OpenResty最佳实践

看云文档小组

目　录

- 序
  - Lua简介
  - Lua环境搭建
  - 基础数据类型
  - 表达式
  - 控制结构
    - if/else
    - while
    - repeat
  - 控制结构for的使用
    - break，return
  - Lua函数
    - 函数的定义
    - 函数的参数
    - 函数的返回值
    - 函数回调
  - 模块
  - String库
  - Table库
  - 日期时间函数
  - 数学库函数
  - 文件操作
  - 元表
  - 面向对象编程
  - FFI
- LuaRestyRedisLibrary
  - select+set_keepalive组合操作引起的数据读写错误
  - redis接口的二次封装（简化建连、拆连等细节）
  - redis接口的二次封装（发布订阅）
  - pipeline压缩请求数量
  - script压缩复杂请求
- LuaCjsonLibrary
  - json解析的异常捕获
  - 稀疏数组

空table编码为array还是object
跨平台的库选择
PostgresNginxModule
调用方式简介
不支持事务
超时
健康监测
SQL注入
LuaNginxModule
执行阶段概念
正确的记录日志
热装载代码
阻塞操作
缓存
sleep
定时任务
禁止某些终端访问
请求返回后继续执行
调试
调用其他C函数动态库
我的lua代码需要调优么
变量的共享范围
动态限速
shared.dict 非队列性质
如何添加自己的lua api
正确使用长链接
如何引用第三方resty库
使用动态DNS来完成HTTP请求
缓存失效风暴
Lua
下标从1开始
局部变量
判断数组大小
非空判断
正则表达式
不用标准库

        虚变量
        函数在调用代码前定义
        抵制使用module()函数来定义Lua模块
        点号与冒号操作符的区别
    测试
        单元测试
        API测试
        性能测试
        持续集成
        灰度发布
    web服务
        API的设计
        数据合法性检测
        协议无痛升级
        代码规范
        连接池
        c10k编程
        TIME_WAIT问题
        与Docker使用的网络瓶颈
    火焰图
        什么时候使用
        显示的是什么
        如何安装火焰图生成工具
        如何定位问题

序

OpenResty最佳实践

在2012年的时候，我加入到奇虎360公司，为新的产品做技术选型。由于之前一直混迹在python圈子里面，也接触过nginx c模块的高性能开发，一直想找到一个兼备python快速开发和nginx c模块高性能的产品。看到OpenResty后，有发现新大陆的感觉。

于是我在新产品里面力推OpenResty，团队里面几乎没有人支持，经过几轮性能测试，虽然轻松击败所有的其他方案，但是其他开发人员并不愿意参与到基于OpenResty这个“陌生”框架的开发中来。于是我一个人开始了OpenResty之旅，刚开始经历了各种技术挑战，庆幸有详细的文档，以及春哥和邮件列表里面热情的帮助，我成了团队里面bug最少和几乎不用加班的同学。

2014年，团队进来了一批新鲜血液，他们都很有技术品味，先后都选择OpenResty来作为技术方向。我不再是一个人在战斗，而另外一个新问题摆在团队面前，如何保证大家都能写出高质量的代码，都能对OpenResty有深入的了解？知识的沉淀和升华，成为一个迫在眉睫的问题。

我们选择把这几年的一些浅薄甚至可能是错误的实践，通过gitbook的方式公开出来，一方面有利于团队自身的技术积累，另一方面，也能让更多的高手一起加入，让OpenResty的使用变得更加简单，更多的应用到服务端开发中，毕竟人生苦短，少一些加班，多一些陪家人。

这本书的定位是最佳实践，同时会对OpenResty做简单的基础介绍。但是我们对初学者的建议是，在看书的同时下载并安装OpenResty，把官方网站的Presentations浏览和实践几遍。

希望你能enjoy OpenResty之旅！

点我看书

本书源码在 Github 上维护，欢迎参与：我要写书。也可以加入QQ群来和我们交流：

Lua简介

Lua简介

Lua是什么

Lua是一个可扩展的轻量级脚本语言，它是用C语言编写的。Lua的设计目是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。Lua代码简洁优美，几乎在所有操作系统和平台上都可以编译、运行。

> 一个完整的Lua解释器不过200k

Lua和Luajit的区别

LuaJIT是采用C语言写的Lua的解释器。LuaJIT被设计成全兼容标准Lua 5.1, 因此LuaJIT代码的语法和标准Lua的语法没多大区别。LuaJIT和Lua的一个区别是，LuaJIT的运行速度比标准Lua快数十倍，可以说是一个lua的高效率版本。

若无特殊说明，我们接下来的章节都是基于LuaJIT进行介绍的。

> Lua官网链接

Lua环境搭建

搭建Lua环境

在Windows上搭建环境

下载安装包

前往官网下载LuaJIT源码压缩包，网址：http://luajit.org/download.html。本书以LuaJIT-2.0.4为例。

安装Lua

使用visual studio编译刚才下载好的源码包：开始 –> 程序 –> Microsoft Visual Studio xx –> Visual Studio Tools –> Visual Studio 命令提示。

然后切换至LuaJIT的src目录，运行msvcbuild.bat

将生成的luajit.exe、lua51.dll、jit 复制到打包工具的相对目录下，这样在工具中就可以直接调用luajit –b source_file out_file (一般都是lua后缀，代码不用改动)

如果你windows系统中没有安装visual studio或者不想手工编译，可以直接在网上搜索下载已经被别人编译好的LuaJIT。

在Linux上搭建环境

本书以Ubuntu为例来说明。首先，使用apt-cache命令查看有哪些版本的luajit可以安装。

安装Luajit

在ubuntun系统上，使用apt-get insall命令安装luajit：

```
sudo apt-get install luajit
```

验证Luajit是否安装成功

输入luajit -v查看luajit版本，如果返回以下类似内容，则说明安装成功：

```
LuaJIT 2.0.3 -- Copyright (C) 2005-2014 Mike Pall. http://luajit.org/
```

如果想了解其他系统安装LuaJIT的步骤，或者安装过程中遇到问题，可以到LuaJIT官网查看：http://luajit.org/install.html

选择一个好用的代码编辑器：Sublime Text
一个好的代码编辑器可以让我们编写代码时更加顺手，虽然Lua有相应的IDE编辑环境，但是我们建议您使用Sublime作为您的代码编辑器。Sublime文本编辑器有很多可选的插件包，可以帮助我们的代码编写。本书的代码都是在Sublime上进行编辑的。

Hello World程序

安装好LuaJIT后，我们来运行我们的第一个程序：HelloWorld.lua。

> 代码

```
function main()
  print("Hello World")
end

main()
```

到HelloWorld.lua所在目录下，运行`luajit ./HelloWorld.lua`运行这个HelloWorld.lua程序，输出如下结果：

```
Hello World
```

基础数据类型

Lua基础数据类型

nil

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
```

boolean

布尔类型，可选值true/false；Lua中nil和false为“假”，其它所有值均为“真”。

```
local a = true
local b = 0
local c = nil
if a then
    print("a")
else
    print("not a")
end

if b then
    print("b")
else
    print("not b")
end

if c then
    print("c")
else
    print("not c")
end

------output:
a
b
not c
```

number

数字，包括整数与浮点数

```
local order = 3
local score = 98.5
```

string

字符串

```
local website = "www.google.com"
```

table

表，关联数组，索引可为字符串string或(整)数number类型

```
local corp = {
    web = "www.example.com",
    telephone = "12345678",
    staff = {"Jack", "Scott", "Gary"},
    100876,
    100191,
    ["City"] = "Beijing"
}

print(corp.web)               -->output:www.google.com
local key = "telephone"
print(corp[key])              -->output:12345678
print(corp[2])                -->output:100191
print(corp["City"])           -->output:"Beijing"
print(corp.staff[1])          -->output:Jack
```

function

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

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

表达式

表达式

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。 整数相除的结果是向下取整
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
```

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
```

逻辑运算符

逻辑运算符	说明
and	逻辑与
or	逻辑或
not	逻辑非

> 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

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。

控制结构

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。

if/else

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。

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

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

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 equa
l 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elseif是连在一起的，若不else与if写成"else if"则相当于在else 里嵌套,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 equa
l 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

while

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,但是提供了功力相当的repeat。while型控制结构语法如下：

```
x = 1
sum = 0
--求1到5的各数相加和
while x <=5 do
    sum = sum + x
    x = x + 1
end
print(sum)
```

> 运行输出：15

repeat

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。

控制结构for的使用

控制结构：for

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。

数字型for
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 = exp1, exp2, exp3 do`

var从exp1变化到exp2，每次变化都以exp3作为步长（step）递增var，并执行一次“执行体”。第三个表达式exp3是可选的，若不指定的话，Lua会将步长默认为1。

> 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
```

泛型for
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", "Satu
rday"
}
```

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","Su
nday"
}
```

```
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"、...)。

break，return

break,return 关键字

break

break用来终止while,repeat,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
    i = i + 1;
    if sum > 100 then
        break
    end
end
print("The result is "..i)
```

> 运行结果如下：The result is 15

在实际应用中，break经常用于嵌套循环中。

return

return主要用于从函数中返回一个或多个值，相关的细节可以参考函数的返回值章节。
return只能写在最后的语句块，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：

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

```
sum = add(10, 20)
print("The sum is "..sum)
answer = is_positive(-10);
print(answer)
```

> 运行结果如下：

> The sum is 30

> -10 is non-positive

Lua函数

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 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。

函数的定义

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
```

函数的定义一定要放在函数调用前。

> 示例代码：

```
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：

```
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、要勤于写注释，注释可以帮助读者理解代码。

函数的参数

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：

```
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：

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

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
```

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：

```
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
```

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：

```
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
```

按址传递

当函数参数是table类型时，传递进来的是table在内存中的地址，这时在函数内部对table所做的修改，不需要使用return返回，就是有效的。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。

函数的返回值

函数的返回值

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

```
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，且函数调用不是一系列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：

```
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
```

函数回调

自定义函数

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 {}
methodName(unpack(args, 1, table.maxn(args)))
```

使用场景

你要调用的函数名是未知的

函数参数类型和数目也是未知的

一般常用于在定时器处理逻辑之中

> 伪代码

```
addTask(endTime, callback, params)

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

小试牛刀

```
local function run(x, y)
    ngx.say('run', x, y)
end

local function attack(targetId)
    ngx.say('targetId', targetId)
end

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

doAction(run, 1, 2)
doAction(attack, 1111)
```

我们再新建一个模块 sample

```
local _M = {}

function _M:hello(str)
    ngx.say('hello', str)
end

function _M.world(str)
    ngx.say('world', str)
end

return _M
```

这个时候我们可以这样调用，代码接上文

因为sample模块的方法声明方式的不同

所以在调用时有些区别 主要是.和:的区别

https://github.com/humbut/openresty-best-practices/blob/master/lua/dot_diff.md

```
local sample = require "sample"
doAction(sample.hello, sample, ' 123')  -- 相当于sample:hello('123')
doAction(sample.world, ' 321') -- 相当于sample.world('321')
```

实战演练

以下代码为360公司公共组件之缓存模块，正是利用了部分特性

```
-- {
--   key="...",           cache key
--   exp_time=0,          default expire time
--   exp_time_fail=3,     success expire time
--   exp_time_succ=60*30, failed  expire time
--   lock={...}           lock opsts(resty.lock)
-- }
function get_data_with_cache( opts, fun, ... )
  local ngx_dict_name = "cache_ngx"

  -- get from cache
  local cache_ngx = ngx.shared[ngx_dict_name]
  local values = cache_ngx:get(opts.key)
  if values then
    values = json_decode(values)
    return values.res, values.err
  end
```

```
    -- cache miss!
    local lock = lock:new(ngx_dict_name, opts.lock)
    local elapsed, err = lock:lock("lock_" .. opts.key)
    if not elapsed then
      return nil, string.format("get data with cache not found and sleep(%s
s) not found again", opts.lock_wait_time)
    end

    -- someone might have already put the value into the cache
    -- so we check it here again:
    values = cache_ngx:get(opts.key)
    if values then
      lock:unlock()

      values = json_decode(values)
      return values.res, values.err
    end

    -- get data
    local exp_time = opts.exp_time or 0 -- default 0s mean forever
    local res, err = fun(...)
    if err then
      exp_time = opts.exp_time_fail or exp_time
    else
      exp_time = opts.exp_time_succ or exp_time
    end

    --  update the shm cache with the newly fetched value
    cache_ngx:set(opts.key, json_encode({res=res, err=err}), exp_time)
    lock:unlock()
    return res, err
end
```

模块

模块

从Lua5.1版本开始，就对模块和包添加了新的支持，可是使用require和module来定义和使用模块和包。require用于使用模块，module用于创建模块。简单的说，一个模块就是一个程序库，可以通过require来加载。然后便得到了一个全局变量，表示一个table。这个table就像是一个命名空间，其内容就是模块中导出的所有东西，比如函数和常量，一个符合规范的模块还应使require返回这个table。

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。下面写一个实现复数加法和减法的模块。

> 把下面的代码保存在文件complex.lua中。

```
local _M = {}   -- 局部变量，模块名称

function _M.new(r, i)
   return {r = r, i = i}
end

 _M.i =  _M.new(0, 1)     -- 定义一个table型常量i

function  _M.add(c1, c2)  --复数加法
    return  _M.new(c1.r + c2.r, c1.i + c2.i)
end

function  _M.sub(c1, c2)  --复数减法
    return  _M.new(c1.r - c2.r, c1.i - c2.i)
end

return  _M  -- 返回模块的table
```

> 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 complex = require "complex"

local com1 = complex.new(0, 1)
local com2 = complex.new(1, 2)

local ans = complex.add(com1, com2)
print(ans.r, ans.i)      -->output  1     3
```

下面定义和使用模块的习惯是极不好的。用这种定义模块是非常危险的，因为引入了全局变量，在引用模块时非常容易覆盖外面的变量。为了引起读者的注意，每行代码前加了一个'?'。

> 把下面的代码保存在文件complex.lua中。(执行代码要去掉前面的'?')

```
? complex = {}   -- 全局变量，模块名称

? function complex.new(r, i)
?     return {r = r, i = i}
? end

? complex.i =complex.new(0, 1)  -- 定义一个table型常量i

? function complex.add(c1, c2)  --复数加法
?     return M.new(c1.r + c2.r, c1.i + c2.i)
? end

? function complex.sub(c1, c2)  --复数减法
?     return M.new(c1.r - c2.r, c1.i - c2.i)
? end

? return complex  -- 返回模块的table
```

> 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。(执行代码要去掉前面的'?')

```
? require "complex"

? local com1 = complex.new(0, 1)
? local com2 = complex.new(1, 2)

? local ans = complex.add(com1, com2)
? print(ans.r, ans.i)      -->output  1     3
```

module函数（不推荐使用）
在Lua5.1中提供了一个新函数module。module(..., package.seeall) 这一行代码会创建一个新的table，将其赋予给模块名对应的全局字段和loaded table，还会将这个table设为主程序块的环境，并且模块还能提供外部访问。但这种写法是不提倡的，官方给出了两点原因：

1. package.seeall 这种方式破坏了模块的高内聚，原本引入"filename"模块只想调用它的内部函数，但是它却可以读写全局属性，例如 "filename.os"。
2. module 函数压栈操作引发的副作用，污染了全局环境变量。例如 module("filename") 会创建一个 filename 的 table*，并将这个 table 注入全局环境变量中，这样使得没有引用它的文件也能调用 filename 模块的方法。

> 把mod1.lua文件改成如下代码。

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

function new(r, i)
    return {r = r, i = i}
end

local i = new(0, 1)

function add(c1, c2)
    return new(c1.r + c2.r, c1.i + c2.i)
end

function sub(c1, c2)
    return new(c1.r - c2.r, c1.i - c2.i)
end

function update( )
   A = A + 1
end

getmetatable(mod1).__newindex = function (table, key, val) --防止模块更改全局变量
    error('attempt to write to undeclared variable "' .. key .. '": ' .. debug.traceback())
end
```

> 把main.lua文件改成如下代码。

```
A = 2
require "mod1"
```

```
local com1 = mod1.new(0, 1)
local com2 = mod1.new(1, 2)

mod1.update()

local ans = mod1.add(com1, com2)
print(ans.r, ans.i)
```

运行main.lua，会报错：

```
lua: .\mod1.lua:22: attempt to write to undeclared variable "A": stack tr
aceback:
    .\mod1.lua:22: in function <.\mod1.lua:21>
    .\mod1.lua:18: in function 'update'
    my.lua:9: in main chunk
    [C]: ?
stack traceback:
    [C]: in function 'error'
    .\mod1.lua:22: in function <.\mod1.lua:21>
    .\mod1.lua:18: in function 'update'
    my.lua:9: in main chunk
    [C]: ?
```

> 把mod1.lua文件改成如下代码。

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

function new(r, i)
    return {r = r, i = i}
end

local i = new(0, 1)

function add(c1, c2)
    return new(c1.r + c2.r, c1.i + c2.i)
end

function sub(c1, c2)
    return new(c1.r - c2.r, c1.i - c2.i)
end

function update( )
   A = A + 1
end
```

运行main.lua，得到结果：

```
1       3
```

String库

String library

lua字符串库包含很多强大的字符操作函数。字符串库中的所有函数都导出在模块string中。在lua5.1中，它还将这些函数导出作为string类型的方法。这样假设要返回一个字符串转的大写形式，可以写成ans = string.upper(s),也能写成 ans = s:upper()。为了避免与之前版本不兼容，此处使用前者。

string.byte(s [, i [, j]])
返回字符s[i]、s[i + 1]、s[i + 2]、…、s[j]所对应的ASCII码。i的默认值为1；j的默认值为i。

> 示例代码

```
print(string.byte("abc", 1, 3))
print(string.byte("abc", 3)) --缺少第三个参数，第三个参数默认与第二个相同，此时为3
print(string.byte("abc")) --缺少第二个和第三个参数，此时这两个参数都默认为 1

-->output
97    98    99
99
97
```

string.char (…)
接收0个或更多的整数（整数范围 ：0~255 ）；返回这些整数所对应的ASCII码字符组成的字符串。当参数为空时，默认是一个 0。

> 示例代码

```
print(string.char(96,97,98))
print(string.char()) --参数为空，默认是一个0，你可以用string.byte(string.char())测试一下
print(string.char(65,66))

-->output
`ab

AB
```

string.upper(s)
接收一个字符串s，返回一个把所有大写字母变成小写字母的字符串。

> 示例代码

```
print(string.upper("Hello Lua"))  -->output  HELLO LUA
```

string.lower(s)
接收一个字符串s，返回一个把所有大写字母变成小写字母的字符串。

> 示例代码

```
print(string.lower("Hello Lua"))  -->output   hello lua
```

string.len(s)
接收一个字符串，返回它的长度。

> 示例代码

```
print(string.len("hello lua")) -->output  9
```

string.find(s, p [, init [, plain]])
在s字符串中第一次匹配 p字符串。若匹配成功，则返回p字符串在s字符串中出现的开始位置和结束位置；若匹配失败，则返回nil。第三个参数init默认为1，并且可以为负整数，当init为负数时，表示从s字符串的string.len(s) + init 索引处开始向后匹配字符串p。第四个参数默认为false，当其为true时，只会把p看成一个字符串对待。

> 示例代码

```
print(string.find("abc cba","ab"))
print(string.find("abc cba","ab",2))
--从索引为2的位置开始匹配字符串：ab
print(string.find("abc cba","ba",-1))   --从索引为7的位置开始匹配字符串：ba
print(string.find("abc cba","ba",-3))   --从索引为6的位置开始匹配字符串：ba
```

```
print(string.find("abc cba", "(%a+)",1))--从索引为1处匹配最长连续且只含字母的字
符串
print(string.find("abc cba", "(%a+)",1,true)) --从索引为1的位置开始匹配字符串
：(%a+)

-->output
1    2
nil
nil
6    7
1    3    abc
nil
```

string.format(formatstring, ···)

按照格式化参数formatstring，返回后面···内容的格式化版本。编写格式化字符串的规则与标准c语言中printf函数的规则基本相同：它由常规文本和指示组成，这些指示控制了每个参数应放到格式化结果的什么位置，及如何放入它们。一个指示由字符'%'加上一个字母组成，这些字母指定了如何格式化参数，例如'd'用于十进制数、'x'用于十六进制数、'o'用于八进制数、'f'用于浮点数、's'用于字符串等。在字符'%'和字母之间可以再指定一些其他选项，用于控制格式的细节。

> 示例代码

```
print(string.format("%.4f", 3.1415926))   --保留4位小数
print(string.format("%d %x %o", 31, 31, 31))--十进制数31转换成不同进制
d = 29; m = 7; y = 2015                    --一行包含几个语句，用；分开
print(string.format("%s %02d/%02d/%d", "today is:", d, m, y))

-->output
3.1416
31 1f 37
today is: 29/07/2015
```

string.match(s, p [, init])

在字符串s中匹配字符串p，若匹配成功，则返回目标字符串中与模式匹配的子串；否则返回nil。第三个参数init默认为1，并且可以为负整数，当init为负数时，表示从s字符串的string.len(s) + init 索引处开始向后匹配字符串p。

> 示例代码

```
print(string.match("hello lua", "lua"))
print(string.match("lua lua", "lua", 2))  --匹配后面那个lua
```

```
print(string.match("lua lua", "hello"))
print(string.match("today is 27/7/2015", "%d+/%d+/%d+"))

-->output
lua
lua
nil
27/7/2015
```

string.gmatch(s, p)

返回一个迭代器函数，通过这个迭代器函数可以遍历到在字符串s中出现模式串p的所有地方。

> 示例代码

```
s = "hello world from Lua"
for w in string.gmatch(s, "%a+") do  --匹配最长连续且只含字母的字符串
    print(w)
end

-->output
hello
world
from
Lua

t = {}
s = "from=world, to=Lua"
for k, v in string.gmatch(s, "(%a+)=(%a+)") do  --匹配两个最长连续且只含字母的
    t[k] = v                                    --字符串，它们之间用等号连接
end
for k, v in pairs(t) do
print (k,v)
end

-->output
to      Lua
from    world
```

string.rep(s, n)

返回字符串s的n次拷贝。

> 示例代码

```
print(string.rep("abc", 3)) --拷贝3次"abc"
```

```
-->output  abcabcabc
```

string.sub(s, i [, j])
返回字符串s中，索引i到索引j之间的子字符串。当j缺省时，默认为-1，也就是字符串s的最后位置。 i可以为负数。当索引i在字符串s的位置在索引j的后面时，将返回一个空字符串。

> 示例代码

```
print(string.sub("Hello Lua", 4, 7))
print(string.sub("Hello Lua", 2))
print(string.sub("Hello Lua", 2, 1)) --看到返回什么了吗
print(string.sub("Hello Lua", -3, -1))

-->output
lo L
ello Lua

Lua
```

string.gsub(s, p, r [, n])
将目标字符串s中所有的子串p替换成字符串r。可选参数n，表示限制替换次数。返回值有两个，第一个是被替换后的字符串，第二个是替换了多少次。

> 示例代码

```
print(string.gsub("Lua Lua Lua", "Lua", "hello"))
print(string.gsub("Lua Lua Lua", "Lua", "hello", 2)) --指明第四个参数

-->output
hello hello hello   3
hello hello Lua     2
```

string.reverse (s)
接收一个字符串s，返回这个字符串的反转。

> 示例代码

```
print(string.reverse("Hello Lua"))  -->output  auL olleH
```

字符串连接

使用 ".."字符串连接符，能够把多个字符串连接起来。如果连接符两端出现不是字符串，那么会自动转换成字符串。

> 示例代码

```
print( "hello " .. "lua" )
print( "today:" .. os.date()) --你的输出和我一样吗？

-->output
hello lua
today:07/29/15 17:29:24
```

Table库

table library

table库是由一些辅助函数构成的，这些函数将table作为数组来操作。

table.concat (table [, sep [, i [, j]]])
对于元素是string或者number类型的表table，返回table[i]..sep..table[i+1] ··· sep..table[j]连接成的字符串。填充字符串sep默认为空白字符串。起始索引位置i默认为1，结束索引位置j默认是table的长度。如果i大于j，返回一个空字符串。

> 示例代码

```
a = {1, 3, 5, "hello" }
print(table.concat(a))
print(table.concat(a, "|"))
print(table.concat(a, " ", 4, 2))
print(table.concat(a, " ", 2, 4))

-->output
135hello
1|3|5|hello

3 5 hello
```

table.insert (table, [pos ,] value)
在表table的pos索引位置插入value，其它元素向后移动到空的地方。pos的默认值是表的长度加一，即默认是插在表的最后。

> 示例代码

```
a = {1, 8}              --a[1] = 1,a[2] = 8
table.insert(a, 1, 3)   --在表索引为1处插入3
print(a[1], a[2], a[3])
table.insert(a, 10)     --在表的最后插入10
print(a[1], a[2], a[3], a[4])

-->output
3    1    8
3    1    8    10
```

table.maxn (table)
返回表table的最大索引编号；如果此表没有正的索引编号，返回0。

> 示例代码

```
a = {}
a[-1] = 10
print(table.maxn(a))
a[5] = 10
print(table.maxn(a))

-->output
0
5
```

table.remove (table [, pos])
在表table中删除索引为pos（pos只能是number型）的元素，并返回这个被删除的元素，它后面所有元素的索引值都会减一。pos的默认值是表的长度，即默认是删除表的最后一个元素。

> 示例代码

```
a = { 1, 2, 3, 4}
print(table.remove(a, 1)) --删除速索引为1的元素
print(a[1], a[2], a[3], a[4])

print(table.remove(a))   --删除最后一个元素
print(a[1], a[2], a[3], a[4])

-->output
1
2    3    4    nil
4
2    3    nil    nil
```

table.sort (table [, comp])
按照给定的比较函数comp给表table排序，也就是从table[1]到table[n]，这里n表示table的长度。比较函数有两个参数，如果希望第一个参数排在第二个的前面，就应该返回true，否则返回false。如果比较函数comp没有给出，默认从小到大排序。

> 示例代码

```
function compare(x, y) --从大到小排序
   return x > y    --如果第一个参数大于第二个就返回true, 否则返回false
end

a = { 1, 7, 3, 4, 25}
table.sort(a)          --默认从小到大排序
print(a[1], a[2], a[3], a[4], a[5])
table.sort(a, compare) --使用比较函数进行排序
print(a[1], a[2], a[3], a[4], a[5])

