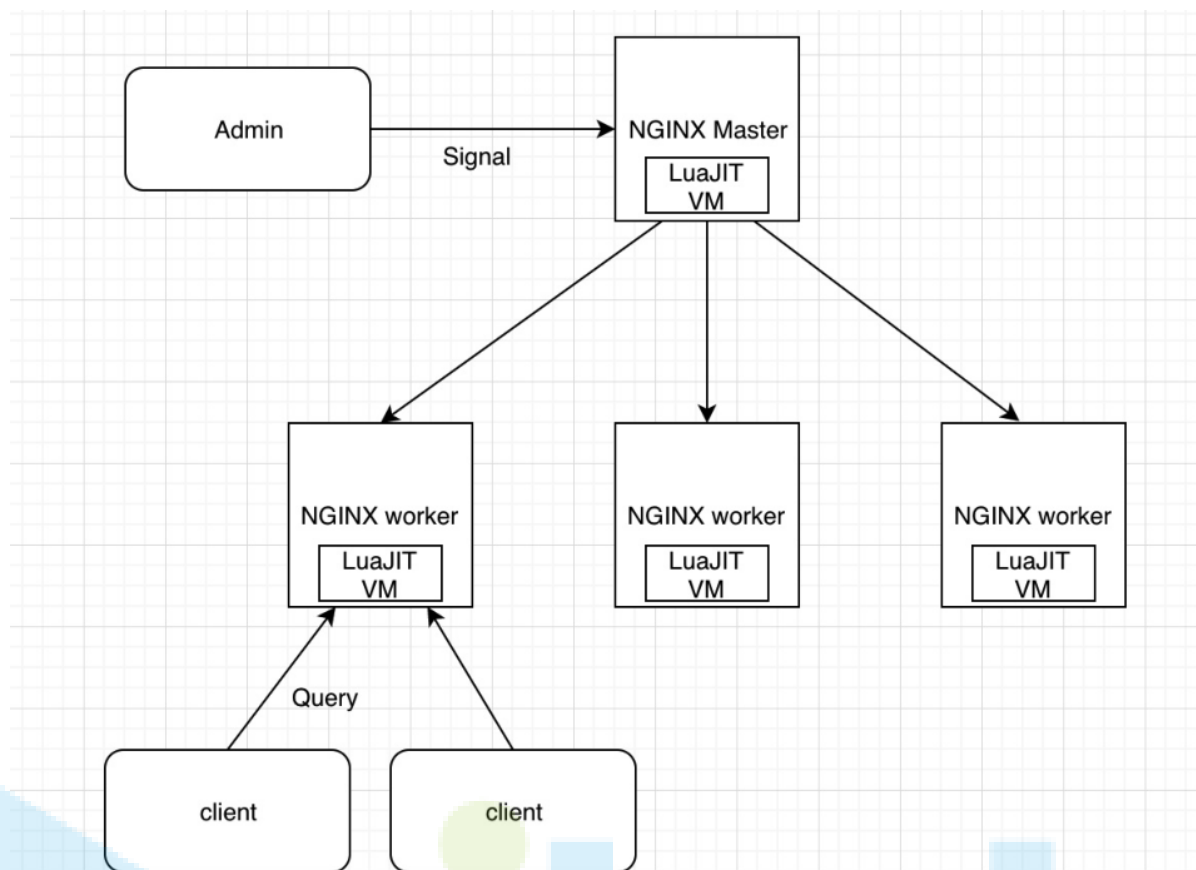
如果你没能把申请资源和释放资源的步骤放一起，那么内存泄露多半会在前方等你。写代码的时候切记这一点。

八、luaJIT&Lua

1、LuaJIT

OpenResty 的另一块基石：LuaJIT。

当然，在 OpenResty 中，写出正确的 LuaJIT 代码的门槛并不高，但要写出高效的 LuaJIT 代码绝非易事；



前面我们提到过，OpenResty 的 worker 进程都是 fork master 进程而得到的，其实，master 进程中的 LuaJIT 虚拟机也会一起 fork 过来。在同一个 worker 内的所有协程，都会共享这个 LuaJIT 虚拟机，Lua 代码的执行也是在这个虚拟机中完成的。这可以算是 OpenResty 的基本原理，后面课程我们再详细聊聊。今天我们先来理顺 Lua 和 LuaJIT 的关系。

2、lua&JIT

先把重要的事情放在前面说：

标准 Lua 和 LuaJIT 是两回事儿，LuaJIT 只是兼容了 Lua 5.1 的语法。

标准 Lua 现在的最新版本是 5.3，LuaJIT 的最新版本则是 2.1.0-beta3。在 OpenResty 几年前的老版本中，编译的时候，你可以选择使用标准 Lua VM，或者 LuaJIT VM 来作为执行环境，不过，现在已经去掉了对标准 Lua 的支持，只支持 LuaJIT。

LuaJIT 的语法兼容 Lua 5.1，并对 Lua 5.2 和 5.3 做了选择性支持。所以我们应该先学习 Lua 5.1 的语法，并在此基础上学习 LuaJIT 的特性。

3、luaJIT特别之处

明白了Lua这四点特别之处，我们继续来说LuaJIT。除了兼容 Lua 5.1 的语法并支持 JIT 外，LuaJIT 还紧密结
合了 FFI (Foreign Function Interface) ，可以让你直接在 Lua 代码中调用外部的 C 函数和使用 C 的数据结
构。
下面是一个最简单的例子：

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

短短这几行代码，就可以直接在 Lua 中调用 C 的 printf 函数，打印出 Hello world!。你可以使用
resty 命令来运行它，看下是否成功。
类似的，我们可以用 FFI 来调用 NGINX、OpenSSL 的 C 函数，来完成更多的功能。实际上，FFI 方式比
传统的 Lua/C API 方式的性能更优，这也是 lua-resty-core 项目存在的意义

九、初步认识openresty

1、openresty的发展

OpenResty 并不像其他的开发语言一样从零开始搭建，而是基于成熟的开源组件——NGINX 和
LuaJIT。

OpenResty 诞生于 2007 年，不过，它的第一个版本并没有选择 Lua，而是用了 Perl，这跟作者章亦春
的技
术偏好有很大关系。

但 Perl 的性能远远不能达到要求，于是，在第二个版本中，Perl 就被 Lua 给替换了。不过，在
OpenResty
官方的项目中，Perl 依然占据着重要的角色，OpenResty 工程化方面都是用 Perl 来构建，比如测试框
架、
Linter、CLI 等，后面我们也会逐步介绍。

后来，章亦春离开了淘宝，加入了美国的 CDN 公司 Cloudflare。因为 OpenResty 高性能和动态的优
势很适
合 CDN 的业务需求，很快，OpenResty 就成为 CDN 的技术标准。通过丰富的 lua-resty 库，
OpenResty
开始逐渐摆脱 NGINX 的影子，形成自己的生态体系，在 API 网关、软WAF 等领域被广泛使用。
其实，我经常说，OpenResty 是一个被广泛使用的技术，但它并不能算得上是热门技术，这听上去有点
矛
盾，到底什么意思呢？

说它应用广，是因为 OpenResty 现在是全球排名第五的 Web 服务器。我们经常用到的 12306 的余
票查询
功能，或者是京东的商品详情页，这些高流量的背后，其实都是 OpenResty 在默默地提供服务。
说它并不热门，那是因为使用 OpenResty 来构建业务系统的比例并不高。使用者大都用OpenResty来
处理
入口流量，并没有深入到业务里面去，自然，对于 OpenResty 的使用也是浅尝辄止，满足当前的需求就可

以了。这当然也与 OpenResty 没有像 Java、Python 那样有成熟的 Web 框架和生态有关。说了这么多，接下来，我重点来介绍下，OpenResty 这个开源项目值得称道和学习的几个地方。

2、openresty三大特性

1) 详尽的文档和测试用例

没错，文档和测试是判断开源项目是否靠谱的关键指标，甚至是排在代码质量和性能之前的。OpenResty 的文档非常详细，作者把每一个需要注意的点都写在了文档中。绝大部分时候，我们只需要仔细查看文档，就能解决遇到的问题，而不用谷歌搜索或者是跟踪到源码中。为了方便起见，OpenResty 还自带了一个命令行工具 `restydoc`，专门用来帮助你通过 shell 查看文档，避免编码过程被打断。

不过，文档中只会有一两个通用的代码片段，并没有完整和复杂的示例，到哪里可以找到这样的例子呢？

对于 OpenResty 来说，自然是 `/t` 目录，它里面就是所有的测试案例。每一个测试案例都包含完整的 NGINX

配置和 Lua 代码，以及测试的输入数据和预期的输出数据。不过，OpenResty 使用的测试框架，与其他断言风格的测试框架完全不同，后面我会用专门章节来做介绍。

2) 同步非阻塞

协程，是很多脚本语言为了提升性能，在近几年新增的特性。但它们实现得并不完美，有些是语法糖，有些还需要显式的关键字声明。

OpenResty 则没有历史包袱，在诞生之初就支持了协程，并基于此实现了 同步非阻塞的编程模式。这一点

是很重要的，毕竟，程序员也是人，代码应该更符合人的思维习惯。显式的回调和异步关键字会打断思路，也给调试带来了困难。

这里我解释一下，什么是同步非阻塞。先说同步，这个很简单，就是按照代码来顺序执行。比如下面这段伪码：

```
local res, err = query-mysql(sql)
local value, err = query-redis(key)
```

在同一请求连接中，如果要等 MySQL 的查询结果返回后，才能继续去查询 Redis，那就是同步；如果不用

等 MySQL 的返回，就能继续往下走，去查询 Redis，那就是异步。对于 OpenResty 来说，绝大部分都是同

步操作，只有 `ngx.timer` 这种后台定时器相关的 API，才是异步操作。

再来说说非阻塞，这是一个很容易和“异步”混淆的概念。这里我们说的“阻塞”，特指阻塞操作系统线程。我们继续看上面的例子，假设查询 MySQL 需要 1s 的时间，如果在这 1s 内，操作系统的资源（CPU）是