-->output
1    3    4    7    25
25    7    4    3    1
```

日期时间函数

日期时间函数

在lua中，函数time、date和difftime提供了所有的日期和时间功能。

os.time ([table])

如果不使用参数table调用time函数，它会返回当前的时间和日期（它表示从某一时刻到现在的秒数）。如果用table参数，它会返回一个数字，表示该table中所描述的日期和时间（它表示从某一时刻到table中描述日期和时间的秒数）。table的字段如下：

字段名称	取值范围
year	四位数字
month	1--12
day	1--31
hour	0--23
min	0--59
sec	0--61
isdst	boolean（true表示夏令时）

对于time函数，如果参数为table，那么table中必须含有year、month、day字段。其他字缺省时段默认为中午（12:00:00）。

> 示例代码：（地点为北京）

```
print(os.time())    -->output  1438243393
a = { year = 1970, month = 1, day = 1, hour = 8, min = 1 }
print(os.time(a))   -->output  60
```

os.difftime (t2, t1)

返回t1到t2的时间差，单位为秒。

> 示例代码:

```
local day1 = { year = 2015, month = 7, day = 30 }
local t1 = os.time(day1)
```

```
local day2 = { year = 2015, month = 7, day = 31 }
local t2 = os.time(day2)
print(os.difftime(t2, t1))   -->output  86400
```

os.date ([format [, time]])
把一个表示日期和时间的数值，转换成更高级的表现形式。其第一个参数format是一个格式化字符串，描述了要返回的时间形式。第二个参数time就是日期和时间的数字表示，缺省时默认为当前的时间。使用格式字符 "*t"，创建一个时间表。

> 示例代码：

```
local tab1 = os.date("*t")  --返回一个描述当前日期和时间的表
local ans1 = "{"
for k, v in pairs(tab1) do  --把tab1转换成一个字符串
    ans1 = string.format("%s %s = %s,", ans1, k, tostring(v))
end

ans1 = ans1 .. "}"
print("tab1 = ", ans1)

local tab2 = os.date("*t", 360)  --返回一个描述日期和时间数为360秒的表
local ans2 = "{"
for k, v in pairs(tab2) do      --把tab2转换成一个字符串
    ans2 = string.format("%s %s = %s,", ans2, k, tostring(v))
end

ans2 = ans2 .. "}"
print("tab2 = ", ans2)

-->output
tab1 = { hour = 17, min = 28, wday = 5, day = 30, month = 7, year = 2015,
 sec = 10, yday = 211, isdst = false,}
tab2 = { hour = 8, min = 6, wday = 5, day = 1, month = 1, year = 1970, se
c = 0, yday = 1, isdst = false,}
```

该表中除了使用到了time函数参数table的字段外，这还提供了星期（wday，星期天为1）和一年中的第几天（yday，一月一日为1）。除了使用 "*t" 格式字符串外，如果使用带标记（见下表）的特殊字符串，os.data函数会将相应的标记位以时间信息进行填充，得到一个包含时间的字符串。表如下：

格式字符	含义
%a	一星期中天数的简写（例如：Wed）
%A	一星期中天数的全称（例如：Wednesday）

%b	月份的简写（例如：Sep）
%B	月份的全称（例如：September）
%c	日期和时间（例如：07/30/15 16:57:24）
%d	一个月中的第几天[01 ~ 31]
%H	24小时制中的小时数[00 ~ 23]
%I	12小时制中的小时数[01 ~ 12]
%j	一年中的第几天[001 ~ 366]
%M	分钟数[00 ~ 59]
%m	月份数[01 ~ 12]
%p	“上午（am）”或“下午（pm）”
%S	秒数[00 ~ 59]
%w	一星期中的第几天[1 ~ 7 = 星期天 ~ 星期六]
%x	日期（例如：07/30/15）
%X	时间（例如：16:57:24）
%y	两位数的年份[00 ~ 99]
%Y	完整的年份（例如：2015）
%%	字符'%'

> 示例代码：

```
print(os.date("today is %A, in %B"))
print(os.date("now is %x %X"))

-->output
today is Thursday, in July
now is 07/30/15 17:39:22
```

数学库函数

数学库

lua数学库由一组标准的数学函数构成。数学库的引入丰富了lua编程语言的功能，同时也方便了程序的编写。常用数学函数见下表：

函数名	函数功能
math.rad(x)	角度x转换成弧度
math.deg(x)	弧度x转换成角度
math.max(x, ...)	返回参数中值最大的那个数，参数必须是number型
math.min(x, ...)	返回参数中值最小的那个数，参数必须是number型
math.random ([m [, n]])	不传入参数时，返回 一个在区间[0,1)内均匀分布的伪随机实数；只使用一个整数参数m时，返回一个在区间[1, m]内均匀分布的伪随机整数；使用两个整数参数时，返回一个在区间[m, n]内均匀分布的伪随机整数
math.randomseed (x)	为伪随机数生成器设置一个种子x，相同的种子将会生成相同的数字序列
math.abs(x)	返回x的绝对值
math.fmod(x, y)	返回 x对y取余数
math.pow(x, y)	返回x的y次方
math.sqrt(x)	返回x的算术平方根
math.exp(x)	返回自然数e的x次方
math.log(x)	返回x的自然对数
math.log10(x)	返回以10为底，x的对数
math.floor(x)	返回最大且不大于x的整数
math.ceil(x)	返回最小且不小于x的整数
math.pi	圆周率
math.sin(x)	求弧度x的正弦值
math.cos(x)	求弧度x的余弦值
math.tan(x)	求弧度x的正切值
math.asin(x)	求x的反正弦值
math.acos(x)	求x的反余弦值
math.atan(x)	求x的反正切值

> 示例代码：

```
print(math.pi)           -->output  3.1415926535898
print(math.rad(180))     -->output  3.1415926535898
print(math.deg(math.pi)) -->output  180

print(math.sin(1))       -->output  0.8414709848079
print(math.cos(math.pi)) -->output  -1
print(math.tan(math.pi / 4))  -->output  1

print(math.atan(1))      -->output  0.78539816339745
print(math.asin(0))      -->output  0

print(math.max(-1, 2, 0, 3.6, 9.1))     -->output  9.1
print(math.min(-1, 2, 0, 3.6, 9.1))     -->output  -1

print(math.fmod(10.1, 3))   -->output  1.1
print(math.sqrt(360))      -->output  18.97366596101

print(math.exp(1))         -->output  2.718281828459
print(math.log(10))        -->output  2.302585092994
print(math.log10(10))      -->output  1

print(math.floor(3.1415))  -->output  3
print(math.ceil(7.998))    -->output  8
```

另外使用nath.random()函数获得伪随机数时，如果不使用math.randomseed ()设置伪随机数生成种子或者设置相同的伪随机数生成种子，那么得得到的伪随机数序列是一样的。

> 示例代码：

```
math.randomseed (100) --把种子设置为100
print(math.random())         -->output  0.0012512588885159
print(math.random(100))      -->output  57
print(math.random(100, 360)) -->output  150
```

稍等片刻，再次运行上面的代码。

```
math.randomseed (100) --把种子设置为100
print(math.random())         -->output  0.0012512588885159
print(math.random(100))      -->output  57
print(math.random(100, 360)) -->output  150
```

两次运行的结果一样。为了避免每次程序启动时得到的都是相同的伪随机数序列，通常是使用当前时间作为种子。

> 修改上例中的代码：

```
math.randomseed (os.time())   --把100换成os.time()
print(math.random())          -->output 0.88369396038697
print(math.random(100))       -->output 66
print(math.random(100, 360))  -->output 228
```

稍等片刻，再次运行上面的代码。

```
math.randomseed (os.time())   --把100换成os.time()
print(math.random())          -->output 0.88946195867794
print(math.random(100))       -->output 68
print(math.random(100, 360))  -->output 129
```

文件操作

文件操作

lua I/O库提供两种不同的方式处理文件：

1、隐式文件描述：设置一个默认的输入或输出文件，然后在这个文件上进行所有的输入或输出操作。所有的操作函数由io表提供。

> 打开已经存在的test1.txt文件，并读取里面的内容

```
file = io.input("test1.txt")  --使用io.input()函数打开文件

repeat
    line = io.read()      --逐行读取内容，文件结束时返回nil
    if nil == line then
        break
    end
    print(line)
until (false)

io.close(file)       --关闭文件

-->output
my test file
hello
lua
```

> 在test1.txt文件的最后添加一行"hello world"

```
file = io.open("test1.txt", "a+") --使用io.open()函数，以添加模式打开文件
io.output(file)                    --使用io.output()函数，设置默认输出文件
io.write("\nhello world")          --使用io.write()函数，把内容写到文件
io.close(file)
```

在相应目录下打开test1.txt文件，查看文件内容发生的变化。

2、显示文件描述：使用file:XXX()函数方式进行操作,其中file为io.open()返回的文件句柄。

> 打开已经存在的test2.txt文件，并读取里面的内容

```
file = io.open("test2.txt", "r")  --使用io.open()函数，以只读模式打开文件

for line in file:lines() do           --使用file:lines()函数逐行读取文件
   print(line)
end

file:close()

-->output
my test2
hello lua
```

> 在test2.txt文件的最后添加一行"hello world"

```
file = io.open("test2.txt", "a")  --使用io.open()函数，以添加模式打开文件
file:write("\nhello world")       --使用file:open()函数，在文件的最后添加一行
内容
file:close()
```

在相应目录下打开test2.txt文件，查看文件内容发生的变化。

文件操作函数

io.open (filename [, mode])

按指定的模式mode，打开一个文件名为filename的文件，成功则返回文件句柄，失败则返回nil加错误信息。模式：

模式	含义	文件不存在时
"r"	读模式 (默认)	返回nil加错误信息
"w"	写模式	创建文件
"a"	添加模式	创建文件
"r+"	更新模式，保存之前的数据	返回nil加错误信息
"w+"	更新模式，清除之前的数据	创建文件
"a+"	添加更新模式，保存之前的数据,在文件尾进行添加	创建文件

模式字符串后面可以有一个'b'，用于在某些系统中打开二进制文件。

file:close ()

关闭文件。注意：当文件句柄被垃圾收集后，文件将自动关闭。句柄将变为一个不可预知的值。

io.close ([file])
关闭文件，和file:close()的作用相同。没有参数file时，关闭默认输出文件。

file:flush ()
把写入缓冲区的所有数据写入到文件file中。

io.flush ()
相当于file:flush(),把写入缓冲区的所有数据写入到默认输出文件。

io.input ([file])
当使用一个文件名调用时，打开这个文件（以文本模式），并设置文件句柄为默认输入文件；当使用一个文件句柄调用时，设置此文件句柄为默认输入文件；当不使用参数调用时，返回默认输入文件句柄。

file:lines ()
返回一个迭代函数,每次调用将获得文件中的一行内容,当到文件尾时，将返回nil,但不关闭文件。

io.lines ([filename])
打开指定的文件filename为读模式并返回一个迭代函数,每次调用将获得文件中的一行内容,当到文件尾时，将返回nil,并自动关闭文件。若不带参数时io.lines() 等价于io.input():lines(); 读取默认输入设备的内容，结束时不关闭文件。

io.output ([file])
类似于io.input，但操作在默认输出文件上。

file:read (···)
按指定的格式读取一个文件。按每个格式将返回一个字符串或数字,如果不能正确读取将返回nil,若没有指定格式将指默认按行方式进行读取。格式：

格式	含义
"*n"	读取一个数字
"*a"	从当前位置读取整个文件。若当前位置为文件尾，则返回空字符串
"*l"	读取下一行的内容。若为文件尾，则返回nil。(默认)
number	读取指定字节数的字符。若为文件尾，则返回nil。如果number为0,则返回空字符串，若为文件尾,则返回nil

io.read (···)
相当于io.input():read

io.type (obj)
检测obj是否一个可用的文件句柄。如果obj是一个打开的文件句柄，则返回"file"；如果obj是一个已关闭的文件句柄，则返回"closed file"；如果obj不是一个文件句柄，则返回nil。

file:write (···)
把每一个参数的值写入文件。参数必须为字符串或数字，若要输出其它值，则需通过tostring或string.format进行转换。

io.write (···)
相当于io.output():write。

file:seek ([whence] [, offset])
设置和获取当前文件位置,成功则返回最终的文件位置(按字节，相对于文件开头),失败则返回nil加错误信息。缺省时，whence默认为"cur"，offset默认为0。参数whence：

whence	含义
"set"	文件开始
"cur"	文件当前位置(默认)
"end"	文件结束

file:setvbuf (mode [, size])
设置输出文件的缓冲模式。模式：

模式	含义
"no"	没有缓冲，即直接输出
"full"	全缓冲，即当缓冲满后才进行输出操作(也可调用flush马上输出)
"line"	以行为单位，进行输出

最后两种模式,size可以指定缓冲的大小(按字节)，忽略size将自动调整为最佳的大小。

元表

元表

在Lua中，元表 (metatable) 的表现行为类似于C++语言中的函数（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。

设置元表的方法很简单，如下：

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

上面的代码可以简写成如下的一行代码：

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

修改表的操作符行为
通过重载"__add"元方法来计算集合的并集实例：

```
set1 = {10, 20, 30}--集合
set2 = {20, 40, 50}--集合

union = function (self, another) --将用于重载__add的函数，注意第一个参数是self
    local set = {}
    local result = {}
    --利用数组来确保集合的互异性
    for i, j in pairs(self) do set[j] = true end
    for i, j in pairs(another) do set[j] = true end
    --加入结果集合
    for i, j in pairs(set) do table.insert(result, i) end
    return result
end
setmetatable(set1, {__add = union}) --重载set1表的__add元方法

set3 = set1 + set2
for _,j in pairs(set3) do
    io.write(j.." ")  -->输出结果30 50 20 40 10
end
```

除了加法可以被重载之外，Lua提供的所有操作符都可以被重载：

元方法	含义
"__add"	+ 操作
"__sub"	- 操作 其行为类似于 "add" 操作
"__mul"	* 操作 其行为类似于 "add" 操作
"__div"	/ 操作 其行为类似于 "add" 操作
"__mod"	% 操作 其行为类似于 "add" 操作
"__pow"	^ （幂）操作 其行为类似于 "add" 操作
"__unm"	一元 - 操作
"__concat"	..（字符串连接）操作
"__len"	# 操作
"__eq"	== 操作 函数 getcomphandler 定义了 Lua 怎样选择一个处理器来作比较操作 仅在两个对象类型相同且有对应操作相同的元方法时才起效
"__lt"	< 操作
"__le"	<= 操作

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

__index元方法

下面的例子中，我们实现了在表中查找键不存在时转而在元表中查找该键的功能：

```
mytable = setmetatable({key1 = "value1"}, --原始表
  {__index = function(self, key)       --重载函数
    if key == "key2" then
      return "metatablevalue"
    else
      return self[key]
    end
  end
```

```
})

print(mytable.key1,mytable.key2)  -->value1 metatablevalue
```

关于index元方法，有很多比较高阶的技巧，例如：index的元方法不需要非是一个函数，他也可以是一个表。

```
t = setmetatable({[1] = "hello"}, {__index = {[2] = "world"}})
print(t[1], t[2])  -->hello world
```

第一句代码有点绕，解释一下：先是把{index = {}}作为元表，但index接受一个表，而不是函数，这个表中包含[2] = "world"这个键值对。所以当t[2]去在自身的表中找不到时，在__index的表中去寻找，然后找到了[2] = "world"这个键值对。

index元方法还可以实现给表中每一个值赋上默认值；和newindex元方法联合监控对表的读取、修改等比较高阶的功能，待读者自己去开发吧。

__tostring元方法
与Java中的toString()函数类似，可以实现自定义的字符串转换。

```
arr = {1, 2, 3, 4}
arr = setmetatable(arr, {__tostring = function (self)
    local result = '{'
    local sep = ''
    for _, i in pairs(self) do
        result = result ..sep .. i
        sep = ', '
    end
    result = result .. '}'
    return result
end})
print(arr)  --> {1, 2, 3, 4}
```

__call元方法
__call元方法的功能类似于C++中的仿函数，使得普通的表也可以被调用。

```
functor = {}
function func1(self, arg)
  print ("called from", arg)
end

setmetatable(functor, {__call = func1})

functor("functor")  --> called from functor
```

```
print(functor)  --> table: 0x00076fc8   后面这串数字可能不一样
```

__metatable元方法

假如我们想保护我们的对象使其使用者既看不到也不能修改 metatables。我们可以对 metatable 设置了__metatable 的值， getmetatable 将返回这个域的值， 而调用 setmetatable将会出错：

```
Object = setmetatable({}, {__metatable = "You cannot access here"})

print(getmetatable(Object)) --> You cannot access here
setmetatable(Object, {})  --> 引发编译器报错
```

面向对象编程

Lua面向对象编程

类

在 Lua 中，我们可以使用表和函数实现面向对象。将函数和相关的数据放置于同一个表中就形成了一个对象。

```
Account = {balance = 0}
function Account:deposit (v)  --注意，此处使用冒号，可以免写self关键字；如果使用
.号，第一个参数必须是self
    self.balance = self.balance + v
end

function Account:withdraw (v)  --注意，此处使用冒号，可以免写self关键字；
    if self.balance > v then
        self.balance = self.balance - v
    else
        error("insufficient funds")
    end
end

function Account:new (o)  --注意，此处使用冒号，可以免写self关键字；
    o = o or {}  -- create object if user does not provide one
    setmetatable(o, {__index = self})
    return o
end

a = Account:new()
a:deposit(100)
b = Account:new()
b:deposit(50)
print(a.balance)  -->100
print(b.balance)  -->50
--本来笔者开始是自己写的例子，但发现的确不如lua作者给的例子经典，所以还是沿用作者的代码
。
```

上面这段代码"setmetatable(o, {index = self})"这句话值得注意。根据我们在元表这一章学到的知识，我们明白，setmetatable将Account作为新建'o'表的原型，所以当o在自己的表内找不到'balance'、'withdraw'这些方法和变量的时候，便会到index所指定的Account类型中去寻找。

继承

继承可以用元表实现，它提供了在父类中查找存在的方法和变量的机制。

```
--定义继承
--定义继承
SpecialAccount = Account:new({limit = 1000}) --开启一个特殊账户类型，这个类型
的账户可以取款超过余额限制1000元
function SpecialAccount:withdraw (v)
    if v - self.balance >= self:getLimit() then
        error("insufficient funds")
    end
    self.balance = self.balance - v
end

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

spacc = SpecialAccount:new()
spacc:withdraw(100)
print(spacc.balance)  --> -100
acc = Account:new()
acc:withdraw(100)     --> 超出账户余额限制，抛出一个错误
```

多重继承

多重继承肯定不能采用我们在单继承中的所使用的方法，因为直接采用setmetatable的方式，会造成metatable的覆盖。在多重继承中，我们自己利用'__index'元方法定义恰当的访问行为。

```
local function search (k, plist)
    for i=1, table.getn(plist) do
    local v = plist[i][k]  -- try 'i'-th superclass
    if v then return v end
    end
end

function createClass (...)
    local c = {} -- new class
    -- class will search for each method in the list of its
    -- parents (`args' is the list of parents)
    args = {...}
    setmetatable(c, {__index = function (self, k)
        return search(k, args)
    end})

    -- prepare `c' to be the metatable of its instances
    c.__index = c

    -- define a new constructor for this new class
    function c:new (o)
        o = o or {}
        setmetatable(o, c)
        return o
    end
```

```
    -- return new class
    return c
end
```

解释一下上面的代码。我们定义了一个通用的创建多重继承类的函数'createClass'，这个函数可以接受多个类。如何让我们新建的多重继承类恰当地访问从不同类中继承来的函数或者成员变量呢？我们就用到了'search'函数，该函数接受两个参数，第一个参数是想要访问的类成员的名字，第二个参数是被继承的类列表。通过一个for循环在列表的各个类中寻找想要访问成员。

我们再定一个新类，来验证'createClass'的正确性。

```
Named = {}
function Named:getname ()
    return self.name
end
function Named:setname (n)
    self.name = n
end

NamedAccount = createClass(Account, Named)   --同时继承Account 和 Named两个
类
account = NamedAccount:new{name = "Paul"}    --使用这个多重继承类定义一个实例
print(account:getname())          --> Pauls
account:deposit(100)
print(account.balance)            --> 100
```

成员私有性

在面向对象当中，如何将成员内部实现细节对使用者隐藏，也是值得关注的一点。在 Lua 中，成员的私有性，使用类似于函数闭包的形式来实现。在我们之前的银行账户的例子中，我们使用一个工厂方法来创建新的账户实例，通过工厂方法对外提供的闭包来暴露对外接口。而不想暴露在外的例如balace成员变量，则被很好的隐藏起来。

```
function newAccount (initialBalance)
    local self = {balance = initialBalance}
    local withdraw = function (v)
        self.balance = self.balance - v
    end
    local deposit = function (v)
        self.balance = self.balance + v
    end
    local getBalance = function () return self.balance end
    return {
        withdraw = withdraw,
        deposit = deposit,
        getBalance = getBalance
```

```
    }
end

a = newAccount(100)
a.deposit(100)
print(a.getBalance()) --> 200
print(a.balance)      --> nil
```

FFI

FFI

调用C函数

`ffi.C` 使用默认的C标准库命名空间，这使得我们可以简单地调用C标准库中的函数。同时，FFI库还会自动检测到 `sdfcall` 函数，所以我们也不用去声明那些函数。当Lua中基本数值类型与被调用的C函数参数不一致时，FFI库会自动完成数值类型的转换。

> 我们来看一个调用FFI库的示例

```
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。

使用C数据结构
userdata 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 userdata自定义一组操作。 userdata 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	x = *p;	x = p[0]
int p; | p = y;	p[0] = y	
Pointer indexing	x = p[i];	x = p[i]
int i, *p;	p[i+1] = y;	p[i+1] = y
Array indexing	x = a[i];	x = a[i]
int i, a[];	a[i+1] = y;	a[i+1] = y
struct/union dereference	x = s.field;	x = s.field
struct foo s;	s.field = y;	s.field = y
struct/union pointer deref.	x = sp->field;	x = s.field
struct foo *sp;	sp->field = y;	s.field = y
int i, *p;	y = p - i;	y = p - i
Pointer difference	x = p1 - p2;	x = p1 - p2
Array element pointer	x = &a[i];	x = a+i

LuaRestyRedisLibrary

LuaRestyRedisLibrary

select+set_keepalive组合操作引起的数据读写错误

select+set_keepalive组合操作引起的数据读写错误

在高并发编程中，我们必须要使用连接池技术，通过减少建连、拆连次数来提高通讯速度。

错误示例代码：

```
local redis = require "resty.redis"
local red = redis:new()

red:set_timeout(1000) -- 1 sec

-- or connect to a unix domain socket file listened
-- by a redis server:
--     local ok, err = red:connect("unix:/path/to/redis.sock")

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

ok, err = red:select(1)
if not ok then
    ngx.say("failed to select db: ", err)
    return
end

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

ok, err = red:set("dog", "an animal")
if not ok then
    ngx.say("failed to set dog: ", err)
    return
end

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

-- put it into the connection pool of size 100,
-- with 10 seconds max idle time
local ok, err = red:set_keepalive(10000, 100)
if not ok then
    ngx.say("failed to set keepalive: ", err)
    return
end
```

如果单独执行这个用例，没有任何问题，用例是成功的。但是这段“没问题”的代码，却导

致了诡异的现象。

我们的大部分redis请求的代码应该是类似这样的：

```
local redis = require "resty.redis"
local red = redis:new()

red:set_timeout(1000) -- 1 sec

-- or connect to a unix domain socket file listened
-- by a redis server:
--     local ok, err = red:connect("unix:/path/to/redis.sock")

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

ok, err = red:set("cat", "an animal too")
if not ok then
    ngx.say("failed to set cat: ", err)
    return
end

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

-- put it into the connection pool of size 100,
-- with 10 seconds max idle time
local ok, err = red:set_keepalive(10000, 100)
if not ok then
    ngx.say("failed to set keepalive: ", err)
    return
end
```

这时候第二个示例代码在生产运行中，会出现cat偶会被写入到数据库1上，且几率大约1%左右。出错的原因在于错误示例代码使用了select(1)操作，并且使用了长连接，那么她就会潜伏在连接池中。当下一个请求刚好从连接池中把他选出来，又没有重置select(0)操作，那么后面所有的数据操作就都会默认触发在数据库1上了。怎么解决，不用我说了吧？

redis接口的二次封装（简化建连、拆连等细节）

redis接口的二次封装

> 先看一下官方的调用示例代码：

```
local redis = require "resty.redis"
local red = redis:new()

red:set_timeout(1000) -- 1 sec

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

ok, err = red:set("dog", "an animal")
if not ok then
    ngx.say("failed to set dog: ", err)
    return
end

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

-- put it into the connection pool of size 100,
-- with 10 seconds max idle time
local ok, err = red:set_keepalive(10000, 100)
if not ok then
    ngx.say("failed to set keepalive: ", err)
    return
end
```

这是一个标准的redis接口调用，如果你的代码中redis被调用频率不高，那么这段代码不会有任何问题。但如果你的项目重度依赖redis，工程中有大量的代码在重复这样一个动作，创建连接-->数据操作-->关闭连接（或放到连接池）这个完整的链路调用完毕，甚至还要考虑不同的return情况做不同处理，就很快发现代码中有大量的重复。

`Lua` 是不支持面向对象的。很多人用尽各种招术利用元表来模拟。可是，`Lua` 的发明者似乎不想看到这样的情形，因为他们把取长度的 `__len` 方法以及析构函数 `__gc` 留给了 C API。纯 Lua 只能望洋兴叹。

> 我们期望的代码应该是这样的：

```
local red = redis:new()
local ok, err = red:set("dog", "an animal")
if not ok then
    ngx.say("failed to set dog: ", 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)
```

并且他自身具备以下几个特征：

- new、connect函数合体，使用时只负责申请，尽量少关心什么时候具体连接、释放
- 默认redis数据库连接地址，但是允许自定义
- 每次redis使用完毕，自动释放redis连接到连接池供其他请求复用
- 要支持redis的重要优化手段 pipeline