空闲着并傻傻地等待返回，那就是阻塞；如果 CPU 趁机去处理其他连接的请求，那就是非阻塞。非阻塞也

是 C10K、C100K 这些高并发能够实现的关键。

同步非阻塞这个概念很重要，建议你仔细琢磨一下。我认为，这一概念最好不要通过类比来理解，因为不当的类比，很可能把你搞得更糊涂。

在 OpenResty 中，上面的伪码就可以直接实现同步非阻塞，而不用任何显式的关键字。这里也再次体现了，让开发者用起来更简单，是 OpenResty 的理念之一。

3) 动态

OpenResty 有一个非常大的优势，并且还没有被充分挖掘，就是它的 动态。

传统的 Web 服务器，比如 NGINX，如果发生任何的变动，都需要你去修改磁盘上的配置文件，然后重新加载才能生效，这也是因为它们并没有提供 API，来控制运行时的行为。所以，在需要频繁变动的微服务领域，NGINX 虽然有多次尝试，但毫无建树。而异军突起的 Envoy，正是凭着 xDS 这种动态控制的 API，大有对 NGINX 造成降维攻击的威胁。

和 NGINX、Envoy 不同的是，OpenResty 是由脚本语言 Lua 来控制逻辑的，而动态，便是 Lua 天生的优势。通过 OpenResty 中 lua-nginx-module 模块中提供的 Lua API，我们可以动态地控制路由、上游、SSL 证书、请求、响应等。甚至更进一步，你可以在不重启 OpenResty 的前提下，修改业务的处理逻辑，并不局限于 OpenResty 提供的 Lua API。

这里有一个很合适的类比，可以帮你理解上面关于动态的说明。你可以把 Web 服务器当做是一个正在高速

公路上飞驰的汽车，NGINX 需要停车才能更换轮胎，更换车漆颜色；Envoy 可以一边跑一边换轮胎和颜色；而 OpenResty 除了具备前者能力外，还可以在不停车的情况下，直接把汽车从 SUV 变成跑车。显然，掌握这种“逆天”的能力后，OpenResty 的能力圈和想象力就扩展到了其他领域，比如 Serverless 和边缘计算等。

3、openresty重点

讲了这么多OpenResty的重点特性，你又该怎么学呢？我认为，学习需要抓重点，围绕主线来展开，而不是眉毛胡子一把抓，这样，你才能构建出脉络清晰的知识体系。

要知道，不管多么全面的课程，都不可能覆盖所有问题，更不能直接帮你解决线上的每个 bug 和异常。

回到OpenResty的学习，在我看来，想要学好 OpenResty，你必须理解下面几个重点：

同步非阻塞的编程模式；

lua使用方法

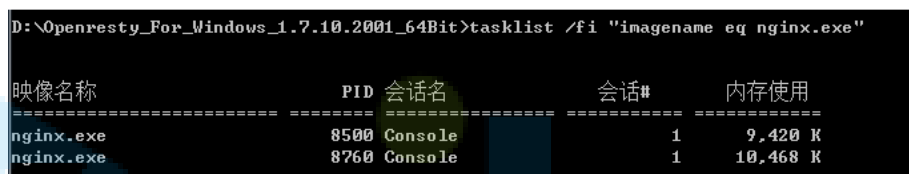
性能优化。

4、openresty安装部署

1) windows安装部署

- 1、下载 Windows 版的 OpenResty 压缩包，这里我下载的是 openresty_for_windows_1.7.10.2001_64bit，你也可以选择 32bit 的版本。如果你对源码感兴趣，下面是源码地址 <https://github.com/LomoX-Offical/nginx-openresty-windows>。
- 2、解压到要安装的目录，这里我选择D盘根目录，你可以根据自己的喜好选择位置。
- 3、进入到 openresty_for_windows_1.7.10.2001_64bit 目录，双击执行 nginx.exe 或者使用命令 start nginx 启动 nginx，如果没有错误现在 nginx 已经开始运行了。
- 4、验证 nginx 是否成功启动：

- 使用 `tasklist /fi "imagename eq nginx.exe"` 命令查看 nginx 进程，其中一个是 master 进程，另一个是 worker 进程，如下图：



映像名称	PID	会话名	会话#	内存使用
nginx.exe	8500	Console	1	9,420 K
nginx.exe	8760	Console	1	10,468 K

- 在浏览器的地址栏输入 localhost，加载 nginx 的欢迎页面。成功加载说明 nginx 正在运行。如下图：



另外当 nginx 成功启动后，master 进程的 pid 存放在 `logs\nginx.pid` 文件中。

PS：OpenResty 当前也发布了 windows 版本，两个版本编译方式还是有区别的，这里更推荐这个版本。

2) linux安装

```
# openresty 下载地址
http://openresty.org
http://openresty.org/cn/download.html

# 安装依赖环境
yum -y install pcre pcre-devel openssl openssl-devel zlib zlib-devel gcc curl
```

```
# 下载openresty,根据最新版本下载即可,版本换为最新版本即可
# https://openresty.org/download/openresty-1.17.8.1.tar.gz
wget https://openresty.org/download/nginx_openresty-1.13.6.1.tar.gz
tar -zxvf ngx_openresty-1.13.6.1.tar.gz
mv openresty-1.13.6.1 openresty
cd openresty
./configure

# 默认会被安装到/usr/local/openresty目录下
# 编译并安装
make && make install
cd /usr/local/openresty

# 启动nginx
/usr/local/openresty/nginx/sbin/nginx
ps -ef | grep nginx
service nginx status
```

5、hello world

使用安装目录下resty来进行字符串的输出;

```
[root@qps001 bin]# ll
total 164
-rwxr-xr-x 1 root root 19185 Jul 27 14:12 md2pod.pl
-rwxr-xr-x 1 root root 15994 Jul 27 14:12 nginx-xml2pod
lrwxrwxrwx 1 root root 37 Jul 27 14:12 openresty -> /usr/local/openresty/nginx/sbin/nginx
-rwxr-xr-x 1 root root 63530 Jul 27 14:12 opm
-rwxr-xr-x 1 root root 35226 Jul 27 14:12 resty
-rwxr-xr-x 1 root root 14957 Jul 27 14:12 restydoc
-rwxr-xr-x 1 root root 8873 Jul 27 14:12 restydoc-index
[root@qps001 bin]# ./resty -e "ngx.say('hello world')"
hello world
[root@qps001 bin]#
```

使用content_by_lua来引入lua (正式的使用方式) :

```
# nginx.conf配置文件中使用时此配置即可
location /lua1 {
    default_type text/html;
    content_by_lua 'ngx.say("hello lua!!");'
}
```

把lua代码从nginx.conf里面抽取出来,保持代码的可读性和可维护性:

```
# 编写lua file 脚本
mkdir -p lua/test1.lua
cat lua/test.lua
ngx.say("hello,world")

# nginx.conf配置实现
location /lua2 {
    default_type text/html;
    content_by_lua_file lua/test1.lua;
}

# 创建test2.lua , 获取请求URI参数
```

```
local args = ngx.req.get_uri_args()
ngx.say("hello openresty! lua is so easy!=="..args.id)

# 请求转发
location /lua3 {
    content_by_lua_file lua/test2.lua;
}

# 创建test3.lua，获取请求URI参数
ngx.exec('/seckill/goods/detail/1');

# 请求转发
location /lua3 {
    content_by_lua_file lua/test3.lua;
}
```

6、安装目录解析

6.1、restydoc

restydoc是OpenResty 提供的文档查看工具，你可以通过它来查看 OpenResty 和 NGINX 的使用文档：

```
$ restydoc -s ngx.say
$ restydoc -s proxy_pass
```

restydoc 这个工具，对服务端工程师的专注开发有很大帮助。

6.2、pod

这里的“pod”，和 k8s 里“pod”的概念完全没有关系。pod 是 Perl 里面的一种标记语言，用于给 Perl 的模块编写文档。

pod 目录中存放的就是OpenResty、NGINX、lua-resty-*、LuaJIT 的文档，这些就和刚才提到的 restydoc 联系在一起了。

十、Openresty实践

1、OpenResty 日志

每个软件都有日志系统，记录软件的运行状态以及出错日志，可以帮助开发者更好的调试程序定位问题。

OpenResty 提供函数 `ngx.log(log_level, ...)` 记录 OpenResty 的运行日志，用法很类似 Lua 的标准库函数 `print`，可以接受任意多个参数，记录任意信息

OpenResty 同时替换了全局函数 `print`，它等价于 `ngx.log(ngx.NOTICE, ...)`

1.1、日志等级

日志是分等级的，特定级别的日志才会真正写入到日志文件，默认是 error级别才写入log文件。

日志等级从高到低依次是：

```
ngx.STDERR    : 日志直接打印到标准输出，最高级别
ngx.EMERG     : 紧急错误
ngx.ALERT     : 严重错误，需要报警给运维系统
ngx.CRIT      : 严重错误
ngx.ERR       : 普通错误
ngx.WARN      : 警告
ngx.NOTICE    : 提醒
ngx.INFO      : 一般信息
ngx.DEBUG     : 调试信息，debug版本才会生效
```

在日常开发中，关键点使用 `INFO` 或 `NOTICE` 级别的日志来调试代码即可。ERR用来做错误异常捕获。

只要改动配置文件中的 `error_log` 设置，就可以开启低等级的日志。

`error.log` 是直接写磁盘的阻塞操作，没有缓冲也没有异步，对性能有很大影响。

正式环境下，一定要将日志级别设置到默认，减少日志写入。

1.2、日志实例

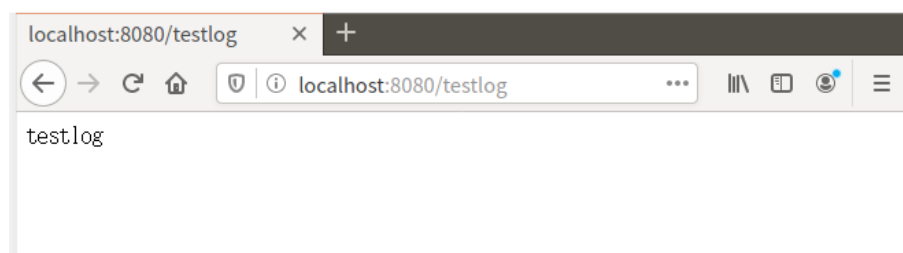
新建lua脚本 `testlog.lua` 代码如下

```
ngx.log(ngx.DEBUG,"debug log")
ngx.log(ngx.INFO,"info log")
ngx.log(ngx.NOTICE,"notice log")
ngx.log(ngx.WARN,"warn log")
ngx.log(ngx.ERR,"error log")
ngx.log(ngx.CRIT,"crit log")
ngx.log(ngx.ALERT,"alert log")
ngx.log(ngx.EMERG,"emerg log")
ngx.log(ngx.STDERR,"stderr log")
ngx.say("testlog")
```

不用重启

访问网址

```
http://localhost:8080/testlog
```



查看日志文件 `logs/error.log`

```

1 2020/05/22 15:36:30 [notice] 88522#88522: signal process started
2 2020/05/22 16:19:03 [notice] 89193#89193: signal process started
3 2020/05/22 16:19:03 [error] 89193#89193: open() "/home/captain/work/logs/nginx.pid" failed (2: No such file or directory)
4 2020/05/22 16:19:32 [notice] 89203#89203: signal process started
5 2020/05/22 16:19:32 [error] 89203#89203: open() "/home/captain/work/logs/nginx.pid" failed (2: No such file or directory)
6 2020/05/22 16:23:58 [notice] 89279#89279: signal process started
7 2020/05/22 16:26:59 [notice] 89305#89305: signal process started
8 2020/05/22 16:32:15 [notice] 89374#89374: signal process started
9 2020/05/22 16:35:37 [notice] 89433#89433: signal process started
10 2020/05/22 16:38:37 [notice] 89540#89540: signal process started
11 2020/05/22 16:39:09 [notice] 89562#89562: signal process started
12 2020/05/22 16:43:25 [notice] 89645#89645: signal process started
13 2020/05/22 16:57:10 [notice] 89809#89809: signal process started
14 2020/05/22 17:20:25 [emerg] 90634#90634: unexpected "}" in /home/captain/work/conf/nginx.conf:12
15 2020/05/22 17:20:56 [notice] 90666#90666: signal process started
16 2020/05/22 17:44:33 [notice] 90969#90969: signal process started
17 2020/05/22 18:08:45 [notice] 91794#91794: signal process started
18 2020/05/22 19:45:35 [notice] 93199#93199: signal process started
19 2020/05/22 19:45:42 [error] 93200#93200: *10 "/home/captain/work/html/index.html" is not found (2: No such file or directory), client: 127.0.0.1, server: , request: GET /index HTTP/1.1, host: "localhost:8080"
20 2020/05/22 20:02:34 [error] 93200#93200: *11 "/home/captain/work/html/index.html" is not found (2: No such file or directory), client: 127.0.0.1, server: , request: GET /index HTTP/1.1, host: "localhost:8080"
21 2020/05/22 20:04:10 [error] 93200#93200: *12 failed to load external Lua file "/home/captain/work/service/http/index.lua": cannot open /home/captain/work/service/h
client: 127.0.0.1, server: , request: "GET /index HTTP/1.1", host: "localhost:8080"
22 2020/05/22 20:32:47 [error] 93200#93200: *13 [lua] testlog.lua:5: error log, client: 127.0.0.1, server: , request: "GET /testlog HTTP/1.1", host: "localhost:8080"
23 2020/05/22 20:32:47 [crit] 93200#93200: *13 [lua] testlog.lua:6: crit log, client: 127.0.0.1, server: , request: "GET /testlog HTTP/1.1", host: "localhost:8080"
24 2020/05/22 20:32:47 [alert] 93200#93200: *13 [lua] testlog.lua:7: alert log, client: 127.0.0.1, server: , request: "GET /testlog HTTP/1.1", host: "localhost:8080"
25 2020/05/22 20:32:47 [emerg] 93200#93200: *13 [lua] testlog.lua:8: emerg log, client: 127.0.0.1, server: , request: "GET /testlog HTTP/1.1", host: "localhost:8080"
26 2020/05/22 20:32:47 [ ] 93200#93200: *13 [lua] testlog.lua:9: stderr log, client: 127.0.0.1, server: , request: "GET /testlog HTTP/1.1", host: "localhost:8080"

```

可以看到，只有 ERR 以及更高级别的日志，输出到了日志文件里。

1.3、开启更多日志

在配置文件中，日志的设置语句格式如下：

```
error_log file level;
```

level就是日志等级，有以下等级可以设置

```
debug|info|notice|warn|error|crit|alert|emerg
```

只有不低于这个等级的日志，才会记录到日志文件中。

下面测试开启所有日志。

将日志等级设置为 debug，debug是最低等级的日志。

```
error_log logs/error.log debug;
```

完整的配置文件内容如下

```

worker_processes 1;
error_log logs/error.log debug;
events {
    worker_connections 1024;
}
http {
    server {
        listen 8080;
        location ~ ^/(w+) {
            default_type text/html;
            content_by_lua_file service/http/$1.lua;
        }
    }
}

```

重启 OpenResty：

```
nginx -p `pwd`/ -s reload
```

再次访问网址

http://localhost:8080/testlog

[查看日志](#)

2、OpenResty 请求处理阶段划分(流水线)

OpenResty把请求处理划分成 `rewrite` `access` `content` 等若干个阶段，每个阶段都可以指定单独的 lua 脚本来做处理。

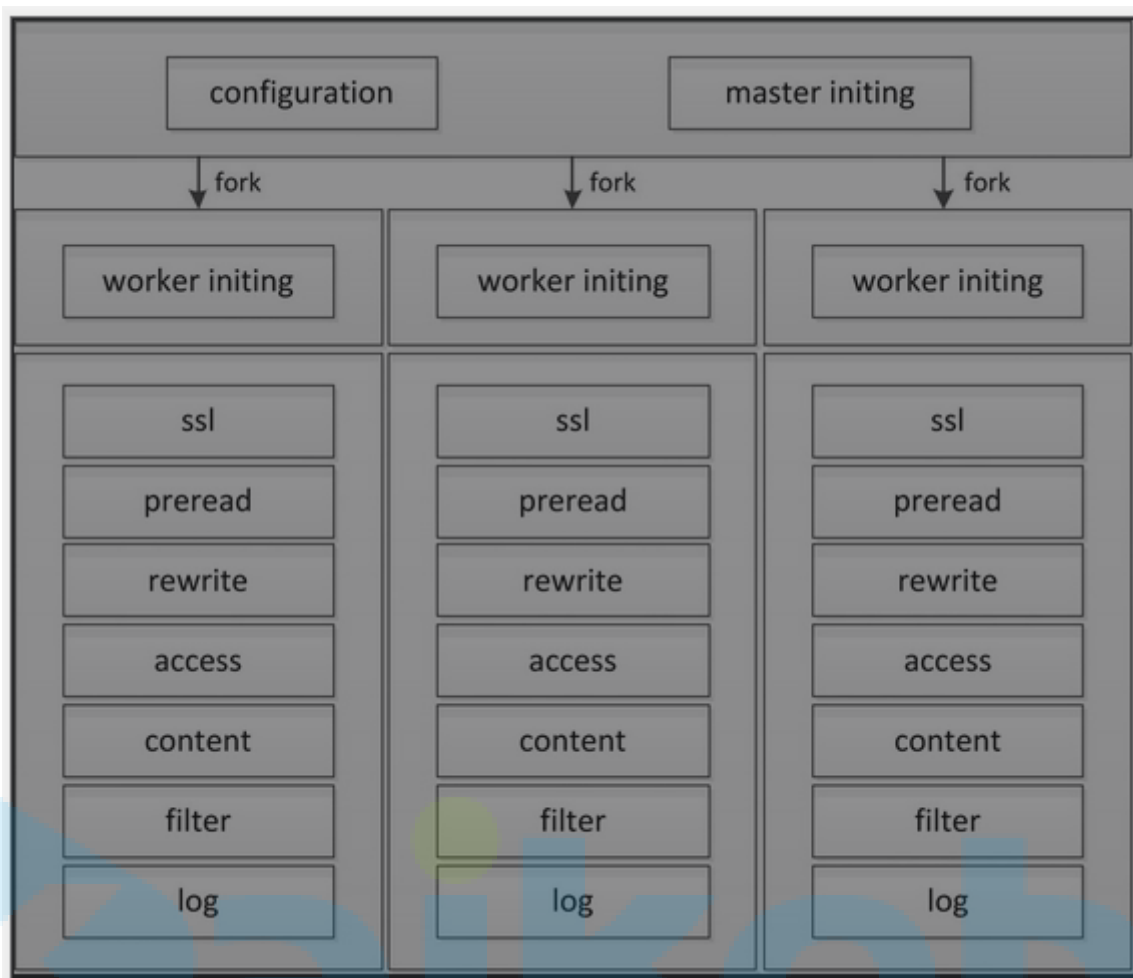
假如我要实现黑名单功能，那么在 `access` 阶段，就可以对客户端的 IP 进行判定,来决定拒绝还是放行，而不用等到 `content` 阶段。

2.1、OpenResty请求处理阶段划分

在收到客户端的请求后，OpenResty对每个请求都会使用一个专门的“流水线”顺序进行处理，“流水线”上就是OpenResty定义的处理阶段，包括：

<code>ssl</code>	SSL/TLS安全通信和验证
<code>prehead</code>	在正式处理之前 预读数据，接受http请求头
<code>rewrite</code>	检查/改写url，实现跳转 重定向
<code>access</code>	访问权限控制
<code>content</code>	产生实际响应内容
<code>filter</code>	对 <code>content</code> 产生的内容进行过滤加工
<code>log</code>	请求处理完毕，记录日志，收尾

这些处理阶段在OpenResty进程里的关系如图



如果只是用来做web服务，那么关注 content 阶段即可。

2.2、OpenResty阶段执行程序

在不同的阶段，可以使用特定的标签，执行lua脚本。常用的有

```
rewrite_by_lua_file --rewrite阶段，检查，改写uri
access_by_lua_file --检查权限，例如ip地址 限制访问次数
content_by_lua_file --主要的逻辑，产生内容，返回给客户端的。
body_filter_by_lua_file --filter阶段，对数据编码 加密 附加额外数据等
log_by_lua_file --可以向后端发送处理完成的回执
```