> 不卖关子，只要干货，我们最后是这样干的，可以这里看到gist代码

```
-- file name: resty/redis_iresty.lua
local redis_c = require "resty.redis"

local ok, new_tab = pcall(require, "table.new")
if not ok or type(new_tab) ~= "function" then
    new_tab = function (narr, nrec) return {} end
end

local _M = new_tab(0, 155)
_M._VERSION = '0.01'

local commands = {
    "append",            "auth",              "bgrewriteaof",
    "bgsave",            "bitcount",          "bitop",
    "blpop",             "brpop",
    "brpoplpush",        "client",            "config",
    "dbsize",
```

```
    "debug",              "decr",               "decrby",
    "del",                "discard",            "dump",
    "echo",
    "eval",               "exec",               "exists",
    "expire",             "expireat",           "flushall",
    "flushdb",            "get",                "getbit",
    "getrange",           "getset",             "hdel",
    "hexists",            "hget",               "hgetall",
    "hincrby",            "hincrbyfloat",       "hkeys",
    "hlen",
    "hmget",               "hmset",      "hscan",
    "hset",
    "hsetnx",             "hvals",              "incr",
    "incrby",             "incrbyfloat",        "info",
    "keys",
    "lastsave",           "lindex",             "linsert",
    "llen",               "lpop",               "lpush",
    "lpushx",             "lrange",             "lrem",
    "lset",               "ltrim",              "mget",
    "migrate",
    "monitor",            "move",               "mset",
    "msetnx",             "multi",              "object",
    "persist",            "pexpire",            "pexpireat",
    "ping",               "psetex",             "psubscribe",
    "pttl",
    "publish",      --[[ "punsubscribe", ]]   "pubsub",
    "quit",
    "randomkey",          "rename",             "renamenx",
    "restore",
    "rpop",               "rpoplpush",          "rpush",
    "rpushx",             "sadd",               "save",
    "scan",               "scard",              "script",
    "sdiff",              "sdiffstore",
    "select",             "set",                "setbit",
    "setex",              "setnx",              "setrange",
    "shutdown",           "sinter",             "sinterstore",
    "sismember",          "slaveof",            "slowlog",
    "smembers",           "smove",              "sort",
    "spop",               "srandmember",        "srem",
    "sscan",
    "strlen",       --[[ "subscribe",  ]]     "sunion",
    "sunionstore",        "sync",               "time",
    "ttl",
    "type",         --[[ "unsubscribe", ]]    "unwatch",
    "watch",              "zadd",               "zcard",
    "zcount",             "zincrby",            "zinterstore",
    "zrange",             "zrangebyscore",      "zrank",
    "zrem",               "zremrangebyrank",    "zremrangebyscore",
    "zrevrange",          "zrevrangebyscore",   "zrevrank",
    "zscan",
    "zscore",             "zunionstore",        "evalsha"
}

local mt = { __index = _M }

local function is_redis_null( res )
```

```
    if type(res) == "table" then
        for k,v in pairs(res) do
            if v ~= ngx.null then
                return false
            end
        end
        return true
    elseif res == ngx.null then
        return true
    elseif res == nil then
        return true
    end

    return false
end

-- change connect address as you need
function _M.connect_mod( self, redis )
    redis:set_timeout(self.timeout)
    return redis:connect("127.0.0.1", 6379)
end

function _M.set_keepalive_mod( redis )
    -- put it into the connection pool of size 100, with 60 seconds max i
dle time
    return redis:set_keepalive(60000, 1000)
end

function _M.init_pipeline( self )
    self._reqs = {}
end

function _M.commit_pipeline( self )
    local reqs = self._reqs

    if nil == reqs or 0 == #reqs then
        return {}, "no pipeline"
    else
        self._reqs = nil
    end

    local redis, err = redis_c:new()
    if not redis then
        return nil, err
    end

    local ok, err = self:connect_mod(redis)
    if not ok then
        return {}, err
    end

    redis:init_pipeline()
    for _, vals in ipairs(reqs) do
        local fun = redis[vals[1]]
        table.remove(vals , 1)
```

```
            fun(redis, unpack(vals))
        end

        local results, err = redis:commit_pipeline()
        if not results or err then
            return {}, err
        end

        if is_redis_null(results) then
            results = {}
            ngx.log(ngx.WARN, "is null")
        end
        -- table.remove (results , 1)

        self.set_keepalive_mod(redis)

        for i,value in ipairs(results) do
            if is_redis_null(value) then
                results[i] = nil
            end
        end

        return results, err
    end

    function _M.subscribe( self, channel )
        local redis, err = redis_c:new()
        if not redis then
            return nil, err
        end

        local ok, err = self:connect_mod(redis)
        if not ok or err then
            return nil, err
        end

        local res, err = redis:subscribe(channel)
        if not res then
            return nil, err
        end

        res, err = redis:read_reply()
        if not res then
            return nil, err
        end

        redis:unsubscribe(channel)
        self.set_keepalive_mod(redis)

        return res, err
    end

    local function do_command(self, cmd, ... )
        if self._reqs then
            table.insert(self._reqs, {cmd, ...})
            return
```

```
    end

    local redis, err = redis_c:new()
    if not redis then
        return nil, err
    end

    local ok, err = self:connect_mod(redis)
    if not ok or err then
        return nil, err
    end

    local fun = redis[cmd]
    local result, err = fun(redis, ...)
    if not result or err then
        -- ngx.log(ngx.ERR, "pipeline result:", result, " err:", err)
        return nil, err
    end

    if is_redis_null(result) then
        result = nil
    end

    self.set_keepalive_mod(redis)

    return result, err
end

function _M.new(self, opts)
    opts = opts or {}
    local timeout = (opts.timeout and opts.timeout * 1000) or 1000
    local db_index= opts.db_index or 0

    for i = 1, #commands do
        local cmd = commands[i]
        _M[cmd] =
                function (self, ...)
                    return do_command(self, cmd, ...)
                end
    end

    return setmetatable({
            timeout = timeout,
            db_index = db_index,
            _reqs = nil }, mt)
end

return _M
```

> 调用示例代码：

```
local redis = require "resty.redis_iresty"
```

```
local red = redis:new()

local ok, err = red:set("dog", "an animal")
if not ok then
    ngx.say("failed to set dog: ", err)
    return
end

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

在最终的示例代码中看到，所有的连接创建、销毁连接、连接池部分，都被完美隐藏了，我们只需要业务就可以了。妈妈再也不用担心我把redis搞垮了。

Todo list：目前 `resty.redis` 并没有对redis 3.0的集群API做支持，既然统一了redis的入口、出口，那么对这个 `redis_iresty` 版本做适当调整完善，就可以支持redis 3.0的集群协议。由于我们目前还没引入redis集群，这里也希望有使用的同学贡献自己的补丁或文章。

redis接口的二次封装（发布订阅）

redis接口的二次封装（发布订阅）

其实这一小节完全可以放到上一个小结，只是这里用了完全不同的玩法，所以我还是决定单拿出来分享一下这个小细节。

上一小结有关订阅部分的代码，请看：

```
function _M.subscribe( self, channel )
    local redis, err = redis_c:new()
    if not redis then
        return nil, err
    end

    local ok, err = self:connect_mod(redis)
    if not ok or err then
        return nil, err
    end

    local res, err = redis:subscribe(channel)
    if not res then
        return nil, err
    end

    res, err = redis:read_reply()
    if not res then
        return nil, err
    end

    redis:unsubscribe(channel)
    self.set_keepalive_mod(redis)

    return res, err
end
```

其实这里的实现是有问题的，各位看官，你能发现这段代码的问题么？给个提示，在高并发订阅场景下，极有可能存在漏掉部分订阅信息。原因在与每次订阅到内容后，都会把redis对象进行释放，处理完订阅信息后再次去连接redis，在这个时间差里面，很可能有消息已经漏掉了。

正确的代码应该是这样的：

```
function _M.subscribe( self, channel )
    local redis, err = redis_c:new()
```

```
    if not redis then
        return nil, err
    end

    local ok, err = self:connect_mod(redis)
    if not ok or err then
        return nil, err
    end

    local res, err = redis:subscribe(channel)
    if not res then
        return nil, err
    end

    local function do_read_func ( do_read )
        if do_read == nil or do_read == true then
            res, err = redis:read_reply()
            if not res then
                return nil, err
            end
            return res
        end

        redis:unsubscribe(channel)
        self.set_keepalive_mod(redis)
        return
    end

    return do_read_func
end
```

调用示例代码：

```
local red     = redis:new({timeout=1000})
local func  = red:subscribe( "channel" )
if not func then
  return nil
end

while true do
    local res, err = func()
    if err then
        func(false)
    end
    ... ...
end

return cbfunc
```

pipeline压缩请求数量

pipeline压缩请求数量

通常情况下，我们每个操作redis的命令都以一个TCP请求发送给redis，这样的做法简单直观。然而，当我们有连续多个命令需要发送给redis时，如果每个命令都以一个数据包发送给redis，将会降低服务端的并发能力。

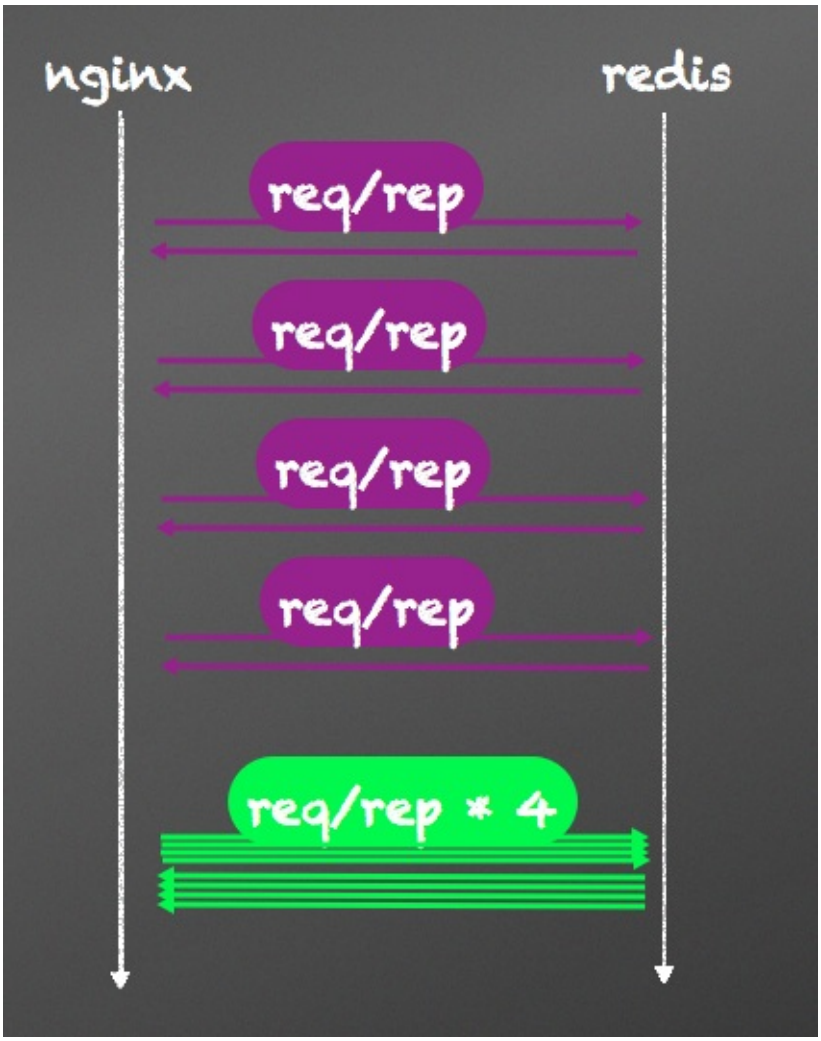
为什么呢？大家知道每发送一个TCP报文，会存在网络延时及操作系统的处理延时。大部分情况下，网络延时要远大于CPU的处理延时。如果一个简单的命令就以一个TCP报文发出，网络延时将成为系统性能瓶颈，使得服务端的并发数量上不去。

首先检查你的代码，是否明确完整使用了redis的长连接机制。作为一个服务端程序员，要对长连接的使用有一定了解，在条件允许的情况下，一定要开启长连接。验证方式也比较简单，直接用tcpdump或wireshark抓包分析一下网络数据即可。

> set_keepalive的参数：按照业务正常运转的并发数量设置，不建议使用峰值情况设置。

如果我们确定开启了长连接，发现这时候Redis的CPU的占用率还是不高，在这种情况下，就要从Redis的使用方法上进行优化。

如果我们可以把所有单次请求，压缩到一起，如下图：



很庆幸Redis早就为我们准备好了这道菜，就等着我们吃了，这道菜就叫`pipeline`。pipeline机制将多个命令汇聚到一个请求中，可以有效减少请求数量，减少网络延时。下面是对比使用pipeline的一个例子：

```
# you do not need the following line if you are using
    # the ngx_openresty bundle:
    lua_package_path "/path/to/lua-resty-redis/lib/?.lua;;";

    server {
        location /withoutpipeline {
            content_by_lua '
                local redis = require "resty.redis"
                local red = redis:new()

                red:set_timeout(1000) -- 1 sec

                -- or connect to a unix domain socket file listened
                -- by a redis server:
                --     local ok, err = red:connect("unix:/path/to/redis.s
```

```
ock")

                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("cat", "Marry")
                ngx.say("set result: ", ok)
                local res, err = red:get("cat")
                ngx.say("cat: ", res)

                ok, err = red:set("horse", "Bob")
                ngx.say("set result: ", ok)
                res, err = red:get("horse")
                ngx.say("horse: ", res)

                -- put it into the connection pool of size 100,
                -- with 10 seconds max idle time
                local ok, err = red:set_keepalive(10000, 100)
                if not ok then
                    ngx.say("failed to set keepalive: ", err)
                    return
                end
            ';
        }

        location /withpipeline {
            content_by_lua '
                local redis = require "resty.redis"
                local red = redis:new()

                red:set_timeout(1000) -- 1 sec

                -- or connect to a unix domain socket file listened
                -- by a redis server:
                --     local ok, err = red:connect("unix:/path/to/redis.s
ock")

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

                red:init_pipeline()
                red:set("cat", "Marry")
                red:set("horse", "Bob")
                red:get("cat")
                red:get("horse")
                local results, err = red:commit_pipeline()
                if not results then
                    ngx.say("failed to commit the pipelined requests: ",
err)
                    return
```

```
                end

                for i, res in ipairs(results) do
                    if type(res) == "table" then
                        if not res[1] then
                            ngx.say("failed to run command ", i, ": ", re
s[2])
                        else
                            -- process the table value
                        end
                    else
                        -- process the scalar value
                    end
                end

                -- put it into the connection pool of size 100,
                -- with 10 seconds max idle time
                local ok, err = red:set_keepalive(10000, 100)
                if not ok then
                    ngx.say("failed to set keepalive: ", err)
                    return
                end
            ';
        }
    }
```

在我们实际应用场景中，正确使用pipeline对性能的提升十分明显。我们曾经某个后台应用，逐个处理大约100万条记录需要几十分钟，经过pileline压缩请求数量后，最后时间缩小到20秒左右。做之前能预计提升性能，但是没想到提升如此巨大。

在360企业安全目前的应用中，Redis的使用瓶颈依然停留在网络上，不得不承认Redis的处理效率相当赞。

script压缩复杂请求

script压缩复杂请求

从pipeline那一章节，我们知道对于多个简单的redis命令可以汇聚到一个请求中，提升服务端的并发能力。然而，在有些场景下，我们每次命令的输入需要引用上个命令的输出，甚至可能还要对第一个命令的输出做一些加工，再把加工结果当成第二个命令的输入。pipeline难以处理这样的场景。庆幸的是，我们可以用redis里的script来压缩这些复杂命令。

script的核心思想是在redis命令里嵌入Lua脚本，来实现一些复杂操作。Redis中和脚本相关的命令有：

- EVAL
- EVALSHA
- SCRIPT EXISTS
- SCRIPT FLUSH
- SCRIPT KILL
- SCRIPT LOAD

官网上给出了这些命令的基本语法，感兴趣的同学可以到这里查阅。其中EVAL的基本语法如下：

> EVAL script numkeys key [key ...] arg [arg ...]

EVAL的第一个参数_script_是一段 Lua 脚本程序。 这段Lua脚本不需要（也不应该）定义函数。它运行在 Redis 服务器中。EVAL的第二个参数_numkeys_是参数的个数，后面的参数key（从第三个参数），表示在脚本中所用到的那些 Redis 键(key)，这些键名参数可以在 Lua 中通过全局变量 KEYS 数组，用 1 为基址的形式访问(KEYS[1] ， KEYS[2] ，以此类推)。在命令的最后，那些不是键名参数的附加参数arg [arg ...] ，可以在 Lua 中通过全局变量 ARGV 数组访问，访问的形式和 KEYS 变量类似(ARGV[1] 、 ARGV[2] ，诸如此类)。下面是执行eval命令的简单例子：

```
eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first second
1) "key1"
2) "key2"
3) "first"
4) "second"
```

openresty中已经对redis的所有原语操作进行了封装。下面我们以EVAL为例，来看一下openresty中如何利用script来压缩请求：

```
# you do not need the following line if you are using
    # the ngx_openresty bundle:
    lua_package_path "/path/to/lua-resty-redis/lib/?.lua;;";

    server {
        location /usescript {
            content_by_lua '
            local redis = require "resty.redis"
            local red = redis:new()

            red:set_timeout(1000) -- 1 sec

            -- or connect to a unix domain socket file listened
            -- by a redis server:
            --     local ok, err = red:connect("unix:/path/to/redis.sock"
)

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

            --- use scripts in eval cmd
            local id = "1"
            ok, err = red:eval([[
                local info = redis.call('get', KEYS[1])
                info = json.decode(info)
                local g_id = info.gid

                local g_info = redis.call('get', g_id)
                return g_info
                ]], 1, id)

            if not ok then
               ngx.say("failed to get the group info: ", err)
               return
            end

            -- put it into the connection pool of size 100,
            -- with 10 seconds max idle time
            local ok, err = red:set_keepalive(10000, 100)
            if not ok then
                ngx.say("failed to set keepalive: ", err)
                return
            end

            -- or just close the connection right away:
            -- local ok, err = red:close()
```

```
            -- if not ok then
            --     ngx.say("failed to close: ", err)
            --     return
            -- end
        ';
    }
}
```

从上面的例子可以看到，我们要根据一个对象的id来查询该id所属gourp的信息时，我们的第一个命令是从redis中读取id为1（id的值可以通过参数的方式传递到script中）的对象的信息（由于这些信息一般json格式存在redis中，因此我们要做一个解码操作，将info转换成lua对象）。然后提取信息中的groupid字段，以groupid作为key查询groupinfo。这样我们就可以把两个get放到一个TCP请求中，做到减少TCP请求数量，减少网络延时的效果啦。

LuaCjsonLibrary

LuaCjsonLibrary

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。它基于ECMAScript的一个子集。 JSON采用完全独立于语言的文本格式，但是也使用了类似于C语言家族的习惯（包括C、C++、C#、Java、JavaScript、Perl、Python等）。这些特性使JSON成为理想的数据交换语言。 易于人阅读和编写，同时也易于机器解析和生成(网络传输速率)。

在360企业版的接口中有大量的JSON使用，有些是REST+JSON api，还有大部分不通应用、组件之间沟通的中间数据也是有JSON来完成的。由于他可读性、体积、编解码效率相比XML有很大优势，非常值得推荐。

json解析的异常捕获

json解析的异常捕获

首先来看最最普通的一个json解析的例子（被解析的json字符串是错误的，缺少一个双引号）：

```
-- http://www.kyne.com.au/~mark/software/lua-cjson.php
-- version: 2.1 devel

local json = require("cjson")
local str  = [[ {"key:"value"} ]]

local t    = json.decode(str)
ngx.say(" --> ", type(t))

-- ... do the other things
ngx.say("all fine")
```

代码执行错误日志如下：

```
2015/06/27 00:01:42 [error] 2714#0: *25 lua entry thread aborted: runtime
 error: ...ork/git/github.com/lua-resty-memcached-server/t/test.lua:8: Ex
pected colon but found invalid token at character 9
stack traceback:
coroutine 0:
    [C]: in function 'decode'
    ...ork/git/github.com/lua-resty-memcached-server/t/test.lua:8: in fun
ction <...ork/git/github.com/lua-resty-memcached-server/t/test.lua:1>, cl
ient: 127.0.0.1, server: localhost, request: "GET /test HTTP/1.1", host:
"127.0.0.1:8001"
```

这可不是我们期望的，decode失败，居然500错误直接退了。改良了一下我们的代码：

```
local json = require("cjson")

function json_decode(str)
    local data = nil
    _, err = pcall(function(str) return json.decode(str) end, str)
    return data, err
end
```

如果需要在Lua中处理错误，必须使用函数pcall（protected call）来包装需要执行的代码。

pcall接收一个函数和要传递给后者的参数，并执行，执行结果：有错误、无错误；返回值true或者或false, errorinfo。pcall以一种"保护模式"来调用第一个参数，因此pcall可以捕获函数执行中的任何错误。有兴趣的同学，请更多了解下Lua中的异常处理。

另外，可以使用CJSON 2.1.0，该版本新增一个cjson.safe模块接口，该接口兼容cjson模块，并且在解析错误时不抛出异常，而是返回nil。

```
local json = require("cjson.safe")
local str  = [[ {"key:"value"} ]]

local t    = json.decode(str)
if t then
    ngx.say(" --> ", type(t))
end
```

稀疏数组

稀疏数组

请看示例代码（注意data的数组下标）：

```
-- http://www.kyne.com.au/~mark/software/lua-cjson.php
-- version: 2.1 devel

local json = require("cjson")

local data = {1, 2}
data[1000] = 99

-- ... do the other things
ngx.say(json.encode(data))
```

运行日志报错结果：

```
2015/06/27 00:23:13 [error] 2714#0: *40 lua entry thread aborted: runtime
 error: ...ork/git/github.com/lua-resty-memcached-server/t/test.lua:13: C
annot serialise table: excessively sparse array
stack traceback:
coroutine 0:
    [C]: in function 'encode'
    ...ork/git/github.com/lua-resty-memcached-server/t/test.lua:13: in fu
nction <...ork/git/github.com/lua-resty-memcached-server/t/test.lua:1>, c
lient: 127.0.0.1, server: localhost, request: "GET /test HTTP/1.1", host:
 "127.0.0.1:8001"
```

如果把data的数组下标修改成5，那么这个json.encode就会是成功的。结果是：[1,2,null,null,99]

为什么下标是1000就失败呢？实际上这么做是cjson想保护你的内存资源。她担心这个下标过大直接撑爆内存（贴心小棉袄啊）。如果我们一定要让这种情况下可以decode，就要尝试encode_sparse_array api了。有兴趣的同学可以自己试一试。我相信你看过有关cjson的代码后，就知道cjson的一个简单危险防范应该是怎样完成的。

空table编码为array还是object

编码为array还是object

首先大家请看这段源码：

```
-- http://www.kyne.com.au/~mark/software/lua-cjson.php
-- version: 2.1 devel

local json = require("cjson")
ngx.say("value --> ", json.encode({dogs={}}))
```

输出结果

> value --> {"dogs":{}}

注意看下encode后key的值类型，"{}" 代表key的值是个object，"[]" 则代表key的值是个数组。对于强类型语言(c/c++, java等)，这时候就有点不爽。因为类型不是他期望的要做容错。对于lua本身，是把数组和字典融合到一起了，所以他是无法区分空数组和空字典的。

参考openresty-cjson中额外贴出测试案例，我们就很容易找到思路了。

```
-- 内容节选lua-cjson-2.1.0.2/tests/agentzh.t
=== TEST 1: empty tables as objects
--- lua
local cjson = require "cjson"
print(cjson.encode({}))
print(cjson.encode({dogs = {}}))
--- out
{}
{"dogs":{}}

=== TEST 2: empty tables as arrays
--- lua
local cjson = require "cjson"
cjson.encode_empty_table_as_object(false)
print(cjson.encode({}))
print(cjson.encode({dogs = {}}))
--- out
[]
{"dogs":[]}
```

综合本章节提到的各种问题，我们可以封装一个json encode的示例函数：

```
function json_encode( data, empty_table_as_object )
  --lua的数据类型里面，array和dict是同一个东西。对应到json encode的时候，就会有不同的判断
  --对于linux，我们用的是cjson库：A Lua table with only positive integer keys of type number will be encoded as a JSON array. All other tables will be encoded as a JSON object.
  --cjson对于空的table，就会被处理为object，也就是{}
  --dkjson默认对空table会处理为array，也就是[]
  --处理方法：对于cjson，使用encode_empty_table_as_object这个方法。文档里面没有，看源码
  --对于dkjson，需要设置meta信息。local a= {}；a.s = {};a.b='中文';setmetatable(a.s,  { __jsontype = 'object' });ngx.say(comm.json_encode(a))

    local json_value = nil
    if json.encode_empty_table_as_object then
        json.encode_empty_table_as_object(empty_table_as_object or false)
 -- 空的table默认为array
    end
    if require("ffi").os ~= "Windows" then
        json.encode_sparse_array(true)
    end
    pcall(function (data) json_value = json.encode(data) end, data)
    return json_value
end
```

跨平台的库选择

跨平台的库选择

大家看过上面三个json的例子就发现，都是围绕cjson库的。原因也比较简单，就是cjson是默认绑定到openresty上的。所以在linux环境下我们也默认的使用了他。在360天擎项目中，linux用户只是很少量的一部分。大部分用户更多的是windows操作系统，但cjson目前还没有windows版本。所以对于windows用户，我们只能选择dkjson（编解码效率没有cjson快，优势是纯lua，完美跨任何平台）。

并且我们的代码肯定不会因为win、linux的并存而写两套程序。那么我们就必须要把json处理部分封装一下，隐藏系统差异造成的差异化处理。

```
local _M = { _VERSION = '1.0' }
-- require("ffi").os 获取系统类型
local json = require(require("ffi").os == "Windows" and "dkjson" or "cjso
n")

function _M.json_decode( str )
    return json.decode(str)
end
function _M.json_encode( data )
    return json.encode(data)
end

return _M
```

在我们的应用中，对于操作系统版本差异、操作系统位数差异、同时支持不通数据库使用等，几乎都是使用这个方法完成的，十分值得推荐。

额外说个点，github上有个项目cloudflare/lua-resty-json，从官方资料上介绍decode的速度更快，我们也做了小范围应用。所以上面的json_decode对象来源，就可以改成这个库。

外面总是有新鲜玩意，多抬头多发现，充实自己，站在巨人肩膀上，总是能够更容易够到高峰。

PostgresNginxModule

PostgresNginxModule

PostgreSQL 是加州大学博客利分校计算机系开发的对象关系型数据库管理系统(ORDBMS)，目前是免费开源的，且是全功能的自由软件数据库。PostgreSQL 支持大部分 SQL 标准，其特性覆盖了 SQL-2/SQL-92 和 SQL-3/SQL-99，并且提供了许多其他现代特点，如复杂查询、外键、触发器、视图、事务完整性、多版本并行控制系统（MVCC）等。PostgreSQL 可以使用许多方法扩展，比如，通过增加新的数据类型、函数、操作符、聚集函数、索引方法、过程语言等。

PostgreSQL 在灵活的 BSD 风格许可证下发行，任何人都可以根据自己的需要免费使用、修改和分发 PostgreSQL，不管是用于私人、商业、还是学术研究。

在360企业安全产品中，PostgreSQL 作为关系型数据库基础组件使用，大量的企业安全信息均分解为若干个对象和关系表存储于 PostgreSQL，Openresty 使用 ngx_postgres 模块，与 PostgreSQL 通讯。

ngx_postgres 是一个提供 nginx 与 PostgreSQL 直接通讯的 upstream 模块。应答数据采用了 rds 格式,所以模块与 ngx_rds_json 和 ngx_drizzle 模块是兼容的。

调用方式简介

PostgresNginxModule模块的调用方式

ngx_postgres模块使用方法