2.3、实例

下面通过一个实例，来了解OpenResty请求处理阶段划分。

修改配置文件，为每个阶段，指定lua脚本。

```
worker_processes 1;
error_log logs/error.log debug;
events {
    worker_connections 1024;
}
http {
    server {
        listen 8080;
```

```
location ~ ^/(\w+) {  
    default_type text/html;  
    rewrite_by_lua_file service/http/rewrite.lua;  
    access_by_lua_file service/http/access.lua;  
    content_by_lua_file service/http/content.lua;  
    body_filter_by_lua_file service/http/filter.lua;  
    log_by_lua_file service/http/log.lua;  
}  
}
```

分别创建上面的脚本文件，内容都是输出当前阶段的名称。



重启 OpenResty:

```
nginx -p `pwd`/ -s reload
```

访问网址

```
http://localhost:8080
```

查看日志

```

server: , request: "GET /testlog HTTP/1.1", host: "localhost:8080"
83 2020/05/22 20:48:23 [] 94181#94181: *16 [lua] testlog.lua:9: stderr log, client: 127.0.0.1,
server: , request: "GET /testlog HTTP/1.1", host: "localhost:8080"
84 2020/05/22 21:22:40 [emerg] 94880#94880: invalid number of arguments in "rewrite_by_lua_file"
directive in /home/captain/work/conf/nginx.conf:14
85 2020/05/22 21:23:05 [emerg] 94891#94891: invalid number of arguments in "rewrite_by_lua_file"
directive in /home/captain/work/conf/nginx.conf:14
86 2020/05/22 21:23:44 [notice] 94897#94897: signal process started
87 2020/05/22 21:23:44 [notice] 89571#89571: signal 1 (SIGHUP) received from 94897, reconfiguring
88 2020/05/22 21:23:44 [notice] 89571#89571: reconfiguring
89 2020/05/22 21:23:44 [notice] 89571#89571: using the "epoll" event method
90 2020/05/22 21:23:44 [notice] 89571#89571: start worker processes
91 2020/05/22 21:23:44 [notice] 89571#89571: start worker process 94898
92 2020/05/22 21:23:44 [notice] 94181#94181: gracefully shutting down
93 2020/05/22 21:23:44 [notice] 94181#94181: exiting
94 2020/05/22 21:23:44 [notice] 94181#94181: exit
95 2020/05/22 21:23:44 [notice] 89571#89571: signal 17 (SIGCHLD) received from 94181
96 2020/05/22 21:23:44 [notice] 89571#89571: worker process 94181 exited with code 0
97 2020/05/22 21:23:44 [notice] 89571#89571: signal 29 (SIGIO) received
98 2020/05/22 21:23:58 [debug] 94898#94898: *17 [lua] rewrite.lua:1: rewrite
99 2020/05/22 21:23:58 [debug] 94898#94898: *17 [lua] access.lua:1: access
00 2020/05/22 21:23:58 [debug] 94898#94898: *17 [lua] content.lua:1: content
01 2020/05/22 21:23:58 [debug] 94898#94898: *17 [lua] filter.lua:1: filter
02 2020/05/22 21:23:58 [debug] 94898#94898: *17 [lua] log.lua:1: log

```

从日志，就可以更加直观的看出各个阶段的顺序。

3、OpenResty ip黑名单功能

上一篇，了解了 OpenResty 请求处理阶段划分，也为每个阶段都添加了对应的 lua 脚本。
下面针对 `access` 阶段，进行处理，实现 ip 黑名单功能。

修改配置文件如下

```

worker_processes 1;
error_log logs/error.log debug;
events {
    worker_connections 1024;
}
http {
    server {
        listen 8080;
        location ~ ^/(\w+) {
            default_type text/html;
            access_by_lua_file service/http/access.lua;
            content_by_lua_file service/http/$1.lua;
        }
    }
}

```

修改 `access.lua` , 内容如下:

```

ngx.log(ngx.DEBUG, "access")
---获取客户端ip
local sIpAddr = ngx.var.remote_addr
ngx.log(ngx.DEBUG, "clientip:" .. sIpAddr)
---判断是否黑名单
if sIpAddr=="127.0.0.1" then
    return ngx.exit(ngx.HTTP_FORBIDDEN);
end

```

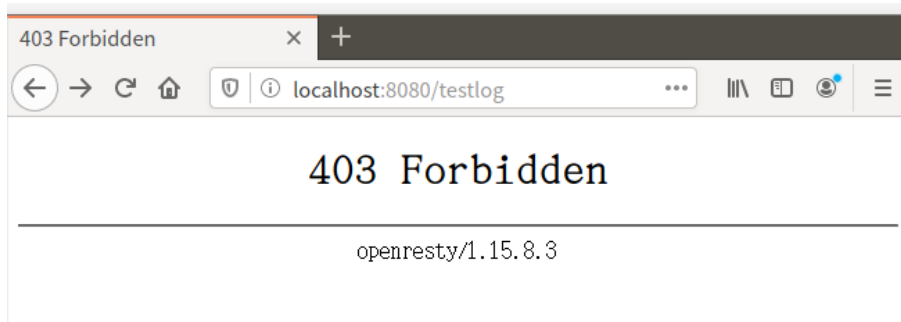
这个代码的功能是判断客户端的ip，如果是 127.0.0.1, 就返回 403 Forbidden。
就是说禁止本机访问。

重启 OpenResty:

```
nginx -p `pwd`/ -s reload
```

在本机打开浏览器，访问网址

```
http://localhost:8080/testlog
```

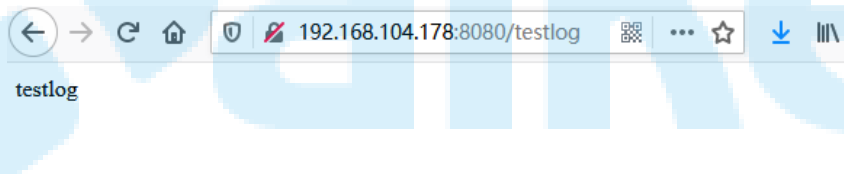


被拦截了。

在另一台电脑上打开网址。

```
http://192.168.104.178:8080/testlog
```

192.168.104.178 就是 OpenResty 的电脑



则可正常访问。

至此，就通过 OpenResty 实现了 ip黑名单功能。

当然，正式应用，需要从mysql/redis 读取黑名单列表。然后还需要一个后台，来更新黑名单。

原理就是这样了。

4、磁盘缓存

nginx一个缓存策略是nginx_proxy_cache策略；

```
# 声明一个缓存,在server配置上面,和upstream同级的位置进行配置
proxy_cache_path /usr/local/openresty/cache/tmp levels=1:2
keys_zone=cache_tmp:100m inactive=7d max_size=10g;
# 说明:
# levels = 1:2 使用2级目录存储缓存,减少寻址消耗,防止一个目录存储缓存文件过多,导致查询缓存效率低下
# keys_zone nginx内存中开辟了100m的空间,存储缓存
# max_size : 文件系统最多存储10g,所有的文件存储超过10g,采用lru淘汰算法

# 在location中增加以下配置
```

```
# 指定proxy_cache_path定义的缓存空间
proxy_cache cache_tmp;
# 指定缓存的key
proxy_cache_key $uri;
# 只有200 206 304 302状态的请求才被缓存, 其他不被缓存
proxy_cache_valid 200 206 304 302 7d;

# nginx 文件级别的缓存, 缓存访问的是本地磁盘文件, 效率比较低, 从磁盘中读取数据, 效率反而变低了....
```

配置proxy_cache如下所示:

```
# 声明一个缓存
proxy_cache_path /usr/local/openresty/cache_tmp levels=1:2 keys_zone=cache_tmp:100m inactive=7d max_size=100m;

server {
    listen      80;
    server_name qps001;

    #charset koi8-r;

    #access_log logs/host.access.log main;

    location /static {
        alias /usr/local/src/nginx-svc/static_vipshop;
        #root html;
        #index index.html index.htm;
    }
    location / {
        #root /usr/local/src/nginx-svc/static_vipshop;
        #root html;
        #index index.html index.htm;
        proxy_pass http://BACKEND;
        proxy_cache cache_tmp;
        proxy_cache_key $uri;
        proxy_cache_valid 200 206 304 302 10d;
        proxy_set_header Host $Host;
        proxy_set_header x-forwarded-for $remote_addr;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

5、内存字典

nginx_proxy_cache缓存每次从磁盘读取文件, 即便使用SSD高性能磁盘进行文件存储, 也改变不了这个缓存是读取文件的效率低下的问题。因此使用nginx+lua方式进行缓存扩展, 使得缓存更高效;

OpenResty 是一个nginx和它的各种三方模块的一个打包而成的软件平台。最重要的一点是它将lua/luajit打包了进来, 使得我们可以使用lua脚本来进行web的开发。有了lua, 我们可以借助于nginx的异步非阻塞的功能, 达到使用lua异步并发访问后端的MySQL, PostgreSQL, Memcached, Redis等等服务。特别是特有的 ngx.location.capture_multi 功能让人印象深刻, 其可以达到极大的减少浏览器的http连接数量, 并且可以异步并发的访问后台Java/PHP/Python等等接口。**OpenResty** 架构的web可以轻松超越Node.js的性能, 并且对后端语言没有限制, 你可以使用Java/PHP/Python等等各种语言。OpenResty(nginx+lua)可以替代node.js的前端渲染的功能。

定义一个缓存: shared dict就是一个共享内存, 所有的worker进程可见, 使用lru淘汰策略。

```
# 在nginx的配置文件 nginx.conf 的 http 端下面加入指令:
# ngx_cache 为缓存的名称, 可以自定义
lua_shared_dict ngx_cache 128m;

# 就定义了一个 名称为 ngx_cache 大小为128m的内存用于缓存, 注意该缓存是所有nginx work process所共享的。
# 在lua脚本中访问缓存:
local ngx_cache = ngx.shared.ngx_cache
local value = ngx_cache:get(key)
local succ, err, forcible = ngx_cache:set(key, value, exptime)
```


下面测试一下，首先在 `nginx.conf` 的 `server` 端中加入：

```
location /cache {
    content_by_lua_file lua/cache.lua;
}
```

一个更简单的例子

```
function get_from_cache(key)
    local cache ngx = ngx.shared.my_cache
    local value = cache_ngx:get(key)
    return value
end
```

```
function set_to_cache(key,value,exptime)
    if not exptime then
        exptime = 0
    end
    local cache_ngx = ngx.shared.my_cache
    local succ,err,forcible = cache_ngx:set(key,value,exptime)
    return succ
end
```

```
local args = ngx.req.get_uri_args()
local id = args["id"]
local item_model = get_from_cache("item_"..id)

if item_model == nil then
    local resp = ngx.location.capture("/item/get?id="..id)
    item_model = resp.body
    ngx.log(ngx.ERR,resp.body)
    set_to_cache("item_"..id,item_model,1*60)
end
ngx.say(item_model)
```

发送请求脚本，这个参数将会被 `lua` 脚本拦截，然后进行请求的转发，获取缓存，或者进行缓存的设置。
`http://qps001/lua1?id=1`

压力测试

这段代码的意思是 写一个 `get` 和 `set` 方法 从 `nginx` 中读取我们配置的 `my_cache` 参数 设置和放入类似 `Map` 的机制

然后下面 获取请求参数 构造 `item_id` 的形式 放入 内存词典中,可以理解为 放入 `map`

然后 有新的请求会先从 内存词典中查找 找不到再传入后面 获取返回值 再放入内存词典中

通过压测我们可以看到

提升量非常大 耗时几乎为0

1) 配置文件 `nginx.conf`

```
lua_shared_dict ngx_cache 128m;
```

定义shared_dict模块, worker进程共享缓存

```
server {
    listen      80;
    server_name qps001;

    #charset koi8-r;

    #access_log logs/host.access.log main;

    location /static {
        alias /usr/local/src/nginx-svc/static_vipshop;
        #root html;
        #index index.html index.htm;
    }
    location /lua1 {
        default_type text/html;
        content_by_lua_file lua/test.lua;
    }

    location /goods/get {
        default_type application/json;
        content_by_lua_file lua/cache.lua;
    }
}
```

定义缓存lua脚本, 把请求代理给lua脚本

location中配置指定了我们最终把请求代理给了lua脚本进行处理。lua脚本中使用了dict共享缓存。

2) 脚本实现

```
[root@qps001 nginx]# cat lua/cache.lua
function set_to_cache(key,value,exptime)
    if not exptime then
        exptime = 0
    end
    local ngx_cache = ngx.shared.ngx_cache
    local succ,err,forcible = ngx_cache:set(key,value,exptime)
    return succ
end

function get_from_cache(key)
    local ngx_cache = ngx.shared.ngx_cache
    local value = ngx_cache:get(key)
    return value
end

local args = ngx.req.get_uri_args()
local id = args.id
local goods = get_from_cache("seckill_goods"..id)

if goods == nil then
    local resp = ngx.location.capture("/seckill/goods/detail/"..id)
    goods = resp.body
    ngx.log(ngx.ERR,resp.body)
    set_to_cache("seckill_goods"..id,goods,60)
end
ngx.say(goods)
```

3) 配置文件如下所示

```
# 设置dict缓存
function set_to_cache(key,value,exptime)
    if not exptime then
        exptime = 0
    end
    local ngx_cache = ngx.shared.ngx_cache
    local succ,err,forcible = ngx_cache:set(key,value,exptime)
    return succ
end

# 获取dict缓存
function get_from_cache(key)
    local ngx_cache = ngx.shared.ngx_cache
```

```

        local value = ngx_cache:get(key)
        return value
    end

    # 缓存逻辑处理
    local args = ngx.req.get_uri_args()
    local id = args.id
    local goods = get_from_cache("seckill_goods_"..id)

    if goods == nil then
        local resp = ngx.location.capture("/seckill/goods/detail/"..id)
        goods = resp.body
        ngx.log(ngx.ERR, resp.body)
        set_to_cache("seckill_goods_"..id, goods, 60)
    end
    ngx.say(goods)

```

6、lua+redis

lua-resty-redis 模块: <https://github.com/openresty/lua-resty-redis> (有文档可以参考)

```

    # 然后编写 cache.lua 脚本:
    # cat cache.lua
    local redis = require "resty.redis"
    local red = redis:new()

    function set_to_cache(key, value, exptime)
        if not exptime then
            exptime = 0
        end
        local ngx_cache = ngx.shared.ngx_cache
        local succ, err, forcible = ngx_cache:set(key, value, exptime)
        return succ
    end

    function get_from_cache(key)
        local ngx_cache = ngx.shared.ngx_cache;
        local value = ngx_cache:get(key)
        if not value then
            value = get_from_redis(key)
            set_to_cache(key, value)
            return value
        end
        ngx.say("get from cache.")
        return value
    end

    function get_from_redis(key)
        red:set_timeout(1000)
        local ok, err = red:connect("127.0.0.1", 6379)
        if not ok then
            ngx.say("failed to connect: ", err)
            return
        end
    end

```

```

        end

        local res, err = red:get(key)
        if not res then
            ngx.say("failed to get doy: ", err)
            return ngx.null
        end

        ngx.say("get from redis.")
        return res
    end

    function set_to_redis(key, value)
        red:set_timeout(1000)
        local ok, err = red:connect("127.0.0.1", 6379)
        if not ok then
            ngx.say("failed to connect: ", err)
            return
        end

        local ok, err = red:set(key, value)
        if not ok then
            ngx.say("failed to set to redis: ", err)
            return
        end
        return ok
    end

    set_to_redis('dog', "Bob")
    local rs = get_from_cache('dog')
    ngx.say(rs)
end

```

测试:

```
[root@localhost ~]# curl localhost/cache
get from redis.
```

Bob

```
[root@localhost ~]# curl localhost/cache
get from cache.
```

Bob

```
[root@localhost ~]# curl localhost/cache
get from cache.
```

Bob

第一次从 redis中获取,以后每次都从cache中获取。

可以使用 ab 测试一下rps(Requests per second):

```
ab -n 1000 -c 100 -k http://127.0.0.1/cache
```

在nginx.conf中加入:

```
location /redis_test{
    content_by_lua_file lua/redis_test.lua;
}
```

redis_test.lua 内容:

```

[root@localhost lua]# cat redis_test.lua
local redis = require "resty.redis"
local red = redis:new()
red:set_timeout(1000)

local ok, err = red:connect("127.0.0.1", 6379)
if not ok then
    ngx.say("failed to connect: ", err)
    return
end

ngx.say("set result: ", ok)

local res, err = red:get("dog")
if not res then
    ngx.say("failed to get dog: ", err)
    return
end

if res == ngx.null then
    ngx.say("dog not found.")
    return
end

ngx.say("dog: ", res)

# 测试访问
[root@localhost lua]# curl localhost/redis_test
set result: 1
dog: an animal

```

```

# 自己的实现，结合redis的方式进行
local redis = require "resty.redis"
local red = redis:new()

function set_to_cache(key,value,exptime)
    if not exptime then
        exptime = 0
    end
    local ngx_cache = ngx.shared.ngx_cache
    local succ,err,forcible = ngx_cache:set(key,value,exptime)
    return succ
end

function get_from_cache(key)
    local ngx_cache = ngx.shared.ngx_cache
    local value = ngx_cache:get(key)
    if not value then
        value = get_from_redis(key)
        set_to_cache(key,value)
        return value
    end
    ngx.say("get from dict cache!")
    return value
end

```

```

function set_to_redis(key,value)
    red:set_timeout(1000)
    local ok,err = red:connect("10.0.3.178",6379);
    if not ok then
        ngx.say("failed to connect:",err)
        return
    end

    local ok,err = red:set(key,value)
    if not ok then
        ngx.say("failed to set to redis",err)
        return
    end
    return ok
end

function get_from_redis(key)
    red:set_timeout(1000)
    local ok,err = red:connect("10.0.3.178",6379);
    if not ok then
        ngx.say("failed to connect:",err)
        return
    end
    local res,err = red:get(key)
    if not res then
        ngx.say("failed get cache:",err)
        return ngx.null
    end
    ngx.say("get from redis...")
    return res
end

local args = ngx.req.get_uri_args()
local id = args.id
local goods = get_from_cache("seckill_goods_"..id)

if goods == nil then
    local resp = ngx.location.capture("/seckill/goods/detail/"..id)
    goods = resp.body
    ngx.log(ngx.ERR,resp.body)
    set_to_cache("seckill_goods_"..id,goods,60)
end
ngx.say(goods)

# 上课最终测试版
local redis = require "resty.redis"
local red = redis:new()

function set_to_cache(key,value,exptime)
    if not exptime then
        exptime = 0
    end
    local ngx_cache = ngx.shared.ngx_cache
    local succ,err,forcible = ngx_cache:set(key,value,exptime)
    set_to_redis(key,value)
    return succ
end

```

```

function get_from_cache(key)
    local ngx_cache = ngx.shared.ngx_cache
    local value = ngx_cache:get(key)
    if not value then
        # 注意此处新建的redv,不能使用上面的value,会冲突
        local redv = get_from_redis(key)
        # 也就是说不能使用 if not value ,不能和上面的if not value一致
        if not redv then
            ngx.say("redis cache not exists")
            return
        end
        set_to_cache(key, redv, 60)
        return value
    end

    return value
end

function set_to_redis(key, value)
    red:set_timeout(1000)
    local ok, err = red:connect("10.0.3.177", 6379)
    if not ok then
        ngx.say("failed to connect:", err)
        return
    end
    local ok, err = red:set(key, value)
    if not ok then
        ngx.say("failed set to redis", err)
        return
    end
    return ok
end

function get_from_redis(key)
    red:set_timeout(1000)
    local ok, err = red:connect("10.0.3.177", 6379)
    if not ok then
        ngx.say("failed to connect:", err)
        return
    end
    local res, err = red:get(key)
    if not res then
        ngx.say("failed get redis cache", err)
        return ngx.null
    end
    ngx.say("get cache from redis.....")
    return res
end

```

```

local args = ngx.req.get_uri_args()
local id = args.id
local goods = get_from_cache("seckill_goods_"..id)

if goods == nil then
    local res = ngx.location.capture("/seckill/goods/detail/"..id)
    goods = res.body
    set_to_cache("seckill_goods_"..id,goods,60)
end
ngx.say(goods)