```
location /postgres {
    internal;

    default_type text/html;
    set_by_lua $query_sql 'return ngx.unescape_uri(ngx.var.arg_sql)';

    postgres_pass   pg_server;
    rds_json          on;
    rds_json_buffer_size 16k;
    postgres_query  $query_sql;
    postgres_connect_timeout 1s;
    postgres_result_timeout 2s;
}
```

这里有很多指令要素：

- internal 这个指令指定所在的 location 只允许使用于处理内部请求，否则返回 404 。
- set_by_lua 这一段内嵌的 lua 代码用于计算出 $query_sql 变量的值，即后续通过指令 postgres_query 发送给 PostgreSQL 处理的 SQL 语句。这里使用了GET请求的 query 参数作为 SQL 语句输入。
- postgres_pass 这个指令可以指定一组提供后台服务的 PostgreSQL 数据库的 upstream 块。
- rds_json 这个指令是 ngx_rds_json 提供的，用于指定 ngx_rds_json 的 output 过滤器的开关状态，其模块作用就是一个用于把 rds 格式数据转换成 json 格式的 output filter。这个指令在这里出现意思是让 ngx_rds_json 模块帮助 ngx_postgres 模块把模块输出数据转换成 json 格式的数据。
- rds_json_buffer_size 这个指令指定 ngx_rds_json 用于每个连接的数据转换的内存大小. 默认是 4/8k,适当加大此参数，有利于减少 CPU 消耗。
- postgres_query 指定 SQL 查询语句，查询语句将会直接发送给 PostgreSQL 数据库。
- postgres_connect_timeout 设置连接超时时间。
- postgres_result_timeout 设置结果返回超时时间。

这样的配置就完成了初步的可以提供其他 location 调用的 location 了。但这里还差一个配置没说明白，就是这一行：

```
postgres_pass   pg_server;
```

其实这一行引入了 名叫 pg_server 的 upstream 块，其定义应该像如下：

```
upstream pg_server {
    postgres_server  192.168.1.2:5432 dbname=pg_database
            user=postgres password=postgres;
    postgres_keepalive max=800 mode=single overflow=reject;
}
```

这里有一些指令要素：

- postgres_server 这个指令是必须带的，但可以配置多个，用于配置服务器连接参数，可以分解成若干参数：
- 直接跟在后面的应该是服务器的 IP:Port
- dbname 是服务器要连接的 PostgreSQL 的数据库名称。
- user 是用于连接 PostgreSQL 服务器的账号名称。
- password 是账号名称对应的密码。
- postgres_keepalive 这个指令用于配置长连接连接池参数，长连接连接池有利于提高通讯效率，可以分解为若干参数：
- max 是工作进程可以维护的连接池最大长连接数量。
- mode 是后端匹配模式，在postgres_server 配置了多个的时候发挥作用，有 single 和 multi 两种值，一般使用 single 即可。
- overflow 是当长连接数量到达 max 之后的处理方案，有 ignore 和 reject 两种值。
- ignore 允许创建新的连接与数据库通信，但完成通信后马上关闭此连接。
- reject 拒绝访问并返回 503 Service Unavailable

这样就构成了我们 PostgreSQL 后端通讯的通用 location，在使用 lua 业务编码的过程中可以直接使用如下代码连接数据库（折腾了这么老半天）：

```
local json = require "cjson"

function test()
    local res = ngx.location.capture('/postgres',
```

```
        { args = {sql = "SELECT * FROM test" } }
    )

    local status = res.status
    local body = json.decode(res.body)

    if status == 200 then
        status = true
    else
        status = false
    end
    return status, body
end
```

与resty-mysql调用方式的不同

先来看一下 `lua-resty-mysql` 模块的调用示例代码。

```
# you do not need the following line if you are using
# the ngx_openresty bundle:
lua_package_path "/path/to/lua-resty-mysql/lib/?.lua;;";

server {
    location /test {
        content_by_lua '
            local mysql = require "resty.mysql"
            local db, err = mysql:new()
            if not db then
                ngx.say("failed to instantiate mysql: ", err)
                return
            end

            db:set_timeout(1000) -- 1 sec

            local ok, err, errno, sqlstate = db:connect{
                host = "127.0.0.1",
                port = 3306,
                database = "ngx_test",
                user = "ngx_test",
                password = "ngx_test",
                max_packet_size = 1024 * 1024 }

            if not ok then
                ngx.say("failed to connect: ", err, ": ", errno, " ", sql
state)
                return
            end

            ngx.say("connected to mysql.")

            -- run a select query, expected about 10 rows in
            -- the result set:
            res, err, errno, sqlstate =
                db:query("select * from cats order by id asc", 10)
```

```
            if not res then
                ngx.say("bad result: ", err, ": ", errno, ": ", sqlstate,
 ".")
                return
            end

            local cjson = require "cjson"
            ngx.say("result: ", cjson.encode(res))

            -- put it into the connection pool of size 100,
            -- with 10 seconds max idle timeout
            local ok, err = db:set_keepalive(10000, 100)
            if not ok then
                ngx.say("failed to set keepalive: ", err)
                return
            end
        ';
    }
}
```

看过这段代码，大家肯定会说：这才是我熟悉的，我想要的。为什么刚刚 `ngx_postgres` 模块的调用这么诡异，配置那么复杂，其实这是发展历史造成的。 `ngx_postgres` 起步比较早，当时 `OpenResty` 也还没开始流行，所以更多的 Nginx 数据库都是以 ngx_c_module 方式存在。有了 `OpenResty` ，才让我们具有了使用完整的语言来描述我们业务能力。

后面我们会单独说一说使用 `ngx_c_module` 的各种不方便，也就是我们所踩过的坑。希望能给大家一个警示，能转到 `lua-resty-***` 这个方向的，就千万不要和 `ngx_c_module` 玩， `ngx_c_module` 的扩展性、可维护性、升级等各方面都没有 `lua-resty-***` 好。

这绝对是经验的总结。不服来辩！

不支持事务

不支持事务

我们继续上一章节的内容，大家应该记得我们 `Lua` 代码中是如何完成 `ngx_postgres` 模块调用的。我们把他简单改造一下，让他更接近真实代码。

```
    local json = require "cjson"

    function db_exec(sql_str)
        local res = ngx.location.capture('/postgres',
            { args = {sql = sql_str } }
        )

        local status = res.status
        local body = json.decode(res.body)

        if status == 200 then
            status = true
        else
            status = false
        end
        return status, body
    end

    -- 转账操作，对ID=100的用户加10，同时对ID=200的用户减10。
?    local status
?    status = db_exec("BEGIN")
?    if status then
?        db_exec("ROLLBACK")
?    end
?
?    status = db_exec("UPDATE ACCOUNT SET MONEY=MONEY+10  WHERE ID = 100"
)
?    if status then
?        db_exec("ROLLBACK")
?    end
?
?    status = db_exec("UPDATE ACCOUNT SET MONEY=MONEY-10  WHERE ID = 200"
)
?    if status then
?        db_exec("ROLLBACK")
?    end
?
?    db_exec("COMMIT")
```

后面这部分有问题的代码，在没有并发的场景下使用，是不会有任何问题的。但是这段代码在高并发应用场景下，错误百出。你会发现最后执行结果完全摸不清楚。明明是个转账逻

辑，一个收入，一直支出，最后却发现总收入比支出要大。如果这个错误发生在金融领域，那不知道要赔多少钱。

如果你能靠自己很快明白错误的原因，那么恭喜你你对数据库连接、`Nginx`机理都是比较清楚的。如果你想不明白，那就听我给你掰一掰这面的小知识。

数据库的事物成功执行，事物相关的所有操作是必须执行在一条连接上的。`SQL`的执行情况类似这样：

```
连接：`BEGIN` -> `SQL(UPDATE、DELETE... ...)` -> `COMMIT`。
```

但如果你创建了两条连接，每条连接提交的SQL语句是下面这样：

```
连接1：`BEGIN` -> `SQL(UPDATE、DELETE... ...)`
连接2：`COMMIT`
```

这时就会出现连接1的内容没有被提交，行锁产生。连接2提交了一个空的`COMMIT`。

说到这里你可能开始鄙视我了，谁疯了非要创建两条连接来这么用SQL啊。有麻烦，又不好看，貌似从来没听说过还有人在一次请求中创建多个数据库连接，简直就是非人类嘛。

或许你不会主动、显示的创建多个连接，但是刚刚的示例代码，高并发下这个事物的每个SQL语句都可能落在不同的连接上。为什么呢？这是因为通过`ngx.location.capture`跳转到`/postgres`小节后，`Nginx`每次都会从连接池中挑选一条空闲连接，二当时那条连接是空闲的，完全没法预估。所以上面的第二个例子，就这么静悄悄的发生了。如果你不了解`Nginx`的机理，那么他肯定会一直困扰你。为什么一会儿好，一会儿不好。

同样的道理，我们推理到`DrizzleNginxModule`、`RedisNginxModule`、`Redis2NginxModule`，他们都是无法做到在两次连续请求落到同一个连接上的。

由于这个`Bug`藏得比较深，并且不太好讲解，所以我觉得生产中最好用`lua-resty-*`这类的库，更符合标准调用习惯，直接可以绕过这些坑。不要为了一点点的性能，牺牲了更大的蛋糕。看得见的，看不见的，都要了解用用，最后再做决定，肯定不吃亏。

不支持事务

超时

超时

健康监测

健康监测

SQL注入

SQL注入

有使用 SQL 语句操作数据库的经验朋友，应该都知道使用 SQL 过程中有一个安全问题叫 SQL 注入。所谓 SQL 注入，就是通过把 SQL 命令插入到 Web 表单提交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的 SQL 命令。为了防止 SQL 注入，在生产环境中使用 Openresty 的时候就要注意添加防范代码。

延续之前的 ngx_postgres 调用代码的使用，

```
    local sql_normal = [[select id, name from user where name=']] .. ngx.
var.arg_name .. [[' and password=']] .. ngx.var.arg_password .. [[' limit
 1;]]

    local res = ngx.location.capture('/postgres',
        { args = {sql = sql } }
    )

    local body = json.decode(res.body)

    if (table.getn(res) > 0) {
        return res[1];
    }

    return nil;
```

假设我们在用户登录使用上 SQL 语句查询账号是否账号密码正确，用户可以通过 GET 方式请求并发送登录信息比如：

```
    http://localhost/login?name=person&password=12345
```

那么我们上面的代码通过 ngx.var.arg_name 和 ngx.var.arg_password 获取查询参数，并且与 SQL 语句格式进行字符串拼接，最终 sql_normal 会是这个样子的：

```
    local sql_normal = [[select id, name from user where name='person' an
d password='12345' limit 1;]]
```

正常情况下，如果 person 账号存在并且 password 是 12345，那么sql执行结果就应该是能返回id号的。这个接口如果暴露在攻击者面前，那么攻击者很可能会让参数这样传入：

```
    name="' or ''='"
    password="' or ''='"
```

那么这个 sql_normal 就会变成一个永远都能执行成功的语句了。

```
    local sql_normal = [[select id, name from user where name='' or ''=''
 and password='' or ''='' limit 1;]]
```

这就是一个简单的 sql inject （注入）的案例，那么问题来了，面对这么凶猛的攻击者，我们有什么办法防止这种 SQL 注入呢？

很简单，我们只要 把 传入参数的变量 做一次字符转义，把不该作为破坏SQL查询语句结构的双引号或者单引号等做转义，把 ' 转义成 \'，那么变量 name 和 password 的内容还是乖乖的作为查询条件传入，他们再也不能为非作歹了。

那么怎么做到字符转义呢？要知道每个数据库支持的SQL语句格式都不太一样啊，尤其是双引号和单引号的应用上。有几个选择：

```
    ndk.set_var.set_quote_sql_str()
    ndk.set_var.set_quote_pgsql_str()
    ngx.quote_sql_str()
```

这三个函数，前面两个是 ndk.set_var 跳转调用，其实是 HttpSetMiscModule 这个模块提供的函数，是一个 C 模块实现的函数，ndk.set_var.set_quote_sql_str() 是用于 MySQL 格式的 SQL 语句字符转义，而 set_quote_pgsql_str 是用于 PostgreSQL 格式的 SQL 语句字符转义。最后 ngx.quote_sql_str 是一个 ngx_lua 模块中实现的函数，也是用于 MySQL 格式的 SQL 语句字符转义。

让我们看看代码怎么写：

```
    local name = ngx.quote_sql_str(ngx.var.arg_name)
    local password = ngx.quote_sql_str(ngx.var.arg_password)
    local sql_normal = [[select id, name from user where name=]] .. name
 .. [[ and password=]] .. password .. [[ limit 1;]]

    local res = ngx.location.capture('/postgres',
        { args = {sql = sql } }
    )

    local body = json.decode(res.body)
```

```
    if (table.getn(res) > 0) {
        return res[1];
    }

    return nil;
```

注意上述代码有两个变化：

```
* 用 ngx.quote_sql_str 把 ngx.var.arg_name 和 ngx.var.arg_password 包了一层
, 把返回值作为 sql 语句拼凑起来。
* 原本在 sql 语句中添加的单引号去掉了, 因为 ngx.quote_sql_str 的返回值正确的带上引
号了。
```

这样子已经可以抵御 SQL 注入的攻击手段了，但开发过程中需要不断的产生新功能新代码，这时候也一定注意不要忽视 SQL 注入的防护，安全防御代码就想织网一样，只要有一处漏洞，鱼儿可就游走了。

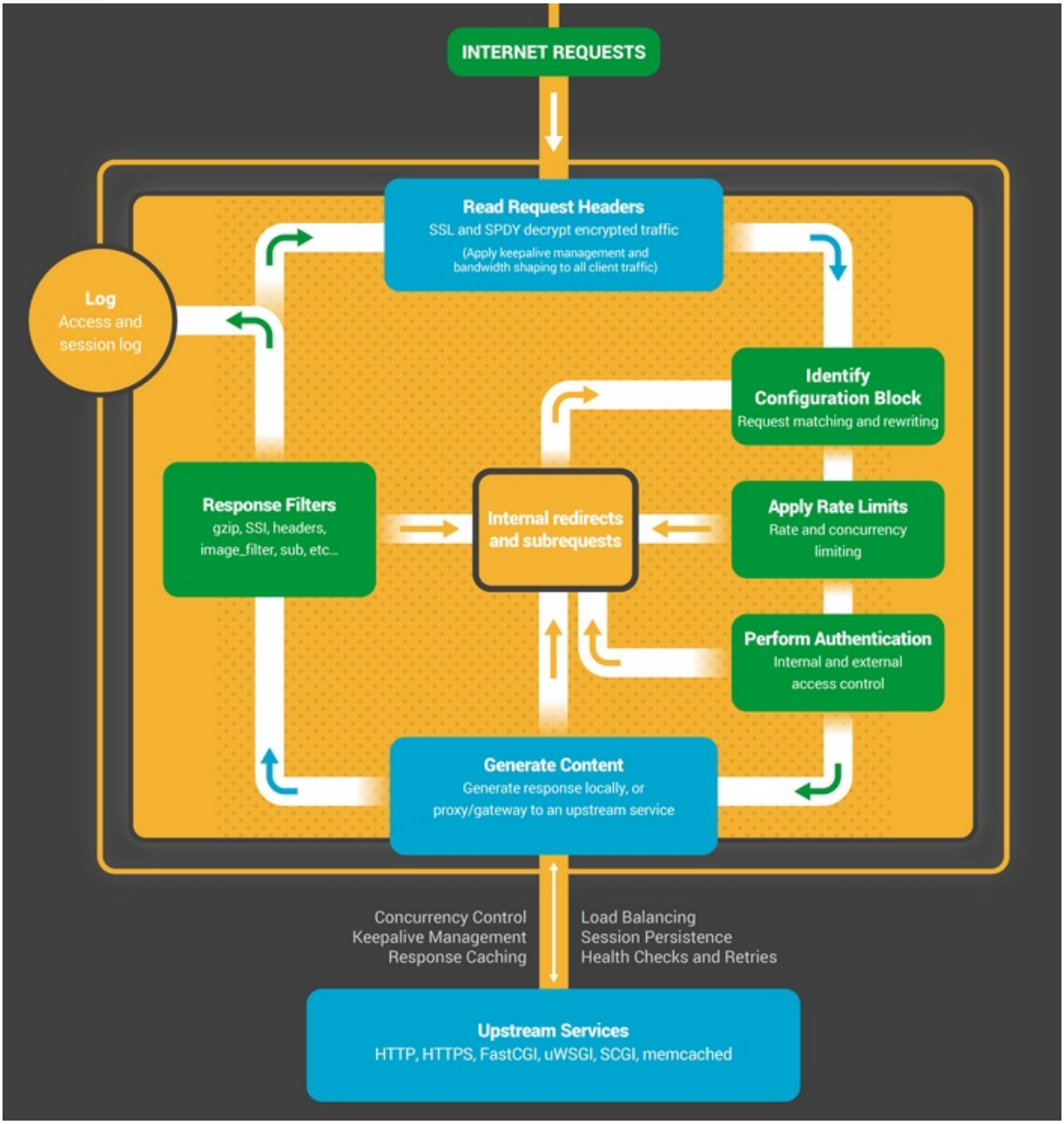
LuaNginxModule

LuaNginxModule

执行阶段概念

执行阶段概念

`Nginx` 处理一个请求，它的处理流程请参考下图：



我们OpenResty做个测试，示例代码如下：

```
location /mixed {
    set_by_lua $a 'ngx.log(ngx.ERR, "set_by_lua")';
    rewrite_by_lua 'ngx.log(ngx.ERR, "rewrite_by_lua")';
    access_by_lua 'ngx.log(ngx.ERR, "access_by_lua")';
    header_filter_by_lua 'ngx.log(ngx.ERR, "header_filter_by_lua")';
    body_filter_by_lua 'ngx.log(ngx.ERR, "body_filter_by_lua")';
    log_by_lua 'ngx.log(ngx.ERR, "log_by_lua")';
    content_by_lua 'ngx.log(ngx.ERR, "content_by_lua")';
}
```

执行结果日志(截取了一下)：

```
set_by_lua
rewrite_by_lua
access_by_lua
content_by_lua
header_filter_by_lua
body_filter_by_lua
log_by_lua
```

这几个阶段的存在，应该是openresty不同于其他多数web server编程的最明显特征了。由于nginx把一个会话分成了很多阶段，这样第三方模块就可以根据自己行为，挂载到不同阶段进行处理达到目的。

这样我们就可以根据我们的需要，在不同的阶段直接完成大部分典型处理了。

- set_by_lua: 流程分之处理判断变量初始化
- rewrite_by_lua: 转发、重定向、缓存等功能(例如特定请求代理到外网)
- access_by_lua: IP准入、接口权限等情况集中处理(例如配合iptable完成简单防火墙)
- content_by_lua: 内容生成
- header_filter_by_lua: 应答HTTP过滤处理(例如添加头部信息)
- body_filter_by_lua: 应答BODY过滤处理(例如完成应答内容统一成大写)
- log_by_lua: 回话完成后本地异步完成日志记录(日志可以记录在本地，还可以同步到其他机器)

实际上我们只使用其中一个阶段content_by_lua，也可以完成所有的处理。但这样做，会让我们的代码比较臃肿，越到后期越发难以维护。把我们的逻辑放在不同阶段，分工明确，代码独立，后期发力可以有很多有意思的玩法。

列举360企业版的一个例子：

```
# 明文协议版本
location /mixed {
    content_by_lua '...';        # 请求处理
}

# 加密协议版本
location /mixed {
    access_by_lua '...';         # 请求加密解码
    content_by_lua '...';        # 请求处理，不需要关心通信协议
    body_filter_by_lua '...';    # 应答加密编码
}
```

内容处理部分都是在content_by_lua阶段完成，第一版本API接口开发都是基于明文。为了传输体积、安全等要求，我们设计了支持压缩、加密的密文协议(上下行)，痛点就来了，我们要更改所有API的入口、出口么？

最后我们是在access_by_lua完成密文协议解码，body_filter_by_lua完成应答加密编码。如此一来世界都宁静了，我们没有更改已实现功能的一行代码，只是利用ngx-lua的阶段处理特性，非常优雅的解决了这个问题。

前两天看到春哥的微博，里面说到github的某个应用里面也使用了openresty做了一些东西。发现他们也是利用阶段特性+lua脚本处理了很多用户证书方面的东东。最终在性能、稳定性都十分让人满意。使用者选型很准，不愧是github的工程师。

不同的阶段，有不同的处理行为，这是openresty的一大特色。学会他，适应他，会给你打开新的一扇门。这些东西不是openresty自身所创，而是nginx c module对外开放的处理阶段。理解了他，也能更好的理解nginx的设计思维。

正确的记录日志

正确的记录日志

看过本章第一节的同学应该还记得，log_by_lua是一个会话阶段最后发生的，文件操作是阻塞的（FreeBSD直接无视），nginx为了实时高效的给请求方应答后，日志记录是在应答后异步记录完成的。由此可见如果我们有日志输出的情况，最好统一到log_by_lua阶段。如果我们自定义放在content_by_lua阶段，那么将线性的增加请求处理时间。

在公司某个定制化项目中，nginx上的日志内容都要输送到syslog日志服务器。我们使用了lua-resty-logger-socket这个库。

> 调用示例代码如下（有问题的）：

```
-- lua_package_path "/path/to/lua-resty-logger-socket/lib/?.lua;;";
--
--    server {
--        location / {
--            content_by_lua lua/log.lua;
--        }
--    }

-- lua/log.lua
local logger = require "resty.logger.socket"
if not logger.initted() then
    local ok, err = logger.init{
        host = 'xxx',
        port = 1234,
        flush_limit = 1,   --日志长度大于flush_limit的时候会将msg信息推送一次
        drop_limit = 99999,
    }
    if not ok then
        ngx.log(ngx.ERR, "failed to initialize the logger: ",err)
        return
    end
end

local msg = string.format(.....)
local bytes, err = logger.log(msg)
if err then
    ngx.log(ngx.ERR, "failed to log message: ", err)
    return
end
```

在实测过程中我们发现了些问题：

- 缓存无效：如果flush_limit的值稍大一些（例如 2000），会导致某些体积比较小的日志出现莫名其妙的丢失，所以我们只能把flush_limit调整的很小
- 自己拼写msg所有内容，比较辛苦

那么我们来看lua-resty-logger-socket这个库的log函数是如何实现的呢，代码如下：

```
function _M.log(msg)
   ...

    if (debug) then
        ngx.update_time()
        ngx_log(DEBUG, ngx.now(), ":log message length: " .. #msg)
    end

    local msg_len = #msg

    if (is_exiting()) then
        exiting = true
        _write_buffer(msg)
        _flush_buffer()
        if (debug) then
            ngx_log(DEBUG, "Nginx worker is exiting")
        end
        bytes = 0
    elseif (msg_len + buffer_size < flush_limit) then  -- 历史日志大小+本地
日志大小小于推送上限
        _write_buffer(msg)
        bytes = msg_len
    elseif (msg_len + buffer_size <= drop_limit) then
        _write_buffer(msg)
        _flush_buffer()
        bytes = msg_len
    else
        _flush_buffer()
        if (debug) then
            ngx_log(DEBUG, "logger buffer is full, this log message will
be "
                    .. "dropped")
        end
        bytes = 0
        --- this log message doesn't fit in buffer, drop it

        ...
```

由于在content_by_lua阶段变量的生命周期会随着会话的终结而终结，所以当日志量小于flush_limit的情况下这些日志就不能被累积，也不会触发_flush_buffer函数，所以小日志会丢失。

这些坑回头看来这么明显，所有的问题都是因为我们把lua/log.lua用错阶段了，应该放到

log_by_lua阶段，所有的问题都不复存在。

> 修正后：

```
lua_package_path "/path/to/lua-resty-logger-socket/lib/?.lua;;";

    server {
        location / {
            content_by_lua lua/content.lua;
            log_by_lua lua/log.lua;
        }
    }
```

这里有个新问题，如果我的log里面需要输出一些content的临时变量，两阶段之间如何传递参数呢？

> 方法肯定有，推荐下面这个：

```
    location /test {
        rewrite_by_lua '
            ngx.say("foo = ", ngx.ctx.foo)
            ngx.ctx.foo = 76
        ';
        access_by_lua '
            ngx.ctx.foo = ngx.ctx.foo + 3
        ';
        content_by_lua '
            ngx.say(ngx.ctx.foo)
        ';
    }
```

更多有关ngx.ctx信息，请看这里。

热装载代码

热装载代码

在Openresty中，提及热加载代码，估计大家的第一反应是lua_code_cache这个开关。在开发阶段我们把它配置成lua_code_cache off，是很方便、有必要的，修改完代码，肯定都希望自动加载最新的代码（否则我们就要噩梦般的reload服务，然后再测试脚本）。

禁用 Lua 代码缓存（即配置 lua_code_cache off）只是为了开发便利，一般不应以高于 1 并发来访问，否则可能会有race condition等等问题。同时因为它会有带来严重的性能衰退，所以不应在生产上使用此种模式。生产上应当总是启用Lua代码缓存，即配置 lua_code_cache on。

那么我们是否可以在生产环境中完成热加载呢？

- 代码有变动时，自动加载最新lua代码，但是nginx本身，不做任何reload
- 自动加载后的代码，享用lua_code_cache on带来的高效特性

这里有多种玩法（引自Openresty讨论组）：

- 使用 HUP reload 或者 binary upgrade 方式动态加载 nginx 配置或重启 nginx。这不会导致中间有请求被 drop 掉。
- 当 content_by_lua_file 里使用 nginx 变量时，是可以动态加载新的 Lua 脚本的，不过要记得对 nginx 变量的值进行基本的合法性验证，以免被注入攻击。