# 最终版本
local redis = require "resty.redis"
local red = redis:new()

function set_to_cache(key,value,exptime)
    if not exptime then
        exptime = 0
    end
    local ngx_cache = ngx.shared.ngx_cache
    local succ,err,forcible = ngx_cache:set(key,value,exptime)
    set_to_redis(key,value)
    return succ
end

function get_from_cache(key)
    local ngx_cache = ngx.shared.ngx_cache
    local value = ngx_cache:get(key)
    if not value then
        local redv = get_from_redis(key)
        if not redv then
            ngx.say("redis cache not exists")
            return
        end
        set_to_cache(key,redv,60)
        return value
    end

    return value
end

function set_to_redis(key,value)
    red:set_timeout(100000)
    local ok,err = red:connect("10.0.3.177",6379)
    if not ok then
        ngx.say("failed to connect:",err)
        return
    end
    local ok,err = red:set(key,value)
    if not ok then
        ngx.say("failed set to redis",err)
        return
    end
    return ok
end

```



```

function get_from_redis(key)
    red:set_timeout(1000)
    local ok,err = red:connect("10.0.3.177",6379)
    if not ok then
        ngx.say("failed to connect:",err)
        return
    end
    local res,err = red:get(key)
    if not res then
        ngx.say("failed get redis cache",err)
        return ngx.null
    end
    ngx.say("get cache from redis.....")
    return res
end

local args = ngx.req.get_uri_args()
local id = args.id
local goods = get_from_cache("seckill_goods_"..id)

if goods == nil then
    local res = ngx.location.capture("/seckill/goods/detail/"..id)
    goods = res.body
    set_to_cache("seckill_goods_"..id,goods,60)
end
ngx.say(goods)

```

7、注意问题

先重温下 Lua 里的真值与假值：除了 nil 和 false 为假，其他值都是真。“其他值”这个概念包括0、空字符串、空表，等等。

在 Lua 里，通常使用 `and` 和 `or` 作为逻辑操作符。比如 `true and false` 返回 `false`，而 `false or true` 返回 `true`。

OK，复习到此结束，让我们看下这几条规则衍生出来的各种坑。

7.1、第一个坑

在 Lua 代码里，作为给参数设置默认值的惯用法，我们通常能看到 `xx = xx or value` 的语句。如果没有给 xx 入参指定值，它的取值为 nil，该语句就会赋值 value 给它。

这里是今天我们遇到的第一个坑。前面说了，

除了 nil 和 false 为假

除了 nil 还有 false 呢！

如果写代码的时候，把前面设置默认值的语句顺手复制一份；抑或由于业务变动，原来的入参变成布尔类型，也许一下子就掉到这个坑里了。所以这种惯用法，虽然便利了书写，但是也得注意一下，多留点心。

7.2、第二个坑

看到前面的第一个坑，有些小伙伴可能想到一个跳过坑的办法：改成 `xx = (xx == nil) and xx or value`。实际跑下会发现，这个语句也跑不过 `xx` 为 `false` 的 case。这就是第二个坑了。

Lua 没有三元操作符！

Lua 没有三元操作符！

Lua 没有三元操作符！

重要的东西说三遍！虽然你可能在 Lua 代码中见过各种三元操作符的模拟，但是它们都是模拟。既然是模拟，也不过是赝品，只不过有些是以假乱真的高仿品。

`a and b or c` 模式是这些高仿品中的一员。这个模式其实包含两个表达式：先 `a and b` 得到 `x`，再执行 `x or c` 得到最终结果 `y`。在大多数时候，它表现得像是三元操作符。但可惜它不是。

如果 `b` 的值为假，那么 `a and b` 的执行结果恒假；如果 `x` 恒假，则 `x or c` 的执行结果恒为 `c`。所以只要 `b` 的值为假，那么最终结果恒为 `c`。

上面例子里面，`xx` 是一个传进来的变量，所以你只需跑下 case，就能看出这是一个坑。但如果 `b` 的位置上是一个函数的返回值呢？例如 `expr and func1() or func2()` 的形式，如果 `func1` 只是偶尔返回假值，一颗定时炸弹就埋下了。

还是像第一个坑一样的结论，惯用法可以用，但是要多留点心。

如果本文到此结束，它的名字应该是 Lua 中的真值与假值与坑。但实际标题是 OpenResty 中的真值与假值与坑，所以下面讲讲 OpenResty 专属的坑。

7.3、第三个坑

由于 Lua 里面 `nil` 不能作为占位符，为了表示数据空缺，比如 `redis` 键对应值为空，OpenResty 引入了 `ngx.null` 这个常量。`ngx.null` 是一个值为 `NULL` 的 `userdata`。

```
$ resty -e 'print(tostring(ngx.null))'
userdata: NULL
```

`ngx.null` 虽然带了个 `null` 字，但是它并不等于 `nil`。根据开头的规则（其他值都是真），`ngx.null` 的布尔值为真。一个布尔值为真的，表示空的常量，说实话，我还没在其他编程环境中见过。这又是一个潜在的坑。因为在思考的时候，一不小心就会把它当作假值考虑了。举个例子，从 `redis` 获取特定键，如果不存在，调用函数 `A`。如果一时半会想不起 `ngx.null` 的特殊性，可能会直接判断返回值是否为真（或者是否为 `nil`），然后就掉到坑里了。尤其是如果底层逻辑没有把 `ngx.null` 包装好，上层调用的人也许压根没想到除了 `nil` 和 `value` 之外，还有一个 `ngx.null` 的存在！

```
local res, err = redis.get('key1')
if not res then
    ...
end

-- 大部分情况都是好的，直到有一天 key1 不存在..... 500 Internal Server Error!
-- res = res + 1
-- 正确做法
if res ~= ngx.null then
    res = res + 1
```

所以这种时候就需要给在底层跟外部数据服务打交道的代码拦上一道岗，确保妥善处理好 `ngx.null`。至于具体怎么处理，是把 `ngx.null` 转换成 `nil` 呢，还是改成业务相关的默认值，这就看具体的业务逻辑了。

7.4、第四个坑

到目前为止，我们已经见识了 `ngx.null` 这个特立独行的空值了。OpenResty 里还有另外一个空值，来源于 LuaJIT FFI 的 `cdatar.NULL`。正如 `ngx.null` 是 `userdata` 范畴内的 `NULL`，`cdatar.NULL` 是 `cdatar` 范畴内的 `NULL`。当你通过一个 FFI 接口调用 C 函数，而这个函数返回一个 `NULL` 指针，在 Lua 代码看来，它收到的是一个 `cdatar.NULL` 值。你可能会想当然地认为，这时候返回的值应该是 `nil`，因为 Lua 里面的 `nil` 对应的，不正是 C 里面的 `NULL` 嘛。但天不遂人意，这时候返回的却是 `cdatar.NULL`，一个怪胎。

为什么说它是个怪胎呢？因为它跟 `nil` 相等，而 `ngx.null` 就不等于 `nil`。但是，跟 `nil` 相等并不意味着它可替换 `nil`。`cdatar.NULL` 依然服从开头提到的规则——意味着它的布尔值为真。又一个布尔值为真的，表示空的常量！而且这次更诡异了，这个空常量跟 `nil` 是相等的！

各位可以看下示例代码，体会一下：

```
#!/usr/bin/env luajit
local ffi = require "ffi"

local cdata_null = ffi.new("void*", nil)
print(tostring(cdata_null))
if cdata_null == nil then
    print('cdatar.NULL is equal to nil')
end
if cdata_null then
    print('...but it is not nil!')
end
```

怎么处理呢？大多数情况下，只要判断 FFI 调用返回的是不是 `cdatar.NULL` 就够了。我们可以利用 `cdatar.NULL` 跟 `nil` 相等这一点：

```
#!/usr/bin/env luajit
local ffi = require "ffi"