```
    location ~ '^/lua/(\w+(?:\/\w+)*)$' {
        content_by_lua_file $1;
    }
```

- 自己从外部数据源（包括文件系统）加载 Lua 源码或字节码，然后使用 loadstring() “eval”进 Lua VM. 可以通过 package.loaded 自己来做缓存，毕竟频繁地加载源码和调用 loadstring()，以及频繁地 JIT 编译还是很昂贵的（类似 lua_code_cache off 的情形）。比如在 CloudFlare 我们从 modsecurity 规则编译出来的 Lua 代码就是通过 KyotoTycoon 动态分发到全球网络中的每一个 nginx 服务器的。无需 reload 或者 binary upgrade.

自定义module的动态装载

对于已经装载的module，我们可以通过package.loaded.* = nil的方式卸载。

不过，值得提醒的是，因为 require 这个内建函数在标准 Lua 5.1 解释器和 LuaJIT 2 中都被实现为 C 函数，所以你在自己的 loader 里可能并不能调用 ngx_lua 那些涉及非阻塞 IO 的 Lua 函数。因为这些 Lua 函数需要 yield 当前的 Lua 协程，而 yield 是无法跨越 Lua 调用栈上的 C 函数帧的。细节见

https://github.com/openresty/lua-nginx-module#lua-coroutine-yieldingresuming

所以直接操纵 package.loaded 是最简单和最有效的做法。我们在 CloudFlare 的 Lua WAF 系统中就是这么做的。

不过，值得提醒的是，从 package.loaded 解注册的 Lua 模块会被 GC 掉。而那些使用下列某一个或某几个特性的 Lua 模块是不能被安全的解注册的：

- 使用 FFI 加载了外部动态库
- 使用 FFI 定义了新的 C 类型
- 使用 FFI 定义了新的 C 函数原型

这个限制对于所有的 Lua 上下文都是适用的。

这样的 Lua 模块应避免手动从 package.loaded 卸载。当然，如果你永不手工卸载这样的模块，只是动态加载的话，倒也无所谓了。但在我们的 Lua WAF 的场景，已动态加载的一些 Lua 模块还需要被热替换掉（但不重新创建 Lua VM）。

自定义lua script的动态装载实现

> 引自Openresty讨论组

一方面使用自定义的环境表 [1]，以白名单的形式提供用户脚本能访问的 API；另一方面，（只）为用户脚本禁用 JIT 编译，同时使用 Lua 的 debug hooks [2] 作脚本 CPU 超时保护（debug hooks 对于 JIT 编译的代码是不会执行的，主要是出于性能方面的考虑）。

下面这个小例子演示了这种玩法：

```
local user_script = [[
    local a = 0
    local rand = math.random
    for i = 1, 200 do
        a = a + rand(i)
    end
```

```
    ngx.say("hi")
]]

local function handle_timeout(typ)
    return error("user script too hot")
end

local function handle_error(err)
    return string.format("%s: %s", err or "", debug.traceback())
end

-- disable JIT in the user script to ensure debug hooks always work:
user_script = [[jit.off(true, true) ]] .. user_script

local f, err = loadstring(user_script, "=user script")
if not f then
    ngx.say("ERROR: failed to load user script: ", err)
    return
end

-- only enable math.*, and ngx.say in our sandbox:
local env = {
    math = math,
    ngx = { say = ngx.say },
    jit = { off = jit.off },
}
setfenv(f, env)

local instruction_limit = 1000
debug.sethook(handle_timeout, "", instruction_limit)
local ok, err = xpcall(f, handle_error)
if not ok then
    ngx.say("failed to run user script: ", err)
end
debug.sethook()  -- turn off the hooks
```

这个例子中我们只允许用户脚本调用 math 模块的所有函数、ngx.say() 以及 jit.off(). 其中 jit.off()是必需引用的，为的是在用户脚本内部禁用 JIT 编译，否则我们注册的 debug hooks 可能不会被调用。

另外，这个例子中我们设置了脚本最多只能执行 1000 条 VM 指令。你可以根据你自己的场景进行调整。

这里很重要的是，不能向用户脚本暴露 pcall 和 xpcall 这两个 Lua 指令，否则恶意用户会利用它故意拦截掉我们在 debug hook 里为中断脚本执行而抛出的 Lua 异常。

另外，require()、loadstring()、loadfile()、dofile()、io.、os. 等等 API 是一定不能暴露给不被信任的 Lua 脚本的。

阻塞操作

阻塞操作

Openresty的诞生，一直对外宣传是非阻塞(100% noblock)的。基于事件通知的Nginx给我们带来了足够强悍的高并发支持，但是也对我们的编码有特殊要求。这个特殊要求就是我们的代码，也必须是非阻塞的。如果你的服务端编程生涯一开始就是从异步框架开始的，恭喜你了。但如果你的编程生涯是从同步框架过来的，而且又是刚刚开始深入了解异步框架，那你就要小心了。

Nginx为了减少系统上下文切换，它的worker是用单进程单线程设计的，事实证明这种做法运行效率很高。Nginx要么是在等待网络讯号，要么就是在处理业务（请求数据解析、过滤、内容应答等），没有任何额外资源消耗。

常见语言代表异步框架

- Golang ：使用协程技术实现
- Python ：gevent基于协程的Python网络库
- Rust ：用的少，只知道语言完备支持异步框架
- Openresty：基于Nginx，使用事件通知机制
- Java ：Netty，使用网络事件通知机制

异步编程的噩梦

异步编程，如果从零开始，难度是非常大的。一个完整的请求，由于网络传输的非连续性，这个请求要被多次挂起、恢复、运行，一旦网络有新数据到达，都需要立刻唤醒恢复原始请求处于运行状态。开发人员不仅仅要考虑异步api接口本身的使用规范，还要考虑业务会话的完整处理，稍有不慎，全盘皆输。

最最重要的噩梦是，我们好不容易搞定异步框架和业务会话完整性，但是却在我们的业务会话上使用了阻塞函数。一开始没有任何感知，只有做压力测试的时候才发现我们的并发量上不去，各种卡曼顿，甚至开始怀疑人生：异步世界也就这样。

Openresty中的阻塞函数

官方有明确说明，Openresty的官方API绝对100% noblock，所以我们只能在她的外面寻找了。我这里大致归纳总结了一下，包含下面几种情况：

- 高CPU的调用（压缩、解压缩、加解密等）
- 高磁盘的调用（所有文件操作）

- 非Openresty提供的网络操作（luasocket等）
- 系统命令行调用（os.execute等）

这些都应该是我们尽量要避免的。理想丰满，现实骨感，谁能保证我们的应用中不使用这些类型的API？没人保证，我们能做的就是把他们的调用数量、频率降低再降低，如果还是不能满足我们需要，那么就考虑把他们封装成独立服务，对外提供TCP/HTTP级别的接口调用，这样我们的Openresty就可以同时享受异步编程的好处又能达到我们的目的。

缓存

缓存

缓存的原则

缓存是一个大型系统中非常重要的一个组成部分。在硬件层面，大部分的计算机硬件都会用缓存来提高速度，比如CPU会有多级缓存、RAID卡也有读写缓存。在软件层面，我们用的数据库就是一个缓存设计非常好的例子，在SQL语句的优化、索引设计、磁盘读写的各个地方，都有缓存，建议大家在设计自己的缓存之前，先去了解下MySQL里面的各种缓存机制，感兴趣的可以去看下High Permance MySQL这本非常有价值的书。

一个生产环境的缓存系统，需要根据自己的业务场景和系统瓶颈，来找出最好的方案，这是一门平衡的艺术。

一般来说，缓存有两个原则。一是越靠近用户的请求越好，比如能用本地缓存的就不要发送HTTP请求，能用CDN缓存的就不要打到web服务器，能用nginx缓存的就不要用数据库的缓存；二是尽量使用本进程和本机的缓存解决，因为跨了进程和机器甚至机房，缓存的网络开销就会非常大，在高并发的时候会非常明显。

OPenResty的缓存

我们介绍下在OpenResty里面，有哪些缓存的方法。

使用lua shared dict

我们看下面这段代码：

```
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
```

这里面用的就是ngx shared dict cache。你可能会奇怪，ngx.shared.my_cache是从哪里冒出来的？没错，少贴了nginx.conf里面的修改：

```
lua_shared_dict my_cache 128m;
```

如同它的名字一样，这个cache是nginx所有worker之间共享的，内部使用的LRU算法（最近经常使用）来判断缓存是否在内存占满时被清除。

使用lua LRU cache
直接复制下春哥的示例代码：

```
local _M = {}

-- alternatively: local lrucache = require "resty.lrucache.pureffi"
local lrucache = require "resty.lrucache"

-- we need to initialize the cache on the lua module level so that
-- it can be shared by all the requests served by each nginx worker proce
ss:
local c = lrucache.new(200)  -- allow up to 200 items in the cache
if not c then
    return error("failed to create the cache: " .. (err or "unknown"))
end

function _M.go()
    c:set("dog", 32)
    c:set("cat", 56)
    ngx.say("dog: ", c:get("dog"))
    ngx.say("cat: ", c:get("cat"))

    c:set("dog", { age = 10 }, 0.1)  -- expire in 0.1 sec
    c:delete("dog")
end

return _M
```

可以看出来，这个cache是worker级别的，不会在nginx wokers之间共享。并且，它是预先分配好key的数量，而shared dcit需要自己用key和value的大小和数量，来估算需要把内存设置为多少。

如何选择？
shared.dict 使用的是共享内存，每次操作都是全局锁，如果高并发环境，不同worker之间容易引起竞争。所以单个shared.dict的体积不能过大。lrucache是worker内使用的，由于nginx是单进程方式存在，所以永远不会触发锁，效率上有优势，并且没有shared.dict的体

积限制，内存上也更弹性，但不同worker之间数据不同享，同一缓存数据可能被冗余存储。

你需要考虑的，一个是lua lru cache提供的API比较少，现在只有get、set和delete，而ngx shared dict还可以add、replace、incr、get_stale（在key过期时也可以返回之前的值）、get_keys（获取所有key，虽然不推荐，但说不定你的业务需要呢）；第二个是内存的占用，由于ngx shared dict是workers之间共享的，所以在多worker的情况下，内存占用比较少。

sleep

sleep

这是一个比较常见的功能，你会怎么做呢？Google一下，你会找到lua的官方指南，

里面介绍了10种sleep不同的方法（操作系统不一样，方法还有区别），选择一个用，然后你就杯具了:(你会发现nginx高并发的特性不见了！

在OpenResty里面选择使用库的时候，有一个基本的原则：尽量使用ngx lua的库函数，尽量不用lua的库函数，因为lua的库都是同步阻塞的。

```
# you do not need the following line if you are using
# the ngx_openresty bundle:
lua_package_path "/path/to/lua-resty-redis/lib/?.lua;;";

server {
    location /non_block {
        content_by_lua '
            ngx.sleep(0.1)
        ';
    }
}
```

本章节内容好少，只是想通过一个真实的例子，来提醒大家，做OpenResty开发，ngx lua的文档是你的首选，lua语言的库都是同步阻塞的，用的时候要三思。

定时任务

定时任务

在请求返回后继续执行章节中，我们介绍了一种实现的方法，这里我们介绍一种更优雅更通用的方法：ngx.timer.at()。这个函数是在后台用nginx轻线程（light thread），在指定的延时后，调用指定的函数。有了这种机制，ngx_lua的功能得到了非常大的扩展，我们有机会做一些更有想象力的功能出来。比如批量提交和cron任务。

需要特别注意的是：有一些ngx lua的API不能在这里调用，比如子请求、ngx.req.*和向下游输出的API(ngx.print、ngx.flush之类)。

禁止某些终端访问

禁止某些终端访问

不同的业务应用场景，会有完全不同的非法终端控制策略，常见的限制策略有终端IP、访问域名端口，这些可以通过防火墙等很多成熟手段完成。可也有一些特定限制策略，例如特定cookie、url、location，甚至请求body包含有特殊内容，这种情况下普通防火墙就比较难限制。

Nginx的是HTTP 7层协议的实现着，相对普通防火墙从通讯协议有自己的弱势，同等的配置下的性能表现绝对远不如防火墙，但它的优势胜在价格便宜、调整方便，还可以完成HTTP协议上一些更具体的控制策略，与iptable的联合使用，让Nginx玩出更多花样。

列举几个限制策略来源

- IP地址
- 域名、端口
- Cookie特定标识
- location
- body中特定标识

> 示例配置（allow、deny）

```
location / {
    deny  192.168.1.1;
    allow 192.168.1.0/24;
    allow 10.1.1.0/16;
    allow 2001:0db8::/32;
    deny  all;
}
```

这些规则都是按照顺序解析执行直到某一条匹配成功。在这里示例中，10.1.1.0/16 and 192.168.1.0/24都是用来限制IPv4的，2001:0db8::/32的配置是用来限制IPv6。具体有关allow、deny配置，请参考这里。

> 示例配置（geo）

```
Example:

geo $country {
    default         ZZ;
    proxy           192.168.100.0/24;

    127.0.0.0/24    US;
    127.0.0.1/32    RU;
    10.1.0.0/16     RU;
    192.168.1.0/24 UK;
}

if ($country == ZZ){
    return 403;
}
```

使用geo，让我们有更多的分之条件。注意：在Nginx的配置中，尽量少用或者不用if，因为"if is evil"。点击查看

目前为止所有的控制，都是用Nginx模块完成，执行效率、配置明确是它的优点。缺点也比较明显，修改配置代价比较高（reload服务）。并且无法完成与第三方服务的对接功能交互（例如调用iptable）。

在Openresty里面，这些问题就都容易解决，还记得access_by_lua么？推荐一个第三方库lua-resty-iputils。

> 示例代码：

```
init_by_lua '
  local iputils = require("resty.iputils")
  iputils.enable_lrucache()
  local whitelist_ips = {
      "127.0.0.1",
      "10.10.10.0/24",
      "192.168.0.0/16",
  }

  -- WARNING: Global variable, recommend this is cached at the module lev
el
  -- https://github.com/openresty/lua-nginx-module#data-sharing-within-an
-nginx-worker
  whitelist = iputils.parse_cidrs(whitelist_ips)
';

access_by_lua '
    local iputils = require("resty.iputils")
    if not iputils.ip_in_cidrs(ngx.var.remote_addr, whitelist) then
```

```
        return ngx.exit(ngx.HTTP_FORBIDDEN)
    end
';
```

以次类推，我们想要完成域名、Cookie、location、特定body的准入控制，甚至可以做到与本地iptable防火墙联动。我们可以把IP规则存到数据库中，这样我们就再也不用reload nginx，在有规则变动的时候，刷新下nginx的缓存就行了。

思路打开，大家后面多尝试各种玩法吧。

请求返回后继续执行

请求返回后继续执行

在一些请求中，我们会做一些日志的推送、用户数据的统计等和返回给终端数据无关的操作。而这些操作，即使你用异步非阻塞的方式，在终端看来，也是会影响速度的。这个和我们的原则：终端请求，需要用最快的速度返回给终端，是冲突的。

这时候，最理想的是，获取完给终端返回的数据后，就断开连接，后面的日志和统计等动作，在断开连接后，后台继续完成即可。

怎么做到呢？我们先看其中的一种方法：

```
local response, user_stat = logic_func.get_response(request)
ngx.say(response)
ngx.eof()

if user_stat then
   local ret = db_redis.update_user_data(user_stat)
end
```

没错，最关键的一行代码就是ngx.eof()， 它可以即时关闭连接，把数据返回给终端，后面的数据库操作还会运行。比如上面代码中的

```
local response, user_stat = logic_func.get_response(request)
```

运行了0.1秒，而

```
db_redis.update_user_data(user_stat)
```

运行了0.2秒，在没有使用ngx.eof()之前，终端感知到的是0.3秒，而加上ngx.eof()之后，终端感知到的只有0.1秒。

需要注意的是，你不能任性的把阻塞的操作加入代码，即使在ngx.eof()之后。 虽然已经返回了终端的请求，但是，nginx的worker还在被你占用。所以在keep alive的情况下，本次请求的总时间，会把上一次eof()之后的时间加上。如果你加入了阻塞的代码，nginx的高并发就是空谈。

有没有其他的方法来解决这个问题呢？我们会在ngx.timer.at里面给大家介绍更优雅的方案。

调试

调试

调试是一个程序猿非常重要的能力，人写的程序总会有bug，所以需要debug。如何方便和快速的定位bug，是我们讨论的重点，只要bug能定位，解决就不是问题。

对于熟悉用Visual Studio和Eclipse这些强大的集成开发环境的来做C++和Java的同学来说，OpenResty的debug要原始很多，但是对于习惯Python开发的同学来说，又是那么的熟悉。张银奎有本《软件调试》的书，windows客户端程序猿应该都看过，大家可以去试读下，看看里面有多复杂:(

对于OpenResty，坏消息是，没有单步调试这些玩意儿（我们尝试搞出来过ngx lua的单步调试，但是没人用...）;好消息是，它像Python一样，非常简单，不用复杂的技术，只靠print和log就能定位绝大部分问题，难题有火焰图这个神器。

- 关闭code cache

这个选项在调试的时候最好关闭。

```
lua_code_cache off;
```

这样，你修改完代码后，不用reload nginx就可以生效了。在生产环境下记得打开这个选项。

- 记录日志

这个看上去谁都会的东西，要想做好也不容易。

你有遇到这样的情况吗？QA发现了一个bug，开发说我修改代码加个日志看看，然后QA重现这个问题，发现日志不够详细，需要再加，反复几次，然后再给QA一个没有日志的版本，继续测试其他功能。

如果产品已经发布到用户那里了呢？如果用户那里是隔离网，不能远程怎么办？

你在写代码的时候，就需要考虑到调试日志。 比如这个代码：

```
local response, err = redis_op.finish_client_task(client_mid, task_id)
```

```
if response then
    put_job(client_mid, result)
    ngx.log(ngx.WARN, "put job:", common.json_encode({channel="task_statu
s", mid=client_mid, data=result}))
end
```

我们在做一个操作后，就把结果记录到nginx的error.log里面，等级是warn。在生产环境下，日志等级默认为error，在我们需要详细日志的时候，把等级调整为warn即可。在我们的实际使用中，我们会把一些很少发生的重要事件，做为error级别记录下来，即使它并不是nginx的错误。

与日志配套的，你需要logrotate来做日志的切分和备份。

调用其他C函数动态库

调用其他C函数动态库

Linux下的动态库一般都以 .so 结束命名，而Windows下一般都以 .dll 结束命名。Lua作为一种嵌入式语言，和C具有非常好的亲缘性，这也是LUA赖以生存、发展的根本，所以Nginx+Lua=Openresty，魔法就这么神奇的发生了。

NgxLuaModule里面尽管提供了十分丰富的API，但他一定不可能满足我们的形形色色的需求。我们总是要和各种组件、算法等形形色色的第三方库进行协作。那么如何在Lua中加载动态加载第三方库，就显得非常有用。

扯一些额外话题，Lua解释器目前有两个最主流分支。

- Lua官方发布的标准版Lua
- Google开发维护的Luajit

Luajit中加入了Just In Time等编译技术，是的Lua的解释、执行效率有非常大的提升。除此以外，还提供了FFI。

> 什么是FFI？

```
The FFI library allows calling external C functions and using C data
structures from pure Lua code.
```

通过FFI的方式加载其他C接口动态库，这样我们就可以有很多有意思的玩法。

当我们碰到CPU密集运算部分，我们可以把他用C的方式实现一个效率最高的版本，对外到处API，打包成动态库，通过FFI来完成API调用。这样我们就可以兼顾程序灵活、执行高效，大大弥补了Luajit自身的不足。

> 使用FFI判断操作系统

```
local ffi = require("ffi")
if ffi.os == "Windows" then
    print("windows")
elseif ffi.os == "OSX" then
```

```
    print("MAC OS X")
else
    print(ffi.os)
end
```

调用zlib压缩库

```
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)
```

自定义定义C类型的方法

```
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
```

> Lua和Luajit对比

可以这么说，Luajit应该是全面胜出，无论是功能、效率都是标准Lua不能比的。目前最新版Openresty默认也都使用Luajit。

世界为我所用，总是有惊喜等着你，如果那天你发现自己站在了顶峰，那我们就静下心来改善一下顶峰，把他推到更高吧。

我的lua代码需要调优么

网上有大量对lua调优的推荐，我们应该如何看待？

lua的解析器有官方的standard lua和luajit，需要明确一点的是目前大量的优化文章都比较陈旧，而且都是针对standard lua解析器的，standard lua解析器在性能上需要书写着自己规避，才能写出高性能来。需要各位看官注意的是，ngx-lua最新版默认已经绑定luajit，优化手段和方法已经略有不同。我们现在的做法是：代码易读是首位，目前还没有碰到同样代码换个写法就有质的提升，如果我们对某个单点功能有性能要求，那么建议用luajit的FFI方法直接调用C接口更直接一点。

代码出处：http://www.cnblogs.com/lovevivi/p/3284643.html

```
3.0 避免使用table.insert()

下面来看看4个实现表插入的方法。在4个方法之中table.insert()在效率上不如其他方法，是应该避免使用的。
使用table.insert
local a = {}
local table_insert = table.insert
for i = 1,100 do
   table_insert( a, i )
end

使用循环的计数

local a = {}
for i = 1,100 do
   a[i] = i
end
使用table的size

local a = {}
for i = 1,100 do
   a[#a+1] = i
end
使用计数器

local a = {}
local index = 1
for i = 1,100 do
   a[index] = i
   index = index+1
end

4.0 减少使用 unpack()函数
Lua的unpack()函数不是一个效率很高的函数。你完全可以写一个循环来代替它的作用。
```

```
使用unpack()

local a = { 100, 200, 300, 400 }
for i = 1,100 do
   print( unpack(a) )
end
代替方法

local a = { 100, 200, 300, 400 }
for i = 1,100 do
   print( a[1],a[2],a[3],a[4] )
end
```

针对这篇文章内容写了一些测试代码：

```
local start = os.clock()

local function sum( ... )
    local args = {...}
    local a = 0
    for k,v in pairs(args) do
        a = a + v
    end
    return a
end

local function test_unit(  )
    -- t1: 0.340182 s
    -- local a = {}
    -- for i = 1,1000 do
    --    table.insert( a, i )
    -- end

    -- t2: 0.332668 s
    -- local a = {}
    -- for i = 1,1000 do
    --    a[#a+1] = i
    -- end

    -- t3: 0.054166 s
    -- local a = {}
    -- local index = 1
    -- for i = 1,1000 do
    --    a[index] = i
    --    index = index+1
    -- end

    -- p1: 0.708012 s
    -- local a = 0
    -- for i=1,1000 do
    --     local t = { 1, 2, 3, 4 }
    --     for i,v in ipairs( t ) do
```

```
    --         a = a + v
    --     end
    -- end

    -- p2: 0.660426 s
    -- local a = 0
    -- for i=1,1000 do
    --     local t = { 1, 2, 3, 4 }
    --     for i = 1,#t do
    --        a = a + t[i]
    --     end
    -- end

    -- u1: 2.121722 s
    -- local a = { 100, 200, 300, 400 }
    -- local b = 1
    -- for i = 1,1000 do
    --    b = sum(unpack(a))
    -- end

    -- u2: 1.701365 s
    -- local a = { 100, 200, 300, 400 }
    -- local b = 1
    -- for i = 1,1000 do
    --    b = sum(a[1], a[2], a[3], a[4])
    -- end

    return b
end

for i=1,10 do
    for j=1,1000 do
        test_unit()
    end
end

print(os.clock()-start)
```

从运行结果来看，除了t3有本质上的性能提升（六倍性能差距，但是t3写法相当丑陋），其他不同的写法都在一个数量级上。你是愿意让代码更易懂还是更牛逼，就看各位看官自己的抉择了。不要盲信，也不要不信，各位要睁开眼自己多做测试。

另外说明：文章提及的使用局部变量、缓存table元素，在luajit中还是很有用的。

todo：优化测试用例，让他更直观，自己先备注一下。

变量的共享范围

变量的共享范围

> 本章内容来自openresty讨论组 这里

先看两段代码：

```
-- index.lua
local uri_args = ngx.req.get_uri_args()
local mo = require('mo')
mo.args = uri_args
```

```
-- mo.lua

local showJs = function(callback, data)
    local cjson = require('cjson')
    ngx.say(callback .. '(' .. cjson.encode(data) .. ')')
end
local self.jsonp = self.args.jsonp
local keyList = string.split(self.args.key_list, ',')
for i=1, #keyList do
    -- do something
    ngx.say(self.args.kind)
end
showJs(self.jsonp, valList)
```

大概代码逻辑如上，然后出现这种情况：

生产服务器中，如果没有用户访问，自己几个人测试，一切正常。

同样生产服务器，我将大量的用户请求接入后，我不停刷新页面的时候会出现部分情况（概率也不低，几分之一，大于10%），输出的callback（也就是来源于self.jsonp，即URL参数中的jsonp变量）和url地址中不一致（我自己测试的值是?jsonp=jsonp1435220570933，而用户的请求基本上都是?jsonp=jquery....)错误的情况都是会出现用户请求才会有的jquery....这种字符串。另外URL参数中的kind是1，我在循环中输出会有“1”或“nil”的情况。不仅这两种参数，几乎所有url中传递的参数，都有可能变成其他请求链接中的参数。

基于以上情况，个人判断会不会是在生产服务器大量用户请求中，不同请求参数串掉了，但是如果这样，是否应该会出现我本次的获取参数是某个其他用户的值，那么for循环中的值也

应该固定的，而不会是一会儿是我自己请求中的参数值，一会儿是其他用户请求中的参数值。

问题在哪里？

Lua module 是 VM 级别共享的，见这里。

self.jsonp变量一不留神全局共享了，而这肯定不是作者期望的。所以导致了高并发应用场景下偶尔出现异常错误的情况。

每请求的数据在传递和存储时须特别小心，只应通过你自己的函数参数来传递，或者通过 ngx.ctx 表。前者是推荐的玩法，因为效率高得多。

贴一个ngx.ctx的例子：

```
    location /test {
        rewrite_by_lua '
            ngx.ctx.foo = 76
        ';
        access_by_lua '
            ngx.ctx.foo = ngx.ctx.foo + 3
        ';
        content_by_lua '
            ngx.say(ngx.ctx.foo)
        ';
    }
```

Then GET /test will yield the output

> 79

动态限速

动态限速

> 内容来源于openresty讨论组，点击这里

在我们的应用场景中，有大量的限制并发、下载传输速率这类要求。突发性的网络峰值会对企业用户的网络环境带来难以预计的网络灾难。

nginx示例配置：

```
location /download_internal/ {
    internal;
    send_timeout 10s;
    limit_conn perserver 100;
    limit_rate 0k;

    chunked_transfer_encoding off;
    default_type application/octet-stream;

    alias ../download/;
}
```

我们从一开始，就想把速度值做成变量，但是发现limit_rate不接受变量。我们就临时的修改配置文件限速值，然后给nginx信号做reload。只是没想到这一临时，我们就用了一年多。

直到刚刚，讨论组有人问起网络限速如何实现的问题，春哥给出了大家都喜欢的办法：

> 地址：https://groups.google.com/forum/#!topic/openresty/aespbrRvWOU

```
可以在 Lua 里面（比如 access_by_lua 里面）动态读取当前的 URL 参数，然后设置 nginx
 的内建变量$limit_rate（在 Lua 里访问就是 ngx.var.limit_rate）。

http://nginx.org/en/docs/http/ngx_http_core_module.html#var_limit_rate
```

改良后的限速代码：

```
location /download_internal/ {
    internal;
    send_timeout 10s;
```

```
    access_by_lua 'ngx.var.limit_rate = "300K"';

    chunked_transfer_encoding off;
    default_type application/octet-stream;

    alias ../download/;
}
```

经过测试，绝对达到要求。有了这个东东，我们就可以在lua上直接操作限速变量实时生效。再也不用之前笨拙的reload方式了。

PS: ngx.var.limit_rate 限速是基于请求的，如果相同终端发起两个连接，那么终端的最大速度将是limit_rate的两倍，原文如下：

```
Syntax: limit_rate rate;
Default:
limit_rate 0;
Context: http, server, location, if in location

Limits the rate of response transmission to a client. The rate is specifi
ed in bytes per second. The zero value disables rate limiting. The limit
is set per a request, and so if a client simultaneously opens two connect
ions, the overall rate will be twice as much as the specified limit.
```

shared.dict 非队列性质

ngx.shared.DICT 非队列性质

执行阶段和主要函数请参考维基百科 HttpLuaModule#ngx.shared.DICT

非队列性质

ngx.shared.DICT的实现是采用红黑树实现，当申请的缓存被占用完后如果有新数据需要存储则采用LRU算法淘汰掉“多余”数据。

这样数据结构的在带有队列性质的业务逻辑下会出现的一些问题：

我们用shared作为缓存，接纳终端输入并存储，然后在另外一个线程中按照固定的速度去处理这些输入，代码如下:

```
-- [ngx.thread.spawn](http://wiki.nginx.org/HttpLuaModule#ngx.thread.spaw
n) #1 存储线程 理解为生产者

    ....
    local cache_str = string.format([[%s&%s&%s&%s&%s&%s&%s]], net, name,
ip,
                    mac, ngx.var.remote_addr, method, md5)
    local ok, err = ngx_nf_data:safe_set(mac, cache_str, 60*60)  --这些是
缓存数据
    if not ok then
        ngx.log(ngx.ERR, "stored nf report data error: "..err)
    end
    ....

-- [ngx.thread.spawn](http://wiki.nginx.org/HttpLuaModule#ngx.thread.spaw
n) #2 取线程 理解为消费者

    while not ngx.worker.exiting() do
        local keys = ngx_share:get_keys(50)  -- 一秒处理50个数据

        for index, key in pairs(keys) do
            str = ((nil ~= str) and str..[[#]]..ngx_share:get(key)) or ng
x_share:get(key)
            ngx_share:delete(key)  --干掉这个key
        end
        .... --一些消费过程，看官不要在意
        ngx.sleep(1)
    end
```

在上述业务逻辑下会出现由生产者生产的某些key-val对永远不会被消费者取出并消费，原因就是shared.DICT不是队列，ngx_shared:get_keys(n)函数不能保证返回的n个键值对是满足

FIFO规则的，从而导致问题发生。

问题解决

问题的原因已经找到，解决方案有如下几种：1.修改暂存机制，采用redis的队列来做暂存；2.调整消费者的消费速度，使其远远大于生产者的速度；3.修改ngx_shared:get_keys()的使用方法，即是不带参数；

方法3和2本质上都是一样的，由于业务已经上线，方法1周期太长，于是采用方法2解决，在后续的业务中不再使用shared.DICT来暂存队列性质的数据

如何添加自己的lua api

如何对nginx lua module添加新api

本文真正的目的，绝对不是告诉大家如何在nginx lua module添加新api这么点东西。而是以此为例，告诉大家nginx模块开发环境搭建、码字编译、编写测试用例、代码提交、申请代码合并等。给大家顺路普及一下git的使用。

目前有个应用场景，需要获取当前nginx worker数量的需要，所以添加一个新的接口ngx.config.workers()。由于这个功能实现简单，非常适合大家当做例子。废话不多说，let's fly now！

> 获取openresty默认安装包（辅助搭建基础环境）：

```
$ wget http://openresty.org/download/ngx_openresty-1.7.10.1.tar.gz
$ tar -xvf ngx_openresty-1.7.10.1.tar.gz
$ cd ngx_openresty-1.7.10.1
```

> 从github上fork代码

- 进入lua-nginx-module，点击右侧的Fork按钮
- Fork完毕后，进入自己的项目，点击 Clone in Desktop 把项目clone到本地

> 预编译，本步骤参考这里：

```
$ ./configure
$ make
```

`注意这里不需要make install`

> 修改自己的源码文件

```
# ngx_lua-0.9.15/src/ngx_http_lua_config.c
```

> 编译变化文件

```
$ rm ./nginx-1.7.10/objs/addon/src/ngx_http_lua_config.o
$ make
```

搭建测试模块

> 安装perl cpan 点击查看

```
$ cpan
cpan[2]> install Test::Nginx::Socket::Lua
```

> 书写测试单元

```
$ cat 131-config-workers.t
# vim:set ft= ts=4 sw=4 et fdm=marker:
use lib 'lib';
use Test::Nginx::Socket::Lua;

#worker_connections(1014);
#master_on();
#workers(2);
#log_level('warn');

repeat_each(2);
#repeat_each(1);

plan tests => repeat_each() * (blocks() * 3);

#no_diff();
#no_long_string();
run_tests();

__DATA__

=== TEST 1: content_by_lua
--- config
    location /lua {
        content_by_lua '
            ngx.say("workers: ", ngx.config.workers())
        ';
    }
--- request
GET /lua
--- response_body_like chop
```

```
^workers: 1$
--- no_error_log
[error]
```

```
$ cat 132-config-workers_5.t
# vim:set ft= ts=4 sw=4 et fdm=marker:
use lib 'lib';
use Test::Nginx::Socket::Lua;

#worker_connections(1014);
#master_on();
workers(5);
#log_level('warn');

repeat_each(2);
#repeat_each(1);

plan tests => repeat_each() * (blocks() * 3);

#no_diff();
#no_long_string();
run_tests();

__DATA__

=== TEST 1: content_by_lua
--- config
    location /lua {
        content_by_lua '
            ngx.say("workers: ", ngx.config.workers())
        ';
    }
--- request
GET /lua
--- response_body_like chop
^workers: 5$
--- no_error_log
[error]
```

单元测试

```
$ export PATH=/path/to/your/nginx/sbin:$PATH    #设置nginx查找路径
$ cd ngx_lua-0.9.15                         # 进入你修改的模块
$ prove t/131-config-workers.t              # 测试指定脚本
t/131-config-workers.t .. ok
All tests successful.
Files=1, Tests=6,  1 wallclock secs ( 0.04 usr  0.00 sys +  0.18 cusr  0.
05 csys =  0.27 CPU)
Result: PASS
$
$ prove t/132-config-workers_5.t            # 测试指定脚本
t/132-config-workers_5.t .. ok
```

```
All tests successful.
Files=1, Tests=6,  0 wallclock secs ( 0.03 usr  0.00 sys +  0.17 cusr  0.
04 csys =  0.24 CPU)
Result: PASS
```

提交代码，推动我们的修改被官方合并

- 首先把代码commit到github
- commit成功后，以次点击github右上角的Pull request -> New pull request
- 这时候github会弹出一个自己与官方版本对比结果的页面，里面包含有我们所有的修改，确定我们的修改都被包含其中，点击Create pull request按钮
- 输入标题、内容（you'd better write in english）,点击Create pull request按钮
- 提交完成，就可以等待官方作者是否会被采纳了（代码+测试用例，必不可少）

来看看我们的成果吧：

pull request : 点击查看commit detail: 点击查看

正确使用长链接

KeepAlive

在OpenResty中，连接池在使用上如果不加以注意，容易产生数据写错地方，或者得到的应答数据异常以及类似的问题，当然使用短连接可以规避这样的问题，但是在一些企业用户环境下，短连接+高并发对企业内部的防火墙是一个巨大的考验，因此，长连接自有其勇武之地，使用它的时候要记住，长连接一定要保持其连接池中所有连接的正确性。

```
-- 错误的代码
local function send()
    for i = 1, count do
        local ssdb_db, err = ssdb:new()
        local ok, err = ssdb_db:connect(SSDB_HOST, SSDB_PORT)
        if not ok then
            ngx.log(ngx.ERR, "create new ssdb failed!")
        else
               local key,err = ssdb_db:qpop(something)
             if not key then
                   ngx.log(ngx.ERR, "ssdb qpop err:", err)
               else
                   local data, err = ssdb_db:get(key[1])
                   -- other operations
               end
        end
    end
    ssdb_db:set_keepalive(SSDB_KEEP_TIMEOUT, SSDB_KEEP_COUNT)
end
-- 调用
while true do
    local ths = {}
    for i=1,THREADS do
        ths[i] = ngx.thread.spawn(send)       ----创建线程
    end
    for i = 1, #ths do
        ngx.thread.wait(ths[i])               ----等待线程执行
    end
    ngx.sleep(0.020)
end
```

以上代码在测试中发现，应该得到get(key)的返回值有一定几率为key。

原因即是在ssdb创建连接时可能会失败，但是当得到失败的结果后依然调用ssdb_db：set_keepalive将此连接并入连接池中。

正确地做法是如果连接池出现错误，则不要将该连接加入连接池。

```
local function send()
    for i = 1, count do
        local ssdb_db, err = ssdb:new()
        local ok, err = ssdb_db:connect(SSDB_HOST, SSDB_PORT)
        if not ok then
            ngx.log(ngx.ERR, "create new ssdb failed!")
            return
        else
              local key,err = ssdb_db:qpop(something)
             if not key then
                   ngx.log(ngx.ERR, "ssdb qpop err:", err)
              else
                  local data, err = ssdb_db:get(key[1])
                  -- other operations
              end
              ssdb_db:set_keepalive(SSDB_KEEP_TIMEOUT, SSDB_KEEP_COUNT)
        end
    end
end
```

所以，当你使用长连接操作db出现结果错乱现象时，首先应该检查下是否存在长连接使用不当的情况。

如何引用第三方resty库

如何引用第三方resty库

使用动态DNS来完成HTTP请求

使用动态DNS来完成HTTP请求

其实针对大多应用场景，DNS是不会频繁变更的，使用nginx默认的resolver配置方式就能解决。

在奇虎360企业版的应用场景下，需要支持的系统众多：win、centos、ubuntu等，不同的操作系统获取dns的方法都不太一样。再加上我们使用docker，导致我们在容器内部获取dns变得更加难以准确。

如何能够让Nginx使用随时可以变化的DNS源，成为我们急待解决的问题。

当我们需要在某一个请求内部发起这样一个http查询，采用proxy_pass是不行的（依赖resolver的dns，如果dns有变化，必须要重新加载配置），并且由于proxy_pass不能直接设置keepconn，导致每次请求都是短链接，性能损失严重。

使用resty.http，目前这个库只支持ip：port的方式定义url，其内部实现并没有支持domain解析。resty.http是支持set_keepalive完成长连接，这样我们只需要让他支持dns解析就能有完美解决方案了。

```
local resolver = require "resty.dns.resolver"
local http     = require "resty.http"

function get_domain_ip_by_dns( domain )
  -- 这里写死了google的域名服务ip，要根据实际情况做调整（例如放到指定配置或数据库中）
  local dns = "8.8.8.8"

  local r, err = resolver:new{
      nameservers = {dns, {dns, 53} },
      retrans = 5,  -- 5 retransmissions on receive timeout
      timeout = 2000,  -- 2 sec
  }

  if not r then
      return nil, "failed to instantiate the resolver: " .. err
  end

  local answers, err = r:query(domain)
  if not answers then
      return nil, "failed to query the DNS server: " .. err
  end

  if answers.errcode then
      return nil, "server returned error code: " .. answers.errcode .. ":
```

```
  " .. answers.errstr
   end

   for i, ans in ipairs(answers) do
     if ans.address then
       return ans.address
     end
   end

   return nil, "not founded"
end

function http_request_with_dns( url, param )
    -- get domain
    local domain = ngx.re.match(url, [[//([\S]+?)/]])
    domain = (domain and 1 == #domain and domain[1]) or nil
    if not domain then
        ngx.log(ngx.ERR, "get the domain fail from url:", url)
        return {status=ngx.HTTP_BAD_REQUEST}
    end

    -- add param
    if not param.headers then
        param.headers = {}
    end
    param.headers.Host = domain

    -- get domain's ip
    local domain_ip, err = get_domain_ip_by_dns(domain)
    if not domain_ip then
        ngx.log(ngx.ERR, "get the domain[", domain ,"] ip by dns failed:"
, err)
        return {status=ngx.HTTP_SERVICE_UNAVAILABLE}
    end

    -- http request
    local httpc = http.new()
    local temp_url = ngx.re.gsub(url, "//"..domain.."/", string.format("/
/%s/", domain_ip))

    local res, err = httpc:request_uri(temp_url, param)
    if err then
        return {status=ngx.HTTP_SERVICE_UNAVAILABLE}
    end

    -- httpc:request_uri 内部已经调用了keepalive，默认支持长连接
    -- httpc:set_keepalive(1000, 100)
    return res
end
```

动态DNS，域名访问，长连接，这些都具备了，貌似可以安稳一下。在压力测试中发现这里面有个机制不太好，就是对于指定域名解析，每次都要和DNS服务回话询问IP地址，实际上这是不需要的。普通的浏览器，都会对DNS的结果进行一定的缓存，那么这里也必须要使用

了。

对于缓存实现代码，请参考ngx_lua相关章节，肯定会有惊喜等着你挖掘碰撞。

缓存失效风暴

缓存失效风暴

看下这个段伪代码：

```
local value = get_from_cache(key)
if not value then
    value = query_db(sql)
    set_to_cache(value,  timeout = 100)
end
return value
```

看上去没有问题，在单元测试情况下，也不会有异常。

但是，进行压力测试的时候，你会发现，每隔100秒，数据库的查询就会出现一次峰值。如果你的cache失效时间设置的比较长，那么这个问题被发现的机率就会降低。

为什么会出现峰值呢？想象一下，在cache失效的瞬间，如果并发请求有1000条同时到了`query_db(sql)`这个函数会怎样？没错，会有1000个请求打向数据库。这就是缓存失效瞬间引起的风暴。它有一个英文名，叫"dog-pile effect"。

怎么解决？自然的想法是发现缓存失效后，加一把锁来控制数据库的请求。具体的细节，春哥在lua-resty-lock的文档里面做了详细的说明，我就不重复了，请看这里。多说一句，ua-resty-lock库本身已经替你完成了wait for lock的过程，看代码的时候需要注意下这个细节。

Lua

Lua

下标从1开始

下标从1开始

- 在_lua_中，数组下标从1开始计数。
- 官方：Lua lists have a base index of 1 because it was thought to be most friendly for non-programmers, as it makes indices correspond to ordinal element positions.
- 在初始化一个数组的时候，若不显式地用键值对方式赋值，则会默认用数字作为下标，从1开始。由于在_lua_内部实际采用哈希表和数组分别保存键值对、普通值，所以不推荐混合使用这两种赋值方式。

```
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
```

局部变量

局部变量

定义

Lua中的局部变量要用local关键字来显示定义，不用local显示定义的变量就是全局变量：

```
g_var = 1          -- global var
local l_var = 2    -- local var
```

作用域

局部变量的生命周期是有限的，它的作用域仅限于声明它的块（block）。 一个块是一个控制结构的执行体、或者是一个函数的执行体再或者是一个程序块（chunk）。我们可以通过下面这个例子来理解一下局部变量作用域的问题：

> 示例代码test.lua

```
x = 10
local i = 1          --程序块中的局部变量i

while i <=x do
  local x = i * 2    --while循环体中的局部变量x
  print(x)           --打印2, 4, 6, 8, ...(实际输出格式不是这样的, 这里只是表示输出结果)
  i = i + 1
end

if i > 20 then
  local x            --then中的局部变量x
  x = 20
  print(x + 2)       --如果i > 20 将会打印22, 此处的x是局部变量
else
  print(x)           --打印10,  这里x是全局变量
end

print(x)             --打印10
```

使用局部变量的好处

使用局部变量的一个好处是，局部变量可以避免将一些无用的名称引入全局环境，避免全局环境的污染。另外，访问局部变量比访问全局变量更快。同时，由于局部变量出了作用域之后生命周期结束，这样可以被垃圾回收器及时释放。

在Lua中，应该尽量让定义变量的语句靠近使用变量的语句，这也可以被看做是一种良好的编程风格。在C这样的语言中，强制程序员在一个块（或一个过程）的起始处声明所有的局部变量，所以有些程序员认为在一个块的中间使用声明语句是一种不良好地习惯。实际上，在需要时才声明变量并且赋予有意义的初值，这样可以提高代码的可读性。对于程序员而言，相比在块中的任意位置顺手声明自己需要的变量，和必须跳到块的起始处声明，大家应该能掂量哪种做法比较方便了吧？

“尽量使用局部变量”是一种良好的编程风格。然而，初学者在使用Lua时，很容易忘记加上“local”来定义局部变量，这时变量就会自动变成全局变量，很可能导致程序出现意想不到的问题。那么我们怎么检测哪些变量是全局变量呢？我们如何防止全局变量导致的影响呢？下面给出一段代码，利用元表的方式来自动检查全局变量，并打印必要的调试信息：

检查模块的函数使用全局变量

> 把下面代码保存在foo.lua文件中。

```
module(..., package.seeall)  --使用module函数定义模块很不安全。如何定义和使用模块请查看模块章节

local function add(a, b)     --两个number型变量相加
    return a + b
end

function update_A()    --更新变量值
    A = 365
end

getmetatable(foo).__newindex = function (table, key, val)   --防止foo模块更改全局变量
   error('attempt to write to undeclared variable "' .. key .. '": ' .. debug.traceback())
end
```

> 把下面代码保存在use_foo.lua文件中。该文件和foo.lua在相同目录。

```
A = 360     --定义全局变量
local foo = require("foo")  --使用模块foo，如何定义和使用模块请查看模块章节

local b = foo.add(A, A)
print("b = ", b)

foo.update_A()
print("A = ", A)
```

运行use_foo.lua文件，输出结果如下:

```
b =  720
lua: .\foo.lua:13: attempt to write to undeclared variable "A": stack tra
ceback:
    .\foo.lua:13: in function <.\foo.lua:12>
    .\foo.lua:9: in function 'update_A'
    my.lua:7: in main chunk
    [C]: ?
stack traceback:
    [C]: in function 'error'
    .\foo.lua:13: in function <.\foo.lua:12>
    .\foo.lua:9: in function 'update_A'
    my.lua:7: in main chunk
    [C]: ?
```

在 _updateA() 函数使用全局变量"A"时，抛出异常。这利用了模块，想了解更多元表的内容，可以查看模块章节。

Lua 上下文中应当严格避免使用自己定义的全局变量。可以使用一个 lua-releng 工具来扫描 Lua 代码，定位使用 Lua 全局变量的地方。lua-releng 的相关链接：http://wiki.nginx.org/HttpLuaModule#Lua_Variable_Scope

把lua-releng.pl文件和上述两个文件放在相同目录下，然后进入该目录，运行lua-releng.pl，得到如下结果：

```
# ~/work/conf$ perl lua-releng.pl
WARNING: No "_VERSION" or "version" field found in `foo.lua`.
Checking use of Lua global variables in file foo.lua...
    op no.    line    instruction    args    ; code
    8    [7]    SETGLOBAL    1 -4    ; update_A
    2    [8]    SETGLOBAL    0 -1    ; A
Checking line length exceeding 80...
WARNING: No "_VERSION" or "version" field found in `use_foo.lua`.
Checking use of Lua global variables in file use_foo.lua...
    op no.    line    instruction    args    ; code
    2    [1]    SETGLOBAL    0 -1    ; A
    7    [4]    GETGLOBAL    2 -1    ; A
    8    [4]    GETGLOBAL    3 -1    ; A
    18    [8]    GETGLOBAL    4 -1    ; A
Checking line length exceeding 80...
```

结果显示：在foo.lua文件中，第7行设置了一个全局变量update_A（注意：在lua中函数也是变量），第8行设置了一个全局变量A；在use_foo.lua文件中，第1行设置了一个全局变量A，第4行使用了两次全局变量A，第8行使用了一次全局变量A。

判断数组大小

判断数组大小

lua数组需要注意的细节

lua中，数组的实现方式其实类似于C++中的map，对于数组中所有的值，都是以键值对的形式来存储（无论是显式还是隐式）， lua 内部实际采用哈希表和数组分别保存键值对、普通值，所以不推荐混合使用这两种赋值方式。尤其需要注意的一点是：lua数组中允许nil值的存在，但是数组默认结束标志却是nil。这类比于C语言中的字符串，字符串中允许'\0'存在，但当读到'\0'时，就认为字符串已经结束了。

初始化是例外，在lua相关源码中，初始化数组时首先判断数组的长度，若长度大于0，并且最后一个值不为nil，返回包括nil的长度；若最后一个值为nil，则返回截至第一个非nil值的长度。

注意！！一定不要使用#操作符来计算包含nil的数组长度，这是一个未定义的操作，不一定报错，但不能保证结果如你所想。如果你要删除一个数组中的元素，请使用remove函数，而不是用nil赋值。

```
local arr1 = {1, 2, 3, [5]=5}
print(#arr1)                -- output: 3

local arr2 = {1, 2, 3, nil, nil}
print(#arr2)                -- output: 3

local arr3 = {1, nil, 2}
arr3[5] = 5
print(#arr3)                -- output: 1

local arr4 = {1,[3]=2}
arr4[4] = 4
print(#arr4)                -- output: 4
```

按照我们上面的分析，应该为1，但这里却是4，所以一定不要使用#操作符来计算包含nil的数组长度。

非空判断

非空判断

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，会提示这样的错误：

```
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 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 _G.next(a) == nil then
    print("_G.next(a) == nil")
end

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

if _G.next(c) == nil then
    print("_G.next(c) == nil")
end

-- error
--if _G.next(d) == nil then
--    print("_G.next(d) == nil")
--end
```

返回的结果如下：

```
0
0
2
d == nil
_G.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 isTableEmpty(t)
    if t == nil or _G.next(t) == nil then
        return true
    else
        return false
    end
end
```

正则表达式

正则表达式

在_OpenResty_中，同时存在两套正则表达式规范： Lua 语言的规范和_Nginx_的规范，即使您对 Lua 语言中的规范非常熟悉，我们仍不建议使用 Lua 中的正则表达式。一是因为 Lua 中正则表达式的性能并不如 Nginx 中的正则表达式优秀；二是 Lua 中的正则表达式并不符合 POSIX 规范，而 Nginx 中实现的是标准的 POSIX 规范，后者明显更具备通用性。

Lua 中的正则表达式与Nginx中的正则表达式相比，有5%-15%的性能损失，而且Lua将表达式编译成Pattern之后，并不会将Pattern缓存，而是每此使用都重新编译一遍，潜在地降低了性能。 Nginx 中的正则表达式可以通过参数缓存编译过后的Pattern，不会有类似的性能损失。

o选项参数用于提高性能，指明该参数之后，被编译的Pattern将会在worker进程中缓存，并且被当前worker进程的每次请求所共享。Pattern缓存的上限值通过lua_regex_cache_max_entries来修改。

```
# nginx.conf
location /test {
    content_by_lua '
        local regex = [[\\d+]]

        -- 参数"o"是开启缓存必须的
        local m = ngx.re.match("hello, 1234", regex, "o")
        if m then
            ngx.say(m[0])
        else
            ngx.say("not matched!")
        end
    ';
}
# 在网址中输入"yourURL/test", 即会在网页中显示1234。
```

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

Lua正则简单汇总

- 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
```

- string.gsub 用来查找匹配模式的串，并将使用替换串其替换掉，但并不修改原字符串，而是返回一个修改后的字符串的副本，函数有目标串，模式串，替换串三个参数，使用范例如下：

```
local a = "Lua is cute"
local b = string.gsub(a, "cute", "great")
print(a) --> Lua is cute
print(b) --> Lua is great
```

- 还有一点值得注意的是，'%b' 用来匹配对称的字符，而不是一般正则表达式中的单词的开始、结束。'%b' 用来匹配对称的字符，而且采用贪婪匹配。常写为 '%bxy' ，x 和 y 是任意两个不同的字符；x 作为匹配的开始，y 作为匹配的结束。比如，'%b()' 匹配以 '(' 开始，以 ')' 结束的字符串：

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

不用标准库

不用标准库

虚变量

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)，以单个下划线（ “_” ）来命名，用它来丢弃不需要的数值，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, finis
h
                                                    -- 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
```

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，静默丢掉，这样就直接得到第二个返回值了。

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：

```
local t = {1, 3, 5}

for i,v in ipairs(t) do
    print(i,v)
end                             --输出1  1
                                   -- 2  3
                                   -- 3  5
```

```
for _,v in ipairs(t) do
    print(v)
end                          --输出 1
                                 -- 3
                                 -- 5
```

函数在调用代码前定义

函数在调用代码前定义

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：

```
local i = 100
i = add_one(i)

local function add_one(i)
    return i + 1
end
```

你会得到一个错误提示：

> [error] 10514#0: *5 lua entry thread aborted: runtime error: attempt to call global 'add_one' (a nil value)

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，自然会得到nil的错误。

一般地，由于全局变量是每请求的生命期，因此以此种方式定义的函数的生命期也是每请求的。为了避免每请求创建和销毁Lua closure的开销，建议将函数的定义都放置在自己的Lua module中，例如：