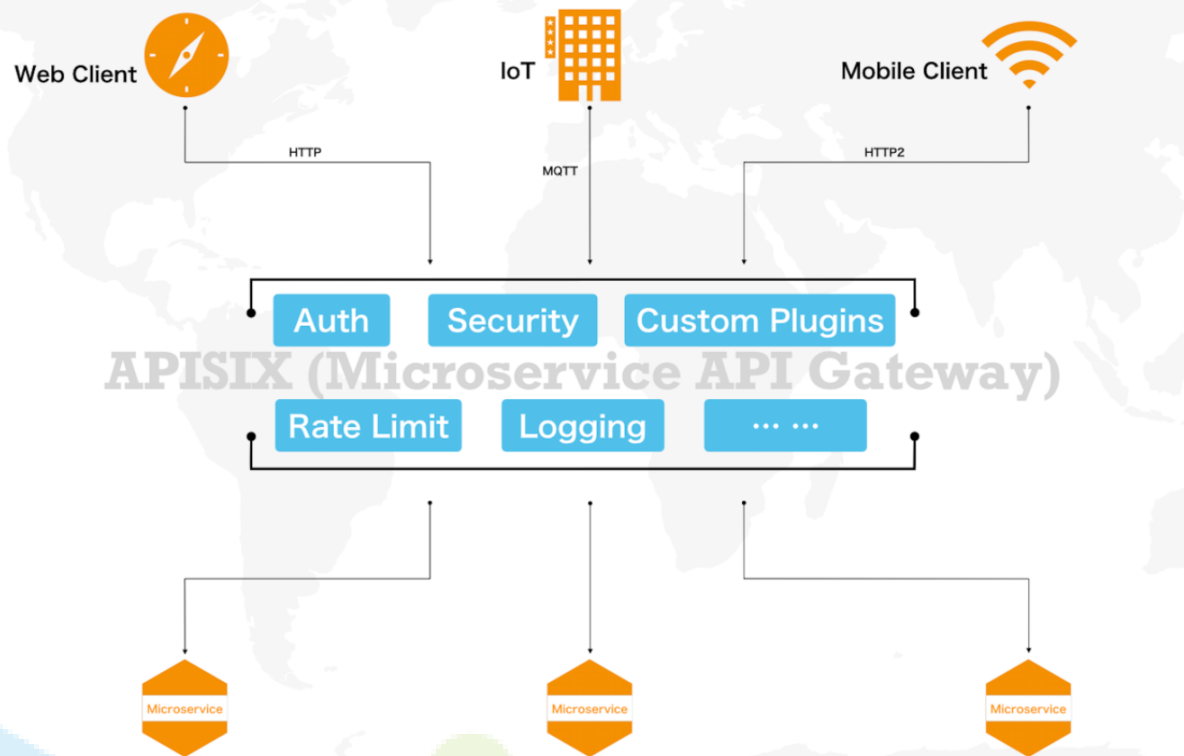
local cdata_null = ffi.new("void*", nil)
if cdata_null == nil then
    print('cdatar.NULL found')
end
```

跟上一个坑一样的，应该在底层跟 C 接口打交道的时候消除掉 `cdatar.NULL`，不要让它扩散出去，“祸害”代码的其他部分。

十一、Openresty网关

1、API 网关有什么用？

让我们先来看下微服务 API 网关的作用。下面这张图，是一个简要的说明：



众所周知，API 网关并非一个新兴的概念，在十几年前就已经存在了，它的作用主要是作为流量的入口，统一处理和业务相关的请求，让请求更加安全、快速和准确地得到处理。它有以下几个传统功能：

- 反向代理和负载均衡，这和 Nginx 的定位和功能是一致的；
- 动态上游、动态 SSL 证书和动态限流限速等运行时的动态功能，这是开源版本 Nginx 并不具备的功能；
- 上游的主动和被动健康检查，以及服务熔断功能；
- 在 API 网关的基础上进行扩展，成为全生命周期的 API 管理平台。

在最近几年，业务相关的流量，不再仅仅由 PC 客户端和浏览器发起，更多的来自手机、IoT 设备等，未来随着 5G 的普及，这些流量会越来越多。同时，随着微服务架构的结构变迁，服务之间的流量也开始爆发性地增长。在这种新的业务场景下，自然也催生了 API 网关更多、更高级的功能：

1. 云原生友好，架构要变得轻巧，便于容器化；
2. 对接 Prometheus、Zipkin、Skywalking 等统计、监控组件；
3. 支持 gRPC 代理，以及 HTTP 到 gRPC 之间的协议转换，把用户的 HTTP 请求转为内部服务的 gRPC 请求；
4. 承担 OpenID Relying Party 的角色，对接 Auth0、Okta 等身份认证提供商的服务，把流量安全作为头等大事来对待；
5. 通过运行时动态执行用户函数的方式来实现 Serverless，让网关的边缘节点更加灵活；
6. 不锁定用户，支持混合云的部署架构；
7. 最后，网关节点要状态无关，可以随意地扩容和缩容。

当一个微服务 API 网关具备了上述十几项功能时，就可以让用户的服务只关心业务本身；而和业务实现无关的功能，比如服务发现、服务熔断、身份认证、限流限速、统计、性能分析等，就可以在独立的网关层面来解决。

从这个角度来看，API 网关既可以替代 Nginx 的所有功能，处理南北向的流量；也可以完成 Istio 控制面和 Envoy 数据面的角色，处理东西向的流量。

2、为什么要新造轮子？

正因为微服务 API 网关的地位如此重要，所以它一直处于兵家必争之地，传统的 IT 巨头在这个领域很早就都有布局。根据 2018 年 Gartner 发布的 API 全生命周期报告，谷歌、CA、IBM、红帽、Salesforce 都是处于领导地位的厂商，开发者更熟悉的 Kong 则处于远见者的区间内。

那么，问题就来了，为什么我们还要新造一个轮子呢？

简单来说，这是因为当前的微服务 API 网关都不足以满足我们的需求。我们首先来看闭源的商业产品，它们的功能都很完善，覆盖了 API 的设计、多语言 SDK、文档、测试和发布等全生命周期管理，并且提供 SaaS 服务，有些还与公有云做了集成，使用起来非常方便。但同时，它们也带来了两个痛点。

第一个痛点，平台锁定问题。API 网关是业务流量的入口，它不像图片、视频等 CDN 加速的这种非业务流量可以随意迁移，API 网关上会绑定不少业务相关的逻辑。你一旦使用了闭源的方案，就很难平滑和低成本地迁移到其他平台。

第二个痛点，无法二次开发的问题。一般的大中型企业都会有自己独特的需求，需要定制开发，但这时候你只能依靠厂商，而不能自己动手去做二次开发。

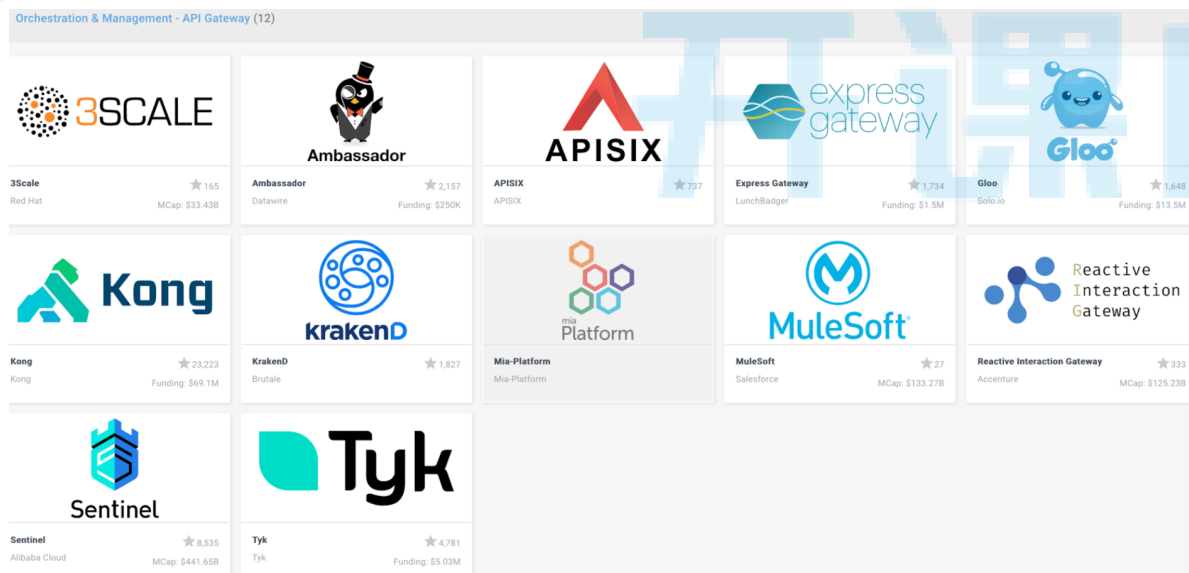
这也是为什么开源的 API 网关方案开始流行的一个原因。不过，现有的开源产品也不是万能的，自身也有很多不足。

第一，依赖 PostgreSQL、MySQL 等关系型数据库。这样，在配置发生变化时，网关节点只能轮询数据库。这不仅造成配置生效慢，也给代码增加了复杂度，让人难以理解；同时，数据库也会成为系统的单点和性能瓶颈，无法保证整体的高可用。如果你把 API 网关用于 K8s 环境下，关系型数据库会显得更加笨重，不利于快速伸缩。

第二，插件不能热加载。当你新增一个插件或者修改现有插件的代码后，必须要重载服务才能生效，这和修改 Nginx 配置后需要重载是一样的，显然会影响用户的请求。

第三，代码结构复杂，难以掌握。有些开源项目做了多层面向对象的封装，一些简单的逻辑也变得雾里看花。但其实，对于 API 网关这种场景，直来直去的表达会更加清晰和高效，也更有利于二次开发。

所以，我们需要一个更轻巧、对云原生和开发友好的 API 网关。当然，我们也不能闭门造车，需要先深入了解已有 API 网关各自的特点，这时候，云原生软件基金会（CNCF）的全景图就是一个很好的参考：



这张图筛选出了业界常见的 API 网关，以开源的方案为主，可以为我们下面的技术选型提供不少有价值的内容。

3、API 网关的核心概念

当然，在具体实现之前，我们还需要了解 API 网关有哪些核心组件。根据我们前面提到的 API 网关具备的功能点，它至少需要下面几个组件才能开始运行。

1)、路由

首先是路由。它通过定义一些规则来匹配客户端的请求，然后根据匹配结果，加载、执行相应的插件，并把请求转发给到指定的上游。这些路由匹配规则可以由 host、uri、请求头等组成，我们熟悉的 Nginx 中的 location，就是路由的一种实现。

2)、插件

其次是插件。这是 API 网关的灵魂所在，身份认证、限流限速、IP 黑白名单、Prometheus、Zipkin 等这些功能，都是通过插件的方式来实现的。既然是插件，那就需要做到即插即用；并且，插件之间不能互相影响，就像我们搭建乐高积木一样，需要用统一规则的、约定好的开发接口，来和底层进行交互。

3)、schema

接着是 schema。既然是处理 API 的网关，那么少不了要对 API 的格式做校验，比如数据类型、允许的字段内容、必须上传的字段等，这时候就需要有一层 schema 来做统一、独立的定义和检查。

4)、存储

最后是存储。它用于存放用户的各种配置，并在有变更时负责推送到所有的网关节点。这是底层非常关键的基础组件，它的选型决定了上层的插件如何编写、系统能否保持高可用和可扩展性等，所以需要我们审慎地决定。

另外，在这些核心组件之上，我们还需要抽象出几个 API 网关的常用概念，它们在不同的 API 网关之间都是通用的。

4、网关技术选型

在明白了微服务 API 网关的核心组件和抽象概念后，我们就要开始技术选型，并动手去实现它了。今天，我们就分别来看下，路由、插件、schema 和存储这四个核心组件的技术选型问题。

4.1、存储

上节课我提到过，存储是底层非常关键的基础组件，它会影响到配置如何同步、集群如何伸缩、高可用如何保证等核心的问题，所以，我们把它放在最开始的位置来选型。

我们先来看看，已有的 API 网关是把数据存储在哪里的。Kong 是把数据储存在 PostgreSQL 或者 Cassandra 中，而同样基于 OpenResty 的 Orange，则是存储在 MySQL 中。不过，这种选择还是有很多缺陷的。

第一，存储需要单独做高可用方案。PostgreSQL、MySQL 数据库虽然有自己的高可用方案，但你还需要 DBA 和机器资源，在发生故障时也很难做到快速切换。

第二，只能轮询数据库来获取配置变更，无法做到推送。这不仅会增加数据库资源的消耗，同时变更的实时性也会大打折扣。

第三，需要自己维护历史版本，并考虑回退和升级。如果用户发布了一个变更，后续可能会有回滚操作，这时候你就需要在代码层面，自己做两个版本之间的 diff，以便配置的回滚。同时，在系统自身升级的时候，还可能会修改数据库的表结构，所以代码层面就需要考虑到新旧版本的兼容和数据升级。

第四，提高了代码的复杂度。在实现网关的功能之外，你还需要为了前面 3 个缺陷，在代码层面去打上补丁，这显然会让代码的可读性降低不少。

第五，增加了部署和运维的难度。部署和维护一个关系型数据库并不是一件简单的事情，如果是一个数据库集群那就更加复杂了，并且我们也无法做到快速扩容和缩容。

针对这样的情况，我们应该如何选择呢？

我们不妨回到 API 网关的原始需求上来，这里存储的都是简单的配置信息，uri、插件参数、上游地址等，并没有涉及到复杂的联表操作，也不需要严格的事务保证。显然，这种情况下使用关系型数据库，可不就是“杀鸡焉用宰牛刀”吗？

事实上，本着最小化够用并且更贴近 K8s 的原则，etcd 就是一个恰到好处的选型了：

- API 网关的配置数据每秒钟的变化次数不会很多，etcd 在性能上是足够的；
- 集群和动态伸缩方面，更是 etcd 天生的优势；
- etcd 还具备 watch 的接口，不用轮询去获取变更。

其实还有一点，可以让我们更加放心地选择 etcd——它已经是 K8s 体系中保存配置的默认选型了，显然已经经过了很多比 API 网关更加复杂的场景的验证。

4.2、路由

路由也是非常重要的技术选型，所有的请求都由路由筛选出需要加载的插件列表，逐个运行后，再转发给指定的上游。不过，考虑到路由规则可能会比较多，所以路由这里的技术选型，我们需要着重从算法的时间复杂度上去考量。

我们先来看下，在 OpenResty 下有哪些现成的路由可以拿来使用。老规矩，让我们在 awesome-resty 的项目中逐个查找一遍，这其中就有专门的 Routing Libraries：