```
-- my_module.lua
module("my_module", package.seeall)
function foo()
    -- your code
end
```

然后，再在content_by_lua_file指向的.lua文件中调用它：

```
local my_module = require "my_module"
my_module.foo()
```

因为Lua module只会在第一次请求时加载一次（除非显式禁用了lua_code_cache配置指令），后续请求便可直接复用。

抵制使用module()函数来定义Lua模块

抵制使用module()函数来定义Lua模块

旧式的模块定义方式是通过 module("filename"[,package.seeall]) 来显示声明一个包，现在官方不推荐再使用这种方式。这种方式将会返回一个由_filename_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。

如果只给 module 函数一个参数（也就是文件名）的话，前面定义的全局变量就都不可用了，包括 print 函数等，如果要让之前的全局变量可见，必须在定义 module 的时候加上参数package.seeall。调用完 module 函数之后，print这些系统函数不可使用的原因，是当前的整个环境被压入栈，不再可达。

module("filename", package.seeall) 这种写法仍然是不提倡的，官方给出了两点原因：

1. package.seeall 这种方式破坏了模块的高内聚，原本引入"filename"模块只想调用它的 foobar() 函数，但是它却可以读写全局属性，例如 "filename.os"。
2. module 函数压栈操作引发的副作用，污染了全局环境变量。例如 module("filename") 会创建一个 filename 的 table ，并将这个 table 注入全局环境变量中，这样使得没有引用它的文件也能调用 filename 模块的方法。

比较推荐的模块定义方法是：

```
-- square.lua 长方形模块
local _M = {}            -- 局部的变量
_M._VERSION = '1.0'      -- 模块版本

local mt = { __index = _M }

function _M.new(self, width, height)
    return setmetatable({ width=width, height=height }, mt)
end

function _M.get_square(self)
    return self.width * self.height
end

function _M.get_circumference(self)
    return (self.width + self.height) * 2
end

return _M
```

> 引用示例代码：

```
local square = require "square"

local s1 = square:new(1, 2)
print(s1:get_square())          --output: 2
print(s1:get_circumference())   --output: 6
```

- 另一个跟lua的module模块相关需要注意的点是，当lua_code_cache on开启时，require加载的模块是会被缓存下来的，这样我们的模块就会以最高效的方式运行，直到被显式地调用如下语句：

```
package.loaded["square"] = nil
```

我们可以利用这个特性代码来做一些进阶玩法。

点号与冒号操作符的区别

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))
```

> output:

```
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
```

参见 官方文档 中的以下片段:“

The colon syntax is used for defining methods, that is, functions that have an implicit

extra parameter self. Thus, the statement

```
function t.a.b.c:f (params) body end
```

is syntactic sugar for

```
t.a.b.c.f = function (self, params) body end
```

”

冒号的操作，只有当变量是类对象时才需要。有关如何使用Lua构造类，大家可参考相关章节。

测试

测试

单元测试

单元测试

单元测试（unit testing），是指对软件中的最小可测试单元进行检查和验证。对于单元测试中单元的含义，一般来说，要根据实际情况去判定其具体含义，如C语言中单元指一个函数，Java里单元指一个类，图形化的软件中可以指一个窗口或一个菜单等。总的来说，单元就是人为规定的最小的被测功能模块。单元测试是在软件开发过程中要进行的最低级别的测试活动，软件的独立单元将在与程序的其他部分相隔离的情况下进行测试。

单元测试的书写、验证，互联网公司几乎都是研发自己完成的，我们要保证代码出手时可交付、符合预期。如果连自己的预期都没达到，后面所有的工作，都将是额外无用功。

Lua中我们没有找到比较好的测试库，参考了Golang、Python等语言的单元测试书写方法以及调用规则，我们编写了lua-resty-test测试库，这里给自己的库推广一下，希望这东东也是你们的真爱。

> nginx示例配置

```
    #you do not need the following line if you are using
    #the ngx_openresty bundle:

    lua_package_path "/path/to/lua-resty-redis/lib/?.lua;;";

    server {
        location /test {
            content_by_lua_file test_case_lua/unit/test_example.lua;
        }
    }
```

> test_case_lua/unit/test_example.lua:

```
local tb    = require "resty.iresty_test"
local test = tb.new({unit_name="bench_example"})

function tb:init(  )
    self:log("init complete")
end

function tb:test_00001(  )
```

```
    error("invalid input")
end

function tb:atest_00002()
    self:log("never be called")
end

function tb:test_00003(  )
   self:log("ok")
end

-- units test
test:run()

-- bench test(total_count, micro_count, parallels)
test:bench_run(100000, 25, 20)
```

- init里面我们可以完成一些基础、公共变量的初始化，例如特定的url等
- test_*****函数中添加我们的单元测试代码
- 搞定测试代码，它即是单元测试，也是成压力测试

> 输出日志：

```
TIME   Name            Log
0.000  [bench_example] unit test start
0.000  [bench_example] init complete
0.000    \_[test_00001] fail ...de/nginx/test_case_lua/unit/test_example.
lua:9: invalid input
0.000    \_[test_00003] ↓ ok
0.000    \_[test_00003] PASS
0.000  [bench_example] unit test complete

0.000  [bench_example] !!!BENCH TEST START!!
0.484  [bench_example] succ count:   100001     QPS:     206613.65
0.484  [bench_example] fail count:   100001     QPS:     206613.65
0.484  [bench_example] loop count:   100000     QPS:     206611.58
0.484  [bench_example] !!!BENCH TEST ALL DONE!!!
```

埋个伏笔：在压力测试例子中，测试到的QPS大约21万的，这是我本机一台Mac Mini压测的结果。构架好，姿势正确，我们可以很轻松做出好产品。

后面会详细说一下用这个工具进行压力测试的独到魅力，做出一个NB的网络处理应用，这个测试库应该是你的利器。

API测试

API测试

API（Application Programming Interface）测试的自动化是软件测试最基本的一种类型。从本质上来说，API测试是用来验证组成软件的那些单个方法的正确性，而不是测试整个系统本身。API测试也称为单元测试（Unit Testing）、模块测试（Module Testing）、组件测试（Component Testing）以及元件测试（Element Testing）。从技术上来说，这些术语是有很大的差别的，但是在日常应用中，你可以认为它们大致相同的意思。它们背后的思想就是，必须确定系统中每个单独的模块工作正常，否则，这个系统作为一个整体不可能是正确的。毫无疑问，API测试对于任何重要的软件系统来说都是必不可少的。

我们对API测试的定位是服务对外输出的API接口测试，属于黑盒、偏重业务的测试步骤。

看过上一章内容的朋友还记得lua-resty-test，我们的API测试同样是需要它来完成。get_client_tasks是终端用来获取当前可执行任务清单的API，我们用它当做例子给大家做个介绍。

> nginx conf:

```
location ~* /api/([\w_]+?)\.json {
    content_by_lua_file lua/$1.lua;
}

location ~* /unit_test/([\w_]+?)\.json {
    lua_check_client_abort on;
    content_by_lua_file test_case_lua/unit/$1.lua;
}
```

> API测试代码：

```
-- unit test for /api/get_client_tasks.json
local tb = require "resty.iresty_test"
local json   = require("cjson")
local test = tb.new({unit_name="get_client_tasks"})

function tb:init(  )
    self.mid = string.rep('0',32)
end
```

```
function tb:test_0000()
    -- 正常请求
    local res = ngx.location.capture(
        '/api/get_client_tasks.json?mid='..self.mid,
        { method = ngx.HTTP_POST, body=[[{"type":[1600,1700]}]] }
    )

    if 200 ~= res.status then
        error("failed code:" .. res.status)
    end
end

function tb:test_0001()
    -- 缺少body
    local res = ngx.location.capture(
        '/api/get_client_tasks.json?mid='..self.mid,
        { method = ngx.HTTP_POST }
    )

    if 400 ~= res.status then
        error("failed code:" .. res.status)
    end
end

function tb:test_0002()
    -- 错误的json内容
    local res = ngx.location.capture(
        '/api/get_client_tasks.json?mid='..self.mid,
        { method = ngx.HTTP_POST, body=[[{"type":"[1600,1700]}]] }
    )

    if 400 ~= res.status then
        error("failed code:" .. res.status)
    end
end

function tb:test_0003()
    -- 错误的json格式
    local res = ngx.location.capture(
        '/api/get_client_tasks.json?mid='..self.mid,
        { method = ngx.HTTP_POST, body=[[{"type":"[1600,1700]"}]] }
    )

    if 400 ~= res.status then
        error("failed code:" .. res.status)
    end
end

test:run()
```

nginx output:

```
0.000  [get_client_tasks] unit test start
0.001    \_[test_0000] PASS
0.001    \_[test_0001] PASS
0.001    \_[test_0002] PASS
0.001    \_[test_0003] PASS
0.001  [get_client_tasks] unit test complete
```

使用capture来模拟请求，其实是不靠谱的。如果我们要完全100%模拟客户请求，这时候就要使用第三方cosocket库，例如lua-resty-http，这样我们才可以完全指定http参数。

性能测试

性能测试

性能测试应该有两个方向：

- 单接口压力测试
- 生产环境模拟用户操作高压力测试

生产环境模拟测试，目前我们都是交给公司的QA团队专门完成的。这块我只能粗略列举一下：

- 获取1000用户以上生产用户的访问日志（统计学要求1000是最小集合）
- 计算指定时间内（例如10分钟），所有接口的触发频率
- 使用测试工具（loadrunner, jmeter等）模拟用户请求接口
- 适当放大压力，就可以模拟2000、5000等用户数的情况

ab 压测

单接口压力测试，我们都是由研发团队自己完成的。传统一点的方法，我们可以使用ab(apache bench)这样的工具。

```
#ab -n10 -c2 http://haosou.com/

-- output:
...
Complete requests:      10
Failed requests:        0
Non-2xx responses:      10
Total transferred:      3620 bytes
HTML transferred:       1780 bytes
Requests per second:    22.00 [#/sec] (mean)
Time per request:       90.923 [ms] (mean)
Time per request:       45.461 [ms] (mean, across all concurrent requests
)
Transfer rate:          7.78 [Kbytes/sec] received
...
```

大家可以看到ab的使用超级简单，简单的有点弱了。在上面的例子中，我们发起了10个请求，每个请求都是一样的，如果每个请求有差异，ab就无能为力。

wrk 压测

单接口压力测试，为了满足每个请求或部分请求有差异，我们试用过很多不同的工具。最后找到了这个和我们距离最近、表现优异的测试工具wrk，这里我们重点介绍一下。

wrk如果要完成和ab一样的压力测试，区别不大，只是命令行参数略有调整。下面给大家举例每个请求都有差异的例子，供大家参考。

> scripts/counter.lua

```
-- example dynamic request script which demonstrates changing
-- the request path and a header for each request
-------------------------------------------------------------
-- NOTE: each wrk thread has an independent Lua scripting
-- context and thus there will be one counter per thread

counter = 0

request = function()
   path = "/" .. counter
   wrk.headers["X-Counter"] = counter
   counter = counter + 1
   return wrk.format(nil, path)
end
```

> shell执行

```
# ./wrk -c10 -d1 -s scripts/counter.lua http://baidu.com
Running 1s test @ http://baidu.com
  2 threads and 10 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    20.44ms    3.74ms  34.87ms   77.48%
    Req/Sec   226.05     42.13   270.00     70.00%
  453 requests in 1.01s, 200.17KB read
  Socket errors: connect 0, read 9, write 0, timeout 0
Requests/sec:    449.85
Transfer/sec:    198.78KB
```

> WireShark抓包印证一下

```
GET /228 HTTP/1.1
Host: baidu.com
X-Counter: 228

...(应答包 省略)
```

```
GET /232 HTTP/1.1
Host: baidu.com
X-Counter: 232

...(应答包 省略)
```

wrk是个非常成功的作品，它的实现更是从多个开源作品中挖掘牛X东西融入自身，如果你每天还在用C/C++，那么wrk的成功，对你应该有绝对的借鉴意义，多抬头，多看牛X代码，我们绝对可以创造奇迹。

引用wrk官方结尾：

```
wrk contains code from a number of open source projects including the 'ae
'
 event loop from redis, the nginx/joyent/node.js 'http-parser', and Mike
 Pall's LuaJIT.
```

持续集成

持续集成

我们做的还不够好，先占个坑。

欢迎贡献章节。

灰度发布

灰度发布

我们做的还不够好，先占个坑。

欢迎贡献章节。

web服务

web服务

API的设计

API的设计

OpenResty，最擅长的应用场景之一就是API Server。如果我们只有简单的几个API出口、入口，那么我们可以相对随意简单一些。

> 举例几个简单API接口输出：

```
server {
    listen       80;
    server_name  localhost;

    location /app/set {
        content_by_lua "ngx.say('set data')";
    }

    location /app/get {
        content_by_lua "ngx.say('get data')";
    }

    location /app/del {
        content_by_lua "ngx.say('del data')";
    }
}
```

当你的API Server接口服务比较多，那么上面的方法显然不适合我们（太啰嗦）。这里推荐一下REST风格。

什么是REST
从资源的角度来观察整个网络，分布在各处的资源由URI确定，而客户端的应用通过URI来获取资源的表示方式。获得这些表徵致使这些应用程序转变了其状态。随着不断获取资源的表示方式，客户端应用不断地在转变着其状态，所谓表述性状态转移（Representational State Transfer）。

这一观点不是凭空臆造的，而是通过观察当前Web互联网的运作方式而抽象出来的。Roy Fielding 认为，

> 设计良好的网络应用表现为一系列的网页，这些网页可以看作的虚拟的状态机，用户选择这些链接导致下一网页传输到用户端展现给使用的人，而这正代表了状态的转变。

> REST是设计风格而不是标准。

REST通常基于使用HTTP，URI，和XML以及HTML这些现有的广泛流行的协议和标准。

- 资源是由URI来指定。
- 对资源的操作包括获取、创建、修改和删除资源，这些操作正好对应HTTP协议提供的GET、POST、PUT和DELETE方法。
- 通过操作资源的表现形式来操作资源。
- 资源的表现形式则是XML或者HTML，取决于读者是机器还是人，是消费web服务的客户软件还是web浏览器。当然也可以是任何其他的格式。

> REST的要求

- 客户端和服务器结构
- 连接协议具有无状态性
- 能够利用Cache机制增进性能
- 层次化的系统

REST使用举例
按照REST的风格引导，我们有关数据的API Server就可以变成这样。

```
server {
    listen       80;
    server_name  localhost;

    location /app/task01 {
        content_by_lua "ngx.say(ngx.req.get_method() .. ' task01')";
    }
    location /app/task02 {
        content_by_lua "ngx.say(ngx.req.get_method() .. ' task02')";
    }
    location /app/task03 {
        content_by_lua "ngx.say(ngx.req.get_method() .. ' task03')";
    }
}
```

对于`/app/task01`接口，这时候我们可以用下面的方法，完成对应的方法调用。

```
# curl -X GET http://127.0.0.1/app/task01
# curl -X PUT http://127.0.0.1/app/task01
# curl -X DELETE http://127.0.0.1/app/task01
```

还有办法压缩不？
上一个章节，如果task类型非常多，那么后面这个配置依然会随着业务调整而调整。其实每个程序员都有一定的洁癖，是否可以以后直接写业务，而不用每次都修改主配置，万一改错了，服务就起不来了。

引用一下HttpLuaModule官方示例代码。

```
# use nginx var in code path
# WARNING: contents in nginx var must be carefully filtered,
# otherwise there'll be great security risk!
location ~ ^/app/([-_a-zA-Z0-9/]+) {
    set $path $1;
    content_by_lua_file /path/to/lua/app/root/$path.lua;
}
```

这下世界宁静了，每天写Lua代码的同学，再也不用去每次修改Nginx主配置了。有新业务，直接开工。顺路还强制了入口文件命名规则。对于后期检查维护更容易。

REST风格的缺点
需要一定的学习成本，如果你的接口是暴露给运维、售后、测试等不同团队，那么他们经常不去确定当时的`method`。当他们查看、模拟的时候，具有一定学习难度。

REST 推崇使用 HTTP 返回码来区分返回结果, 但最大的问题在于 HTTP 的错误返回码 (4xx 系列为主) 不够多，而且订得很随意。比如用 API 创建一个用户，那么错误可能有：

- 调用格式错误(一般返回 400,405)
- 授权错误(一般返回 403)
- "运行期"错误
- 用户名冲突
- 用户名不合法
- email 冲突
- email 不合法

数据合法性检测

数据合法性检测

对用户输入的数据进行合法性检查，避免错误非法的数据进入服务，这是业务系统最常见的需求。很可惜Lua目前没有特别好的数据合法性检查库。

坦诚我们自己做的也不够好，这里只能抛砖引玉，看看大家是否有更好的办法。

我们有这么几个主要的合法性检查场景：

- JSON数据格式
- 关键字段编码为HEX，长度不定
- TABLE内部字段类型

JSON数据格式
这里主要是`JSON DECODE`时，可能存在Crash的问题。我们已经在json解析的异常捕获一章中详细说明了问题本身，以及解决方法。这里就不再重复。

关键字段编码为HEX，长度不定
todo list:

- 到公司补充一下，需要公共模块代码

TABLE内部字段类型
todo list:

- 到公司补充一下，需要公共模块代码

协议无痛升级

协议无痛升级

使用度最高的通讯协议，一定是`HTTP`了。优点有多少，相信大家肯定有切身体会。我相信每家公司对`HTTP`的使用都有自己的规则，甚至偏好。这东西没有谁对谁错，符合业务需求、量体裁衣是王道。这里我们想通过亲身体会，告诉大家利用好`OpenResty`的一些特性，会给我们带来惊喜。

在产品初期，由于产品初期存在极大不确定性、不稳定性，所以要暴露给开发团队、测试团队完全透明的传输协议，所以我们1.0版本就是一个没有任何处理的明文版本`HTTP+JSON`。但随着产品功能的丰富，质量的逐步提高，具备一定的交付能力，这时候通讯协议必须要升级了。

为了更好的安全、效率控制，我们需要支持压缩、防篡改、防重复、简单加密等特性，为此我们设计了全新2.0通讯协议。如何让这个协议升级无感知、改动少，并且简单呢？

> 1.0明文协议配置

```
location ~ ^/api/([-_a-zA-Z0-9/]+).json {
    content_by_lua_file /path/to/lua/api/$1.lua;
}
```

> 1.0明文协议引用示例：

```
# curl http://ip:port/api/hearbeat.json?key=value -d '...'
```

> 2.0密文协议引用示例：

```
# curl http://ip:port/api/hearbeat.json?key=value&ver=2.0 -d '...'
```

从引用示例中看到，我们的密文协议主要都是在请求`body`中做的处理。最生硬的办法就是我们在每个业务入口、出口分别做协议的解析、编码处理。如果你只有几个API接口，那么直

来直去的修改所有API接口源码，最为直接，容易理解。但如果你需要修改几十个API入口，那就要静下来考虑一下，修改的代价是否完全可控。

最后我们使用了`OpenResty`阶段的概念完成协议的转换。

```
location ~ ^/api/([-_a-zA-Z0-9/]+).json {
    access_by_lua_file  /path/to/lua/api/protocal_decode.lua;
    content_by_lua_file /path/to/lua/api/$1.lua;
    body_filter_by_lua_file  /path/to/lua/api/protocal_encode.lua;
}
```

> 内部处理流程说明

- `Nginx`中这三个阶段的执行顺序：access --> content --> body_filter；
- access_by_lua_file：获取协议版本 --> 获取body数据 --> 协议解码 --> 设置body数据；
- content_by_lua_file：正常业务逻辑处理，零修改；
- body_filter_by_lua_file：判断协议版本 --> 协议编码。

刚好前些日子春哥公开了一篇`Github`中引入了`OpenResty`解决SSL证书的问题，他们的解决思路和我们差不多。都是利用access阶段做一些非标准HTTP(S)上的自定义修改，但对于已有业务是不需要任何感知的。

我们这个通讯协议的无痛升级，实际上是有很多玩法可以实现，如果我们的业务从一开始有个相对稳定的框架，可能完全不需要操这个心。没有引入框架，一来是现在没有哪个框架比较成熟，而来是从基础开始更容易摸到细节。对于目前`OpenResty`可参考资料少的情况下，我们更倾向于从最小工作集开始，减少不确定性、复杂度。

也许在后面，我们会推出我们的开发框架，用来统一规避现在碰到的问题，提供完整、可靠、高效的解决方法，我们正在努力ing，请大家拭目以待。

代码规范

代码规范

其实选择 `OpenResty` 的同学，应该都是对执行性能、开发效率比较在乎的，而对于代码风格、规范等这些 `小事` 不太在意。作为一个从Linux C/C++转过来的研发，脚本语言的开发速度，接近C/C++的执行速度，在我轻视了代码规范后，一个BUG的发生告诉我，没规矩不成方圆。

既然我们玩的是 `OpenResty` ，那么很自然的联想到， `OpenResty` 自身组件代码风格是怎样的呢？

> lua-resty-string 的 string.lua

```
local ffi = require "ffi"
local ffi_new = ffi.new
local ffi_str = ffi.string
local C = ffi.C
local setmetatable = setmetatable
local error = error
local tonumber = tonumber

local _M = { _VERSION = '0.09' }

ffi.cdef[[
typedef unsigned char u_char;

u_char * ngx_hex_dump(u_char *dst, const u_char *src, size_t len);

intptr_t ngx_atoi(const unsigned char *line, size_t n);
]]

local str_type = ffi.typeof("uint8_t[?]")

function _M.to_hex(s)
    local len = #s * 2
    local buf = ffi_new(str_type, len)
    C.ngx_hex_dump(buf, s, #s)
    return ffi_str(buf, len)
end

function _M.atoi(s)
    return tonumber(C.ngx_atoi(s, #s))
end

return _M
```

代码虽短，但我们可以从中获取很多信息：

1. 没有全局变量，所有的变量均使用`local`限制作用域
2. 提取公共函数到本地变量，使用本地变量缓存函数指针，加速下次使用
3. 函数名称全部小写，使用下划线进行分割
4. 两个函数之间距离两个空行

这里的第2条，是有争议的。当你按照这个方式写业务的时候，会有些痛苦。因为我们总是把标准API命名成自己的别名，不同开发协作人员，命名结果一定不一样，最后导致同一个标准API在不同地方变成不同别名，会给开发造成极大困惑。

因为这个可预期的麻烦，我们没有遵循第2条标准，尤其是具体业务上游模块。但对于被调用的次数比较多基础模块，可以使用这个方式进行调优。其实这里最好最完美的方法，应该是Lua编译成Luac的时候，直接做Lua Byte Code的调优，直接隐藏这个简单的处理逻辑。

有关更多代码细节，其实我觉得主要还是多看写的漂亮的代码，一旦看他们看的顺眼、形成习惯，那么就很自然能写出风格一致的代码。规定的条条框框死记硬背总是很不爽的，所以多去看看春哥开源的`resty`系列代码，顺手品一品一下不同组件的玩法也别有一番心得。

说说我上面提及的因为风格问题造出来的坑吧。

```
local
function test()
    -- do something
end

function test2()
    -- do something
end
```

这是我当时不记得从哪里看到的一个`Lua`风格，在被引入项目初期，自我感觉良好。可突然从某个时间点开始，新合并进来的代码无法正常工作。查看最后的代码发现原来是`test()`函数作废，被删掉，手抖没有把上面的`local`也删掉。这个隐形的`local`就作用到了下一个函数，最终导致异常。

连接池

连接池

作为一个专业的服务端开发工程师，我们必须要对连接池、线程池、内存池等有较深理解，并且有自己熟悉的库函数可以让我们轻松驾驭这些不同的`池子`。既然他们都叫某某池，那么他们从基础概念上讲，原理和目的几乎是一样的，那就是`复用`。

以连接池做引子，我们说说服务端工程师基础必修课。

从我们应用最多的HTTP连接、数据库连接、消息推送、日志存储等，所有点到点之间，都需要花样繁多的各色连接。为了传输数据，我们需要完成创建连接、收发数据、拆除连接。对并发量不高的场景，我们为每个请求都完整走这三步（短连接），开发工作基本只考虑业务即可，基本上也不会有什么问题。一旦挪到高并发应用场景，那么可能我们就要郁闷了。

你将会碰到下面几个常见问题：

- 性能普遍上不去
- CPU大量资源被系统消耗
- 网络一旦抖动，会有大量TIME_WAIT产生，不得不定期重启服务或定期重启机器
- 服务器工作不稳定，QPS忽高忽低

这时候我们可以优化的第一件事情就是把短链接改成长连接。也就是改成创建连接、收发数据、收发数据...拆除连接，这样我们就可以减少大量创建连接、拆除连接的时间。从性能上来说肯定要比短链接好很多。但这里还是有比较大的浪费。

举例：请求进入时，直接分配数据库长连接资源，假设有80%时间在与关系型数据库通讯，20%时间是在与Nosql数据库通讯。当有50K个并行请求时，后端要分配50K*2=100K的长连接支撑请求。无疑这时候系统压力是非常大的。数据库在牛逼也抵不住滥用不是？

连接池终于要出厂了，它的解决思路是先把所有长连接存起来，谁需要使用，从这里取走，干完活立马放回来。那么按照这个思路，刚刚的50K的并发请求，最多占用后端50K的长连接就够了。省了一半啊有木有？

在OpenResty中，所有具备set_keepalive的类、库函数，说明他都是支持连接池的。

来点代码，给大家提提神，看看连接池使用时的一些注意点，麻雀虽小，五脏俱全。

```
server {
    location /test {
        content_by_lua '
            local redis = require "resty.redis"
            local red = redis:new()

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

            -- red:set_keepalive(10000, 100)       -- 坑①

            ok, err = red:set("dog", "an animal")
            if not ok then
                -- red:set_keepalive(10000, 100)   -- 坑②
                return
            end

            -- 坑③
            red:set_keepalive(10000, 100)
        ';
    }
}
```

- 坑①：只有数据传输完毕了，才能放到池子里，系统无法帮你自动做这个事情
- 坑②：不能把状态位置的连接放回池子里，你不知道这个连接后面会触发什么错误
- 坑③：逗你玩，这个不是坑，是正确的

尤其是掉进了第二个坑，你一定会莫名抓狂。不信的话，你就自己模拟试试，老带劲了。

理解了连接池，那么线程池、内存池，就应该都明白了，只是存放的东西不一样，思想没有任何区别。

c10k编程

c10k编程

比较传统的服务端程序（PHP、FAST CGI等），大多都是通过每产生一个请求，都会有一个进程与之相对应，请求处理完毕后相关进程自动释放。由于进程创建、销毁对资源占用比较高，所以很多语言都通过常驻进程、线程等方式降低资源开销。即使是资源占用最小的线程，当并发数量超过1k的时候，操作系统的处理能力就开始出现明显下降，因为有太多的CPU时间都消耗在系统上下文切换。

由此催生了c10k编程，指的是服务器同时支持成千上万个连接，也就是concurrent 10 000 connection（这也是c10k这个名字的由来）。由于硬件成本的大幅度降低和硬件技术的进步，加上一台服务器同时能够服务更多的客户端，就意味着服务每一个客户端的成本大幅度降低，从这个角度来看，c10k问题显得非常有意义。

理想情况下，具备c10k能力的服务端处理能力是c1k的十倍，返回来说我们可以减少90%的服务器资源，多么诱人的结果。

c10k解决了这几个主要问题：

- 单个进程或线程可以服务于多个客户端请求
- 事件触发替代业务轮询
- IO采用非阻塞方式，减少额外不必要性能损耗

c10k编程的世界，一定是异步编程的世界，他俩绝对是一对儿好基友。服务端一直都不缺乏新秀，各种语言、框架层出不穷。笔者了解的就有OpenResty，Golang，Node.js，Rust，Python(gevent)等。每个语言或解决方案，都有自己完全不同的定位和表现，甚至设计哲学。但是他们从系统底层API应用、基本结构，都是相差不大。这些语言自身的实现机理、运行方式可能差别很大，但只要没有严重的代码错误，他们的性能指标都应该是在同一个级别的。

如果你用了这些解决方案，发现自己的性能非常低，就要好好看看自己是不是姿势有问题。

> c1k --> c10k --> c100k --> ???

人类前进的步伐，没有尽头的，总是在不停的往前跑。c10k的问题，早就被解决，而且方法

还不止一个。目前方案优化手段给力，做到c100k也是可以达到的。后面还有世界么？我们还能走么？

告诉你肯定是有的，那就是c10m。推荐大家了解一下dpdk这个项目，并搜索一些相关领域的知识。要做到c10m，可以说系统网络内核、内存管理，都成为瓶颈了。所以要揭竿起义，统统推到重来。直接操作网卡绕过内核对网络的封装，直接使用用户态内存，再次绕过系统内核。

c10m这个动作比较大，而且还需要特定的硬件型号支持（主要是网卡，网络处理嘛），所以目前这个项目进展还比较缓慢。不过对于有追求的人，可能就要两眼放光了。

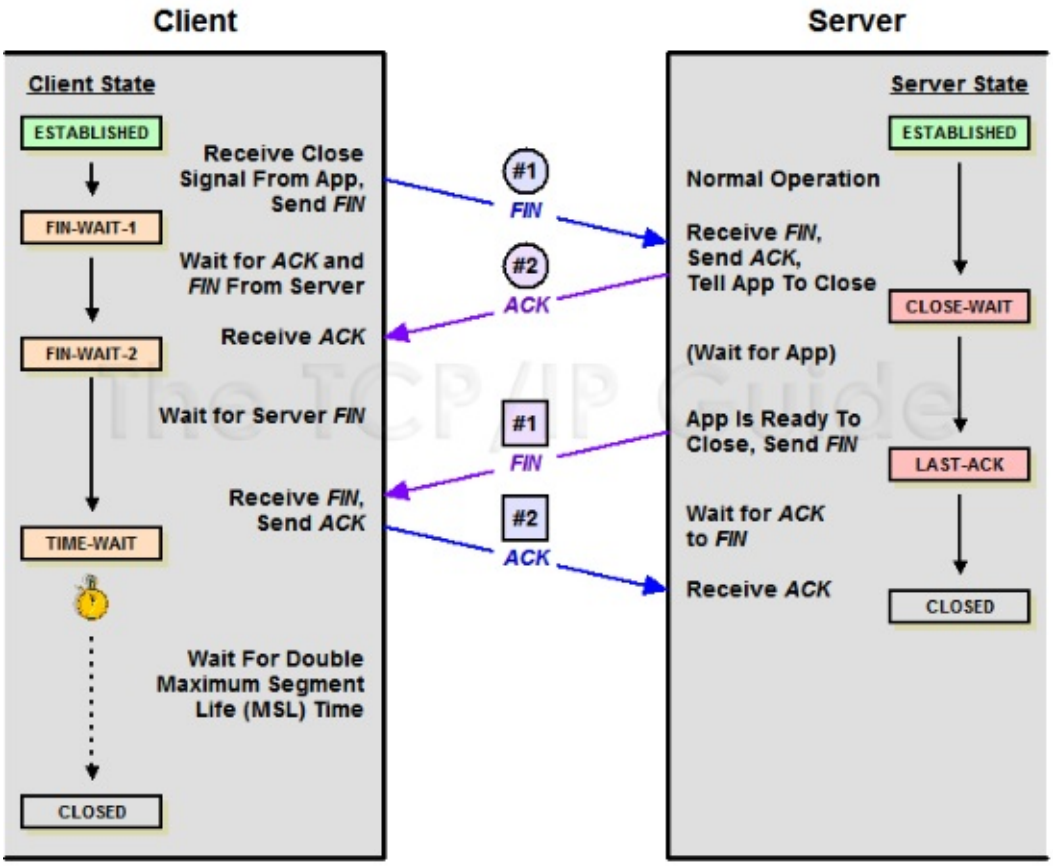
前些日子dpdk组织国内CDN厂商开了一个小会，阿里的朋友说已经用这个开发出了c10m级别的产品。小伙伴们，你们怎么看？心动了，行动不？

TIME_WAIT问题

TIME_WAIT

这个是高并发服务端常见的一个问题，一般的做法是修改sysctl的参数来解决。但是，做为一个有追求的程序猿，你需要多问几个为什么，为什么会出现TIME_WAIT？出现这个合理吗？

我们需要先回顾下tcp的知识，请看下面的状态转换图（图片来自「The TCP/IP Guide」）：



因为TCP连接是双向的，所以在关闭连接的时候，两个方向各自都需要关闭。先发FIN包的一方执行的是主动关闭；后发FIN包的一方执行的是被动关闭。_主动关闭的一方会进入TIMEWAIT状态，并且在此状态停留两倍的MSL时长。

修改sysctl的参数，只是控制TIME_WAIT的数量。你需要很明确的知道，在你的应用场景里面，你预期是服务端还是客户端来主动关闭连接的。一般来说，都是客户端来主动关闭的。

nginx在某些情况下，会主动关闭客户端的请求，这个时候，返回值的connection为close。我们看两个例子：

- http 1.0协议

请求包：

```
GET /hello HTTP/1.0
User-Agent: curl/7.37.1
Host: 127.0.0.1
Accept: */*
Accept-Encoding: deflate, gzip
```

应答包：

```
HTTP/1.1 200 OK
Date: Wed, 08 Jul 2015 02:53:54 GMT
Content-Type: text/plain
Connection: close
Server: 360 web server

hello world
```

对于http 1.0协议，如果请求头里面没有包含connection，那么应答默认是返回Connection: close，也就是说nginx会主动关闭连接。

- user agent
  请求包：

```
POST /api/heartbeat.json HTTP/1.1

Content-Type: application/x-www-form-urlencoded
Cache-Control: no-cache
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT)
Accept-Encoding: gzip, deflate
Accept: */*
Connection: Keep-Alive
Content-Length: 0
```

应答包：

```
HTTP/1.1 200 OK
Date: Mon, 06 Jul 2015 09:35:34 GMT
Content-Type: text/plain
Transfer-Encoding: chunked
Connection: close
Server: 360 web server
Content-Encoding: gzip
```

这个请求包是http1.1的协议，也声明了Connection: Keep-Alive，为什么还会被nginx主动关闭呢？问题出在User-Agent，nginx认为终端的浏览器版本太低，不支持keep alive，所以直接close了。

在我们应用的场景下，终端不是通过浏览器而是后台请求的，而我们也没法控制终端的User-Agent，那有什么方法不让nginx主动去关闭连接呢？可以用keepalive_disable这个参数来解决。这个参数并不是字面的意思，用来关闭keepalive，而是用来定义哪些古代的浏览器不支持keepalive的，默认值是MSIE6。

```
keepalive_disable none;
```

修改为none，就是认为不再通过User-Agent中的浏览器信息，来决定是否keepalive。

注：本文内容参考了火丁笔记和Nginx开发从入门到精通，感谢大牛的分享。

与Docker使用的网络瓶颈

与Docker使用的网络瓶颈

Docker 是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。容器是完全使用沙箱机制，相互之间不会有任何接口（类似 iPhone 的 app）。几乎没有性能开销,可以很容易地在机器和数据中心中运行。最重要的是,他们不依赖于任何语言、框架包括系统。

Docker自2013年以来非常火热，无论是从 github 上的代码活跃度，还是Redhat在RHEL6.5中集成对Docker的支持, 就连 Google 的 Compute Engine 也支持 docker 在其之上运行。

在360企业版安全中，我们使用Docker的原因和目的，可能与其他公司不太一样。我们一直存在比较特殊的"分发"需求，Docker主要是用来屏蔽企业用户平台的不一致性。我们的企业用户使用的系统比较杂，仅仅主流系统就有Ubuntu, Centos，RedHat，AIX，还有一些定制裁减系统等，百花齐放。

虽然OpenResty具有良好的跨平台特性，无奈我们的安全项目比较重，组件比较多，是不可能逐一适配不同平台的，工作量、稳定性等，难度和后期维护复杂度是难以想象的。如果您的应用和我们一样需要二次分发，非常建议考虑使用docker。这个年代是云的时代，二次分发其实成本很高，后期维护成本也很高，所以尽量做到云端。

说说Docker与OpenResty之间的"坑"吧，你们肯定对这个更感兴趣。

我们刚开始使用的时候，是这样启动的：

```
docker run -d -p 80:80 openresty
```

首次压测过程中发现Docker进程CPU占用率100%，单机接口4-5万的QPS就上不去了。经过我们多方探讨交流，终于明白原来是网络瓶颈所致（OpenResty太彪悍，Docker默认的虚拟网卡受不了了 ^_^）。

最终我们绕过这个默认的桥接网卡，使用`--net`参数即可完成。

```
docker run -d -p --net=host 80:80 openresty
```

多么简单，就这么一个参数，居然困扰了我们好几天。一度怀疑我们是否要忍受引入docker带来的低效率网络。所以有时候多出来交流、学习，真的可以让我们学到好多。虽然这个点是我们自己挖出来的，但是在交流过程中还学到了很多好东西。

> Docker Network settings，引自：http://www.lupaworld.com/article-250439-1.html

```
默认情况下，所有的容器都开启了网络接口，同时可以接受任何外部的数据请求。
--dns=[]          : Set custom dns servers for the container
--net="bridge"    : Set the Network mode for the container
                          'bridge': creates a new network stack for the c
ontainer on the docker bridge
                          'none': no networking for this container
                          'container:<name|id>': reuses another container
 network stack
                          'host': use the host network stack inside the c
ontainer
--add-host=""     : Add a line to /etc/hosts (host:IP)
--mac-address="" : Sets the container's Ethernet device's MAC address
```

你可以通过`docker run --net none`来关闭网络接口，此时将关闭所有网络数据的输入输出，你只能通过STDIN、STDOUT或者files来完成I/O操作。默认情况下，容器使用主机的DNS设置，你也可以通过`--dns`来覆盖容器内的DNS设置。同时Docker为容器默认生成一个MAC地址，你可以通过`--mac-address 12:34:56:78:9a:bc`来设置你自己的MAC地址。

Docker支持的网络模式有：

- none。关闭容器内的网络连接
- bridge。通过veth接口来连接容器，默认配置。
- host。允许容器使用host的网络堆栈信息。 注意：这种方式将允许容器访问host中类似D-BUS之类的系统服务，所以认为是不安全的。
- container。使用另外一个容器的网络堆栈信息。

None模式
将网络模式设置为none时，这个容器将不允许访问任何外部router。这个容器内部只会有一个loopback接口，而且不存在任何可以访问外部网络的router。

Bridge模式
Docker默认会将容器设置为bridge模式。此时在主机上面将会存在一个docker0的网络接

口，同时会针对容器创建一对veth接口。其中一个veth接口是在主机充当网卡桥接作用，另外一个veth接口存在于容器的命名空间中，并且指向容器的loopback。Docker会自动给这个容器分配一个IP，并且将容器内的数据通过桥接转发到外部。

Host模式
当网络模式设置为host时，这个容器将完全共享host的网络堆栈。host所有的网络接口将完全对容器开放。容器的主机名也会存在于主机的hostname中。这时，容器所有对外暴露的端口和对其它容器的连接，将完全失效。

Container模式
当网络模式设置为Container时，这个容器将完全复用另外一个容器的网络堆栈。同时使用时这个容器的名称必须要符合下面的格式：--net container:.

比如当前有一个绑定了本地地址localhost的Redis容器。如果另外一个容器需要复用这个网络堆栈，则需要如下操作：

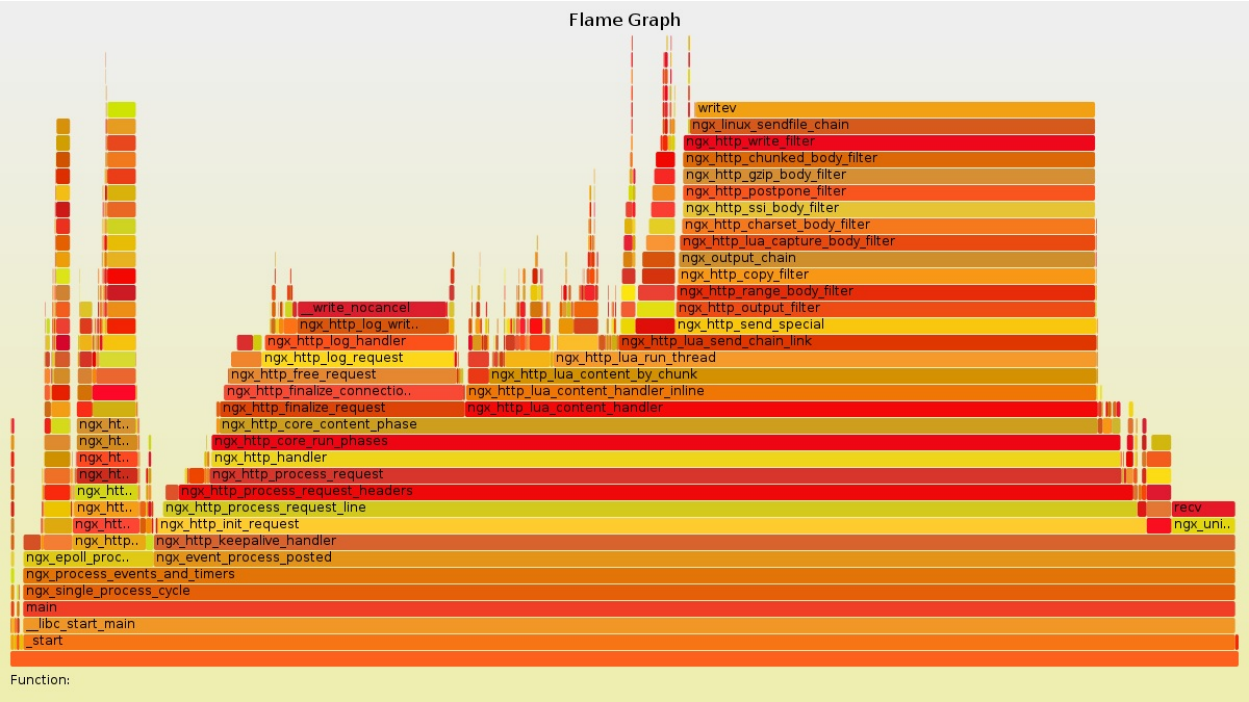
```
$ sudo docker run -d --name redis example/redis --bind 127.0.0.1
$ # use the redis container's network stack to access localhost
$ sudo docker run --rm -ti --net container:redis example/redis-cli -h 127
.0.0.1
```

火焰图

火焰图

火焰图是定位疑难杂症的神器，比如CPU占用高、内存泄漏等问题。特别是Lua级别的火焰图，可以定位到函数和代码级别。

下图来自openresty的官网，显示的是一个正常运行的openresty应用的火焰图，先不用了解细节，有一个直观的了解。



里面的颜色是随机选取的，并没有特殊含义。火焰图的数据来源，是通过systemtap定期收集。

什么时候使用

什么时候使用

一般来说，当发现CPU的占用率和实际业务应该出现的占用率不相符，或者对nginx worker的资源使用率（CPU，内存，磁盘IO）出现怀疑的情况下，都可以使用火焰图进行抓取。另外，对CPU占用率低、吐吞量低的情况也可以使用火焰图的方式排查程序中是否有阻塞调用导致整个架构的吞吐量低下。

关于Github上提供的由perl脚本完成的栈抓取的程序是一个傻瓜化的stap脚本，如果有需要可以自行使用stap进行栈的抓取并生成火焰图，各位看官可以自行尝试。

显示的是什么

显示的是什么

如何安装火焰图生成工具

如何安装火焰图生成工具

安装SystemTap

> 环境 CentOS 6.5 2.6.32-504.23.4.el6.x86_64

SystemTap是一个诊断Linux系统性能或功能问题的开源软件，为了诊断系统问题或性能，开发者或调试人员只需要写一些脚本，然后通过SystemTap提供的命令行接口就可以对正在运行的内核进行诊断调试。

首先需要安装内核开发包和调试包（这一步非常重要并且最为繁琐）：

```
# #Installaion:
# rpm -ivh kernel-debuginfo-($version).rpm
# rpm -ivh kernel-debuginfo-common-($version).rpm
# rpm -ivh kernel-devel-($version).rpm
```

其中$version使用linux命令 uname -r 查看，需要保证内核版本和上述开发包版本一致才能使用systemtap。(下载)

安装systemtap：

```
# yum install systemtap
# ...
# 测试systemtap安装成功否：
# stap -v -e 'probe vfs.read {printf("read performed\n"); exit()}'

Pass 1: parsed user script and 103 library script(s) using 201628virt/295
08res/3144shr/26860data kb, in 10usr/190sys/219real ms.
Pass 2: analyzed script: 1 probe(s), 1 function(s), 3 embed(s), 0 global(
s) using 296120virt/124876res/4120shr/121352data kb, in 660usr/1020sys/18
89real ms.
Pass 3: translated to C into "/tmp/stapffFP7E/stap_82c0f95e47d351a956e158
7c4dd4cee1_1459_src.c" using 296120virt/125204res/4448shr/121352data kb,
in 10usr/50sys/56real ms.
Pass 4: compiled C into "stap_82c0f95e47d351a956e1587c4dd4cee1_1459.ko" i
n 620usr/620sys/1379real ms.
Pass 5: starting run.
read performed
Pass 5: run completed in 20usr/30sys/354real ms.
```

如果出现如上输出表示安装成功。

火焰图绘制
首先，需要下载ngx工具包：Github地址，该工具包即是用perl生成stap探测脚本并运行的脚本，如果是要抓lua级别的情况，请使用工具 ngx-sample-lua-bt

```
# ps -ef | grep nginx   (ps：得到类似这样的输出，其中15010即使worker进程的pid，后面需要用到)
hippo    14857     1  0 Jul01 ?        00:00:00 nginx: master process /opt/openresty/nginx/sbin/nginx -p /home/hippo/skylar_server_code/nginx/main_server/ -c conf/nginx.conf
hippo    15010 14857  0 Jul01 ?        00:00:12 nginx: worker process
# ./ngx-sample-lua-bt -p 15010 --luajit20 -t 5 > tmp.bt  (-p 是要抓的进程的pid --luajit20|--luajit51 是luajit的版本 -t是探测的时间，单位是秒，  探测结果输出到tmp.bt)
# ./fix-lua-bt tmp.bt > flame.bt  (处理ngx-sample-lua-bt的输出，使其可读性更佳)
```

其次，下载Flame-Graphic生成包：Github地址,该工具包中包含多个火焰图生成工具，其中，stackcollapse-stap.pl才是为SystemTap抓取的栈信息的生成工具

```
# stackcollapse-stap.pl flame.bt > flame.cbt
# flamegraph.pl flame.cbt > flame.svg
```

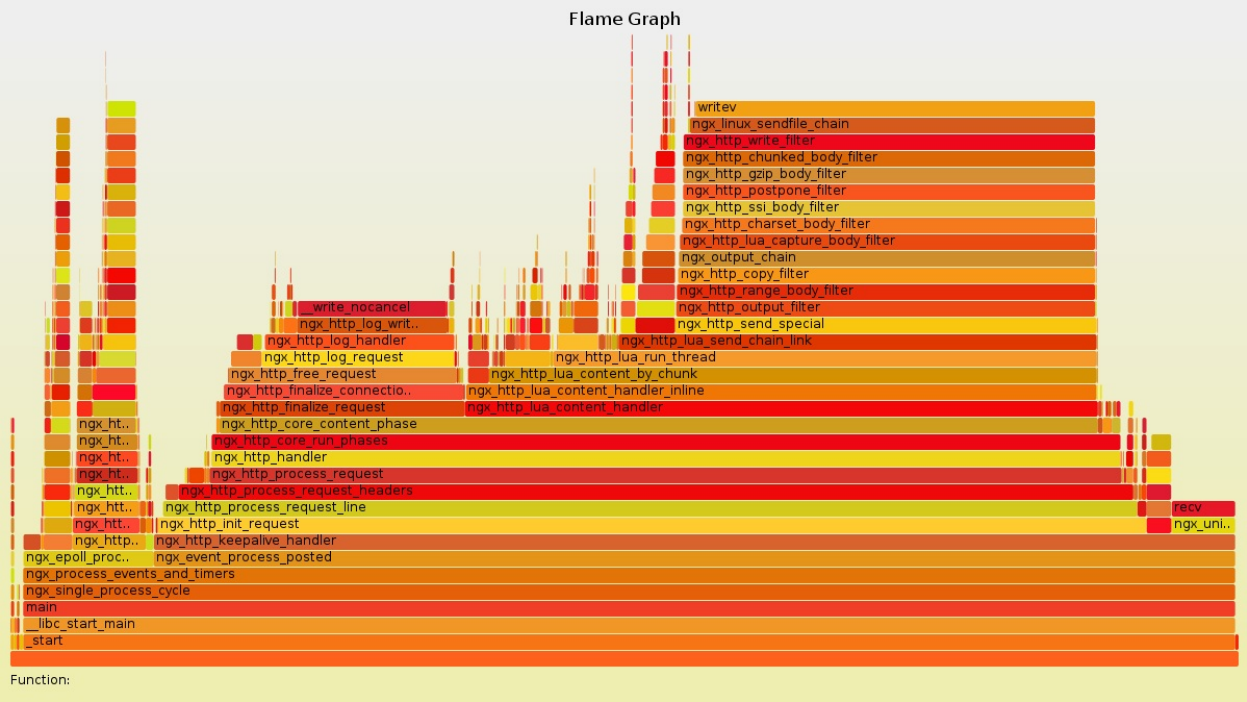
如果一切正常，那么会生成flame.svg，这便是火焰图，用浏览器打开即可。

问题回顾
在整个安装部署过程中，遇到的最大问题便是内核开发包和调试信息包的安装，找不到和内核版本对应的，好不容易找到了又不能下载，@！￥#@……%@#，于是升级了内核，在后面的过程便没遇到什么问题。ps：如果在执行ngx-sample-lua-bt的时间周期内（上面的命令是5秒）,抓取的worker没有任何业务在跑，那么生成的火焰图便没有业务内容，不要惊讶哦~

如何定位问题

如何定位问题

一个正常的火焰图，应该呈现出如官网给出的样例（官网的火焰图是抓C级别函数）：



从上图可以看出，正常业务下的火焰图形状类似的“山脉”，“山脉”的“海拔”表示worker中业务函数的调用深度，“山脉”的“长度”表示worker中业务函数占用cpu的比例。

下面将用一个实际应用中遇到问题抽象出来的示例（CPU占用过高）来说明如何通过火焰图定位问题。

问题表现，nginx worker运行一段时间后出现CPU占用100%的情况，reload后一段时间后复现，当出现CPU占用率高情况的时候是某个worker 占用率高。

问题分析，单worker cpu高的情况一定是某个input中包含的信息不能被lua函数以正确地方式处理导致的，因此上火焰图找出具体的函数，抓取的过程需要抓取C级别的函数和lua级别的函数，抓取相同的时间，两张图一起分析才能得到准确的结果。

抓取步骤：

1. 安装SystemTap;
2. 获取CPU异常的worker的进程ID；

```
ps -ef | grep nginx
```

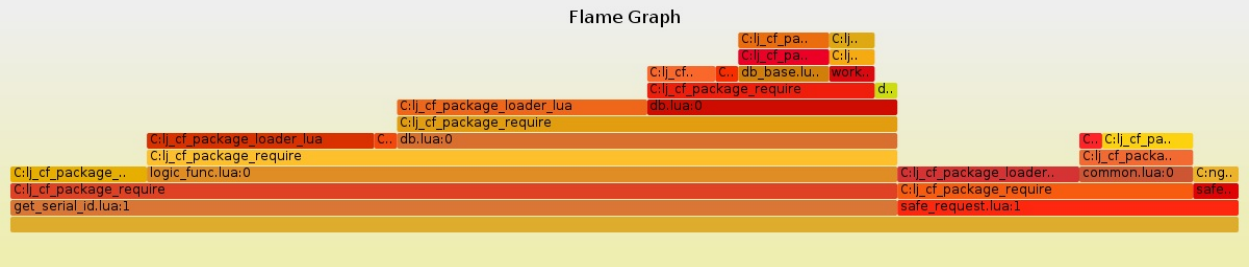
1. 使用ngx-sample-lua-bt抓取栈信息,并用fix-lua-bt工具处理；

```
./ngx-sample-lua-bt -p 9768 --luajit20 -t 5 > tmp.bt
./fix-lua-bt tmp.bt > a.bt
```

1. 使用stackcollapse-stap.pl和；

```
./stackcollapse-stap.pl a.bt > a.cbt
./flamegraph.pl a.cbt > a.svg
```

1. a.svg即是火焰图，拖入浏览器即可：



1. 从上图可以清楚的看到get_serial_id这个函数占用了绝大部分的CPU比例，问题的排查可以从这里入手，找到其调用栈中异常的函数。

ps：一般来说一个正常的火焰图看起来像一座座连绵起伏的“山峰”，而一个异常的火焰图看起来像一座“平顶山”。