```
lua-resty-route - A URL routing library for OpenResty supporting multiple route
matchers, middleware, and HTTP and WebSockets handlers to mention a few of its
features
  • router.lua - A barebones router for Lua, it matches URLs and executes
    Lua functions
  • lua-resty-r3 - libr3 OpenResty implementation, libr3 is a high-
    performance path dispatching library. It compiles your route paths into a prefix
    tree (trie). By using the constructed prefix trie in the start-up time, you may
    dispatch your routes with efficiency
  • lua-resty-libr3 - High-performance path dispatching library base on
    libr3 for OpenResty
```

你可以看到，这里面包含了四个路由库的实现。前面两个路由都是纯 Lua 实现，相对比较简单，所以有不少功能的欠缺，还不能达到生成的要求。

后面两个库，其实都是基于 libr3 这个 C 库，并使用 FFI 的方式做了一层封装，而 libr3 自身使用的是前缀树。这种算法和存储了多少条规则的数目 N 无关，只和匹配数据的长度 K 有关，所以时间复杂度为 O(K)。

但是，libr3 也是有缺点的，它的匹配规则和我们熟悉的 Nginx location 的规则不同，而且不支持回调。这样，我们就没有办法根据请求头、cookie、Nginx 变量来设置路由的条件，对于 API 网关的场景来说显然不够灵活。

不过，虽说我们尝试从 awesome-resty 中找到可用路由库的努力没有成功，但 libr3 的实现，还是给我们指引了一个新的方向：用 C 来实现前缀树以及 FFI 封装，这样应该可以接近时间复杂度和代码性能上的最优方案。

正好，Redis 的作者开源了一个基数树，也就是压缩前缀树的 C 实现。顺藤摸瓜，我们还可以找到 radix 在 OpenResty 中可用的 FFI 封装库，它的示例代码如下：

```
local radix = require("resty.radixtree")
local rx = radix.new({
```

```

{
    path = "/aa",
    host = "foo.com",
    method = {"GET", "POST"},
    remote_addr = "127.0.0.1",
},
{
    path = "/bb*",
    host = {"*.bar.com", "gloo.com"},
    method = {"GET", "POST", "PUT"},
    remote_addr = "fe80:fe80::/64",
    vars = {"arg_name", "jack"},
}
})

ngx.say(rx:match("/aa", {host = "foo.com",
    method = "GET",
    remote_addr = "127.0.0.1"
}))

```

从中你也可以看出，`lua-resty-radixtree` 支持根据 uri、host、http method、http header、Nginx 变量、IP 地址等多个维度，作为路由查找的条件；同时，基数树的时间复杂度为 $O(K)$ ，性能远比现有 API 网关常用的“遍历 + hash 缓存”的方式，来得更为高效。

4.3、schema

schema 的选择其实要容易得多，我们在前面介绍过的 `lua-rapidjson`，就是非常好的一个选择。这部分你完全没有必要自己去写一个，json schema 已经足够强大了。下面就是一个简单的示例：

```

local schema = {
    type = "object",
    properties = {
        count = {type = "integer", minimum = 0},
        time_window = {type = "integer", minimum = 0},
        key = {type = "string", enum = {"remote_addr", "server_addr"}},
        rejected_code = {type = "integer", minimum = 200, maximum = 600},
    },
    additionalProperties = false,
    required = {"count", "time_window", "key", "rejected_code"},
}

```

4.4、插件

有了上面存储、路由和 schema 的基础，上层的插件应该如何实现，其实就清晰多了。插件并没有现成的开源库可以使用，需要我们自己来实现。插件在设计的时候，主要有三个方面需要我们考虑清楚。

首先是如何挂载。我们希望插件可以挂载到 `rewrite`、`access`、`header filter`、`body filter` 和 `log` 阶段，甚至在 `balancer` 阶段也可以设置自己的负载均衡算法。所以，我们应该在 Nginx 的配置文件中暴露这些阶段，并在对插件的实现中预留好接口。

其次是如何获取配置的变更。由于没有关系型数据库的束缚，插件参数的变更可以通过 etcd 的 watch 来实现，这会让整体框架的代码逻辑变得更加明了易懂。

最后是插件的优先级。具体来说，比如，身份认证和限流限速的插件，应该先执行哪一个呢？绑定在 route 和绑定在 service 上的插件发生冲突时，又应该以哪一个为准呢？这些都是我们需要考虑到位的。

在梳理清楚插件的这三个问题后，我们就可以得到插件内部的一个流程图了：

Nginx 本身在处理一个用户请求时，会按照不同的阶段进行处理，总共会分为 11 个阶段。而 openresty 的执行指令，就是在这 11 个步骤中挂载 lua 执行脚本实现扩展，我们分别看看每个指令的作用。

initbylua：当 Nginx master 进程加载 nginx 配置文件时会运行这段 lua 脚本，一般用来注册全局变量或者预加载 lua 模块

initwokerby_lua：每个 Nginx worker 进程启动时会执行的 lua 脚本，可以用来做健康检查

setbylua：设置一个变量

rewritebylua：在 rewrite 阶段执行，为每个请求执行指定的 lua 脚本

accessbylua：为每个请求在访问阶段调用 lua 脚本

contentbylua：前面演示过，通过 lua 脚本生成 content 输出给 http 响应

balancerbylua：实现动态负载均衡，如果不是走 contentbylua，则走 proxy_pass，再通过 upstream 进行转发

headerfilterby_lua：通过 lua 来设置 headers 或者 cookie

bodyfilterby_lua：对响应数据进行过滤

logbylua：在 log 阶段执行的脚本，一般用来做数据统计，将请求数据传输到后端进行分析

5、网关灰度发布

实现网关的灰度发布功能

灰度发布就是新版本刚上线时，可以只对某一部分用户开放，比如 90% 的客户还是只能访问到旧版本，另外 10% 的用户被添加到灰度名单中可以访问最新版本。

openresty 根目录下创建 gray 目录 gray 目录下创建 conf logs lua 三个目录 如下：

```
[root@zk03 gray]# vi conf/nginx.conf
worker_processes 1;
error_log logs/error.log;
events{
    worker_connections 1024;
}
http{
    lua_package_path "$prefix/lualib/?.lua;";
    lua_package_cpath "$prefix/lualib/?.so;";
    upstream prod {
        server zk03:8080;
    }
    upstream pre {
        server zk03:8081;
    }
    server {
        listen 80;
        server_name localhost;
        location /api {
            content_by_lua_file lua/gray.lua;
        }
        location @prod {
            proxy_pass http://prod;
        }
        location @pre {
```

```

        proxy_pass http://pre;
    }
}
server {
    listen 8080;
    location / {
        content_by_lua_block {
            ngx.say("I'm prod env");
        }
    }
}

server {
    listen 8081;
    location / {
        content_by_lua_block {
            ngx.say("I'm pre env");
        }
    }
}
}

```

在gray/lua 目录下新增 gray.lua文件如下 实现逻辑：在名单中的ip 访问 pre环境 否则访问prod环境：

```

[root@zk03 gray]# vi lua/gray.lua
local redis=require "resty.redis";
local red=redis:new();
red:set_timeout("1000");
local ok,err=red:connect("127.0.0.1",6379);
if not ok then
    ngx.say("failed to connect redis",err);
    return;
end
local ip=ngx.var.remote_addr;
local ip_lists=red:get("gray");
if string.find(ip_lists,ip) == nil then
    ngx.exec("@prod");
else
    ngx.exec("@pre");
end
local ok,err=red:close();

```

启动openresty

```

[root@zk03 gray]# ../nginx/sbin/nginx -s reload -p /usr/local/openresty/gray
1

```

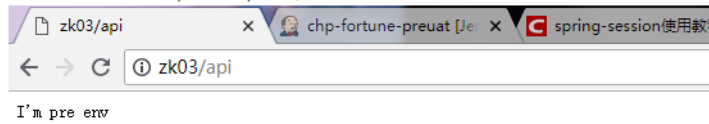
连接redis 将 gray 设置为 192.168.68.2 即将该ip加入到名单中 如下：

```

[root@zk03 src]# ./redis-cli
127.0.0.1:6379> set gray 192.168.68.2
OK
123

```

浏览器访问 zk03/api 返回 pre 环境



修改 redis 中的 gray 为 192.168.68.1 如下:

```
127.0.0.1:6379> set gray 192.168.68.1
OK
12
```

浏览器访问 zk03/api 返回 prod 环境 如下:

