OpenResty最佳实践

极客学院出版

前言

在 2012 年的时候，我(作者)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，并不会对 OpenResty 做基础的介绍。想了解基础的同学，请不要看书，而是马上安装 OpenResty ，把官方网站的 Presentations 浏览和实践几遍。

希望你能 enjoy OpenResty 之旅!

适用人群

本课程适用于那些对 OpenResty 这个框架感兴趣，希望在 Go、node.js 之外，多了一个选择并希望获得实战经验的开发人员。

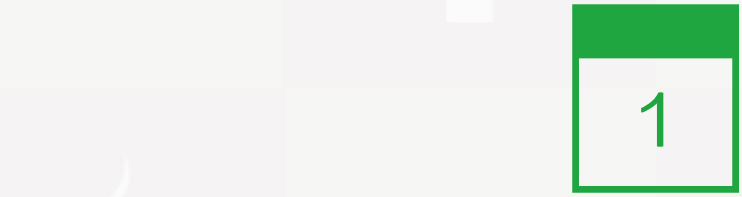
学习前提

我们假定你在学习本课程之前对 Nginx 和 Lua 有基本的了解，最好是之前听说过 OpenResty 框架。

> 说明: 本课程遵守 Common Creative 协议。本书源码在 Github 上维护，欢迎参与：GitHub 地址。也可以加入 QQ 群来和我们交流：

图片 .1 openresty 技术交流群

目录



LuaRestyRedisLibrary

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

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

-- 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("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)

-- 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，那么我们对这段代码不会有任何感觉。如果你的项目重度依赖 Redis ，每次都要把创建连接、建立连接、数据操作、关闭连接（放到连接池）这个完整的链路走完，甚至还要考虑不同的 return 情况，就很快发现代码看上去很不美。

也许我们期望的代码应该是这个样子：

```
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释放 * 默认 redis 数据库地址允许自定义 * 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 * 要具备支持 pipeline 的场景

不卖关子，只要干货，我们二次封装代码是这样干的：

```
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

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 )
    return redis:set_keepalive(60000, 1000) -- put it into the connection pool of size 100, with 60 seconds max idle time
```

```
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
```

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

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。

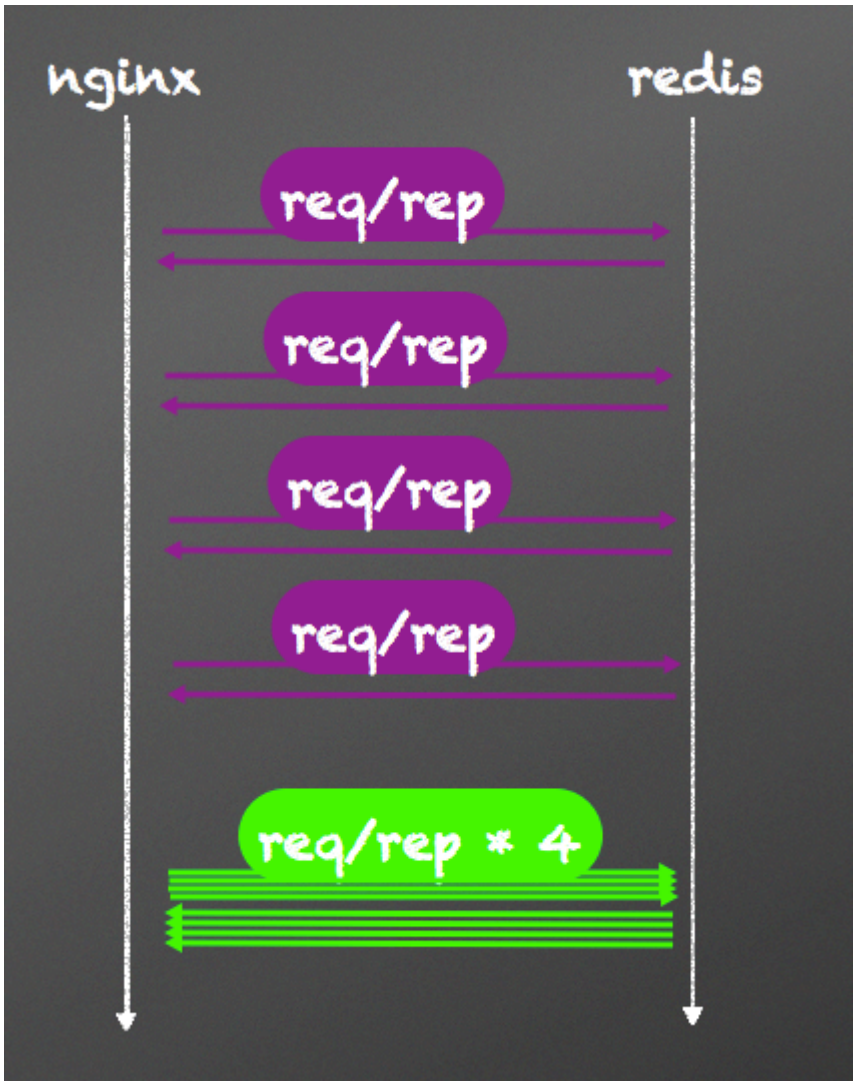
为什么呢？大家知道每发送一个 TCP 报文，会存在网络延时及操作系统的处理延时。大部分情况下，网络延时要远大于 CPU 的处理延时。如果一个简单的命令就以一个 TCP 报文发出，网络延时将成为系统性能瓶颈，使得服务端的并发数量上不去。

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：



图片 1.1 请求示意图

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.sock")

        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.sock")

        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, ": ", res[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

从 [pipeline 那一章节href="https://github.com/moonbingbing/openresty-best-practices/blob/master/redis/pipeline.md")，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。

2

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

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-memcache
stack traceback:
coroutine 0:
    [C]: in function 'decode'
    ...ork/git/github.com/lua-resty-memcached-server/t/test.lua:8: in function <...ork/git/github.com/lua-resty-memcached
```

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

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-memcache
stack traceback:
coroutine 0:
    [C]: in function 'encode'
    ...ork/git/github.com/lua-resty-memcached-server/t/test.lua:13: in function <...ork/git/github.com/lua-resty-memcache
```

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。

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
  --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(

    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

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 "cjson")

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

return _M
```

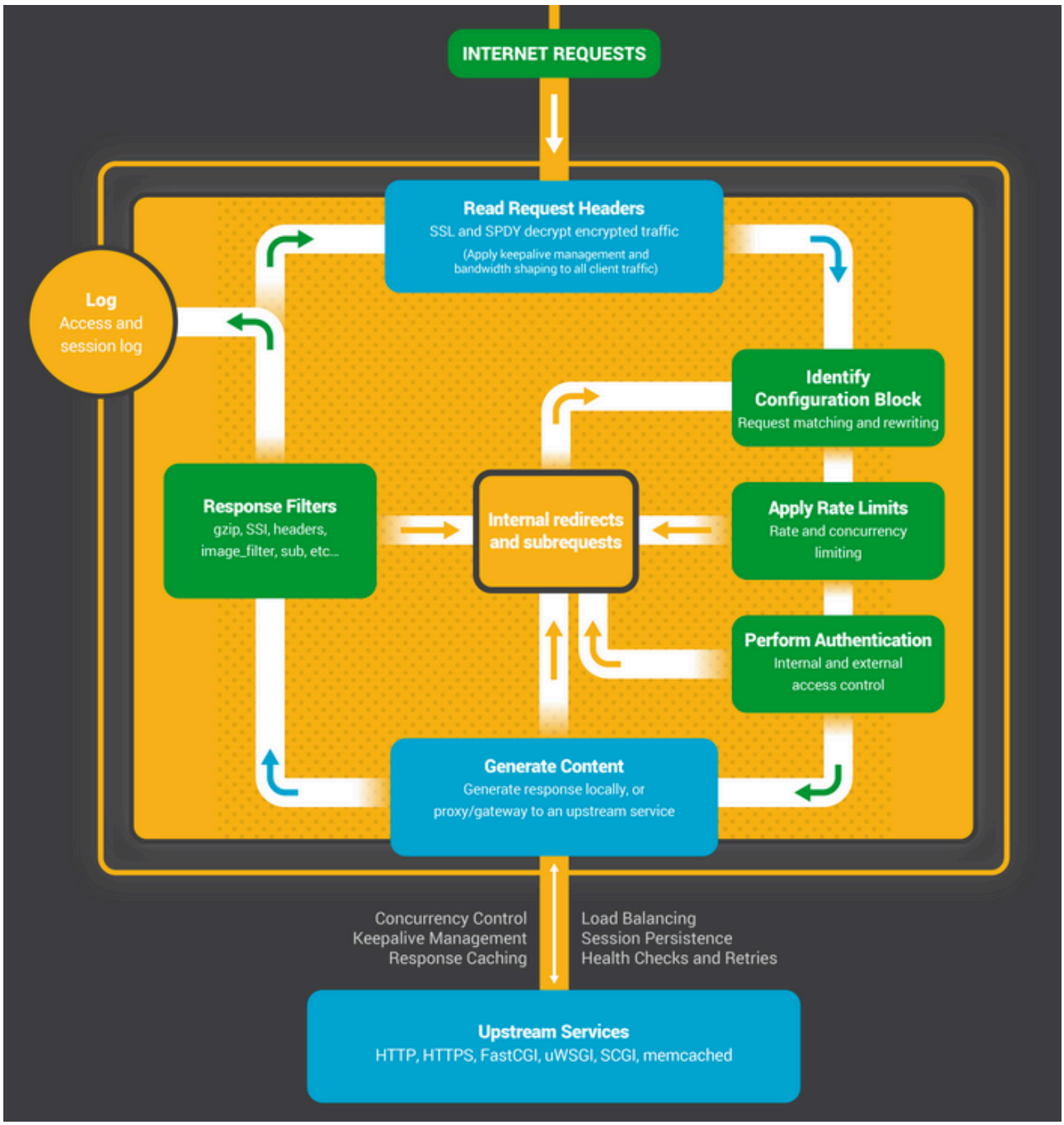
在我们的应用中，对于操作系统版本差异、操作系统位数差异、同时支持不通数据库使用等，几乎都是使用这个方法完成的，十分值得推荐。

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 decode json 对象来源，就可以改成这个库。世界总是有新鲜玩意，多了解多发发现，然后再充实自己吧。

LuaNginxModule

执行阶段概念

Nginx 处理一个请求，它的处理流程请参考下图：



图片 3.1 nginx_internet_request

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

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

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](https://groups.google.com/forum/#!searchin/openresty/package.loaded/openresty/-MZ9 AzXaaG8/TeXTyLCuoYUJ)）：

- 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](https://groups.google.com/forum/#!searchin/openresty/%E5%8 A%A8%E6%80%81%E5%8 A%A0%E8%BD%BDlua%E8%84%9 A%E6%9 C%AC/openresty/-MZ9 AzXaaG8/TeXTyLCuoYUJ)

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

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 128 m;
```

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 process:
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?

在性能上，两个并没有什么差异，都是在 Nginx 的进程中获取到缓存，这都比从本机的 memcached 或者 Redis 里面获取，要快很多。

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

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。

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 2001:0 db8::/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， 2001:0 db8::/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 level
  -- 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

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_status", 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

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。

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

> 本章内容来自 openresty 讨论组 [这里](https://groups.google.com/forum/#!topic/openresty/3 ylMdtvUJqg)

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

> 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 10 s;
    limit_conn perserver 100;
    limit_rate 0 k;

    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

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 10 s;
    access_by_lua 'ngx.var.limit_rate = "300 K"';

    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 specified in bytes per second. The zero value disables rate
```

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.spawn) #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.spawn) #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 ngx_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.worker s()。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](https://github.com/membphis/lua-nginx-module/commit/9 d991677 c090 e1 f 86 fa5840 b19 e02 e56 a4 a17 f86)

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
```

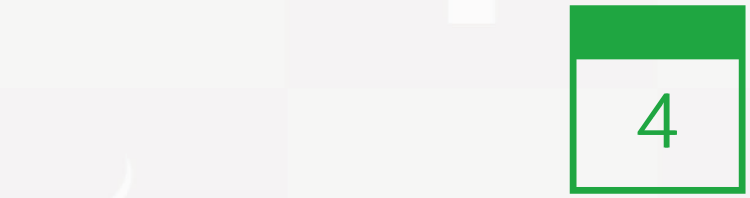
以上代码在测试中发现，应该得到 get(key)的返回值有一定几率为 key 。

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。



LuaRestyDNSLibrary

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。

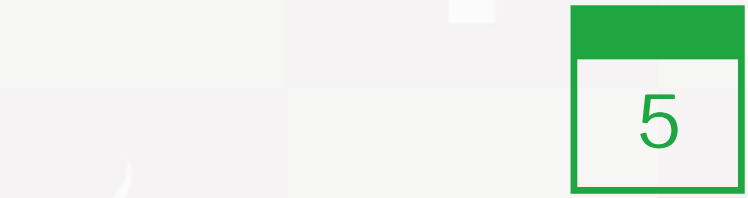
使用 resty.http ，目前这个库只支持 ip： port 的方式定义 url ，其内部实现并没有支持 domain 解析。 resty.http 是支持 set_keepalive 完成长连接，这样我们只需要让他支持 dns 解析就能有完美解决方案了。

```
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。



LuaRestyLock

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。

6

Lua

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

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
```

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：

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

while i <=x do
  local x = i * 2   --while 循环体中的局部变量
  print(x)         --打印 2, 4, 6, 8, ...
  i = i + 1
end

if i > 20 then
  local x          --then 中的局部变量
  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。

> “尽量使用局部变量”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?

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

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

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()", ""))
```

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， finish
                                -- 值为结束下标
print ( start, finish )                 -- 输出 1   2

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


local _,finish = string.find("hello", "he")   --采用虚变量（即下划线），接收起
                                --始下标值，然后丢弃， finish 接收
                                --结束下标值
print ( finish )                         --输出 2
```

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、函数输入等。

> 在 for 循环中的使用：

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

for i,v in ipairs(table_name) do
   print(i,v)
end

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

> 在函数定义中的使用：

```
local _M = { _VERSION = '0.04' }
local mt = { __index = _M }

function _M.new(_, param)
  local self = {
       param=param
     }
```

```
  return setmetatable(self, mt)
end

-- ...

return _M
```

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

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

> 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 。而逗号操作，只是 内容 的展开。

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。

7

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 （ 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-2 xx 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 1 s test @ http://baidu.com
  2 threads and 10 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    20.44 ms    3.74 ms  34.87 ms   77.48%
    Req/Sec   226.05     42.13   270.00     70.00%
  453 requests in 1.01 s, 200.17 KB read
  Socket errors: connect 0, read 9, write 0, timeout 0
Requests/sec:    449.85
Transfer/sec:    198.78 KB
```

> 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

我们做的还不够好，先占个坑。

欢迎贡献章节。GitHub 地址

灰度发布

我们做的还不够好，先占个坑。

欢迎贡献章节。GitHub 地址

8

Web 服务

c10 k 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 1 k 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。

由此产生了 c10 k 编程，指的是服务器同时支持成千上万个客户端的问题，也就是 concurrent 10 000 connection（这也是 c10 k 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，如果一台服务器同时能够服务更多的客户端，那么也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， c10 k 问题显得非常有意义。

c10 k 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训
- IO 采用非阻塞方式，减少额外不必要轮训

c10 k 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。笔者比较熟悉的就有 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表现。但是他们从系统底层 API 应用上，都是相差不大。每个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。

> c1 k --> c10 k --> c100 k --> ???

人类前进的步伐，永远是没有尽头的，总是在不停的往前追跑。 c10 k 的问题，早就被解决，而且方法还不止一个。目前如果方案优化手段给力，做到 c100 k 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 c10 m 。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 c10 m ，可以说系统网络内核、内存管理，都成为瓶颈了。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，内存从系统中拿过来，自己玩。由于这个动作太大，而且还绑定硬件型号（主要是网卡），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 c10 m 级别的产品。小伙伴们，你们怎么看？心动了，行动不？

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 」）：

tcp

图片 8.1 tcp

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 TIME_WAIT 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。

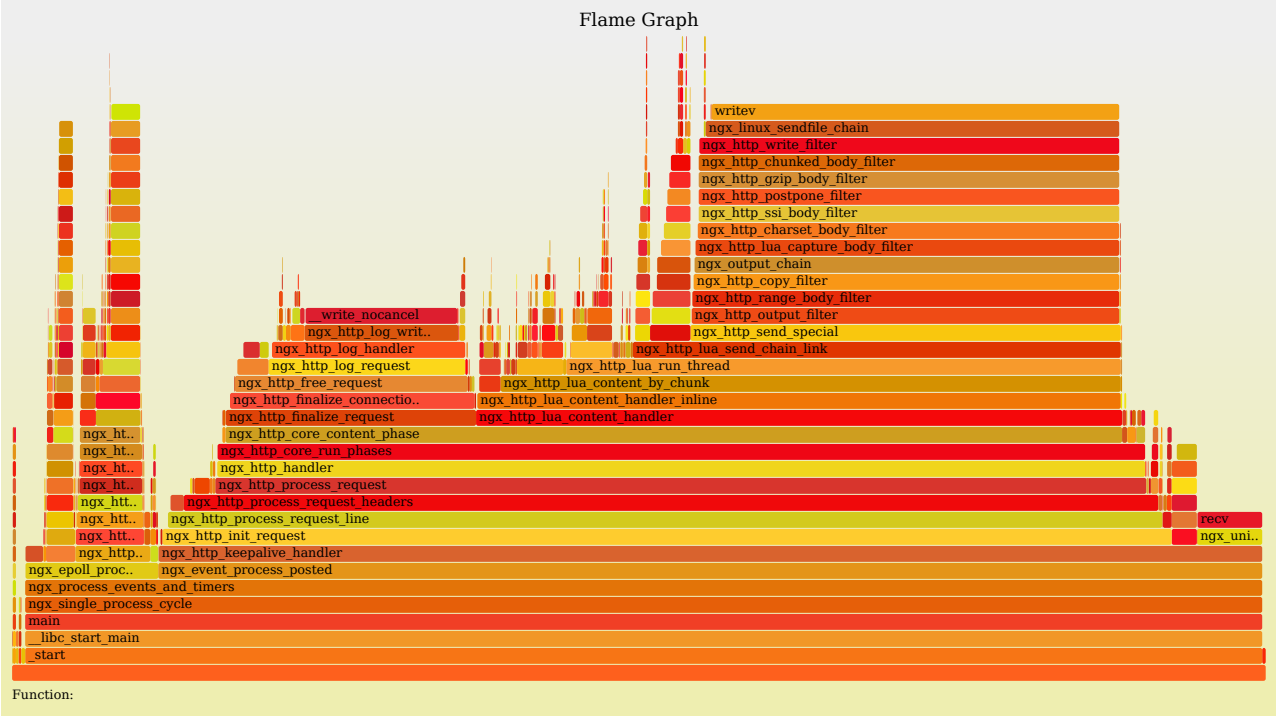
docker

9 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。



图片 9.1 Alt text

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

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。

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 201628 virt/29508 res/3144 shr/26860 data kb, in 10 usr/190 sys/
Pass 2: analyzed script: 1 probe(s), 1 function(s), 3 embed(s), 0 global(s) using 296120 virt/124876 res/4120 shr/121352 da
Pass 3: translated to C into "/tmp/stapffFP7 E/stap_82 c0 f95 e47 d351 a956 e1587 c4 dd4 cee1_1459_src.c" using 29612
Pass 4: compiled C into "stap_82 c0 f95 e47 d351 a956 e1587 c4 dd4 cee1_1459.ko" in 620 usr/620 sys/1379 real ms.
Pass 5: starting run.
read performed
Pass 5: run completed in 20 usr/30 sys/354 real 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_s
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 是抓

# ./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
```

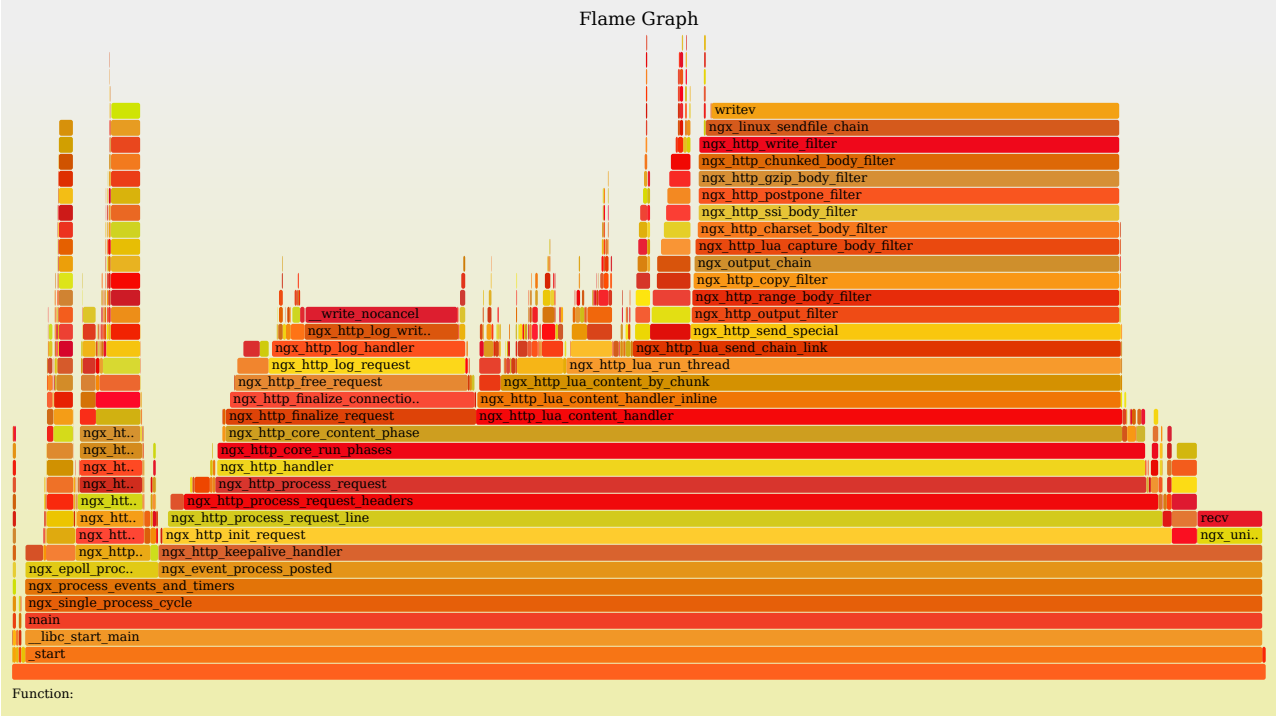
如果一切正常，那么会生成 flame.svg ，这便是火焰图，用浏览器打开即可。

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

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）：



图片 9.2 Alt text

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：

- 安装 SystemTap;
- 获取 CPU 异常的 worker 的进程 ID；

```
ps -ef | grep nginx
```

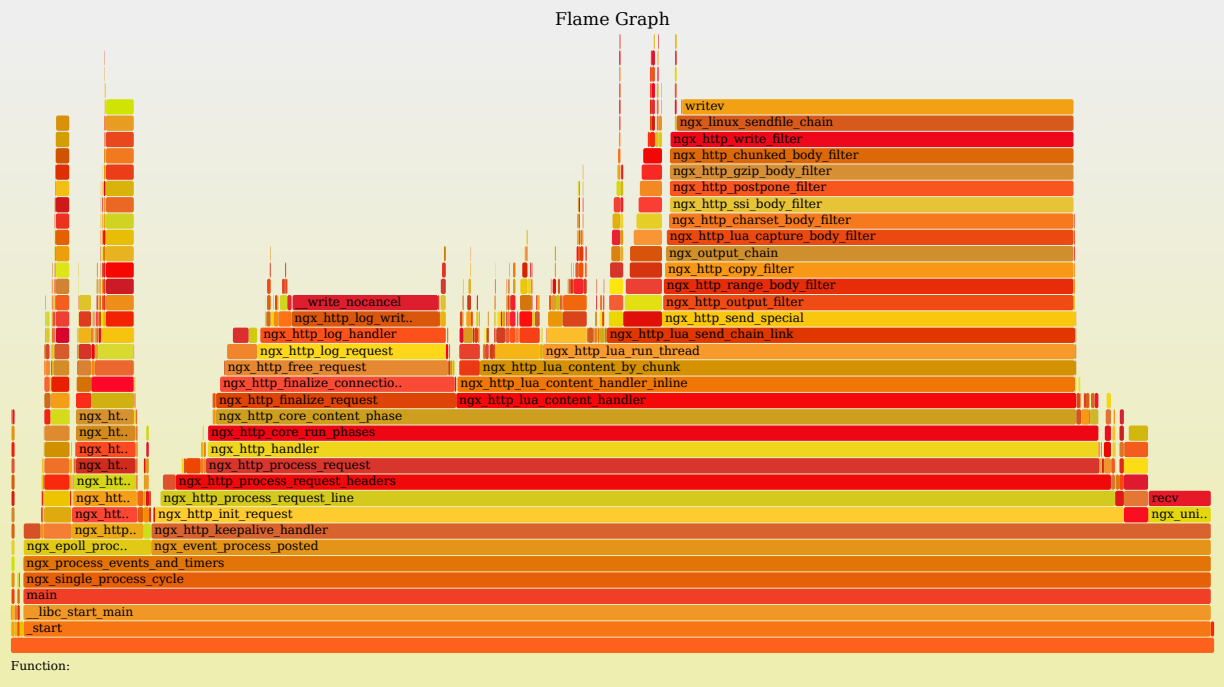
- 使用 ngx-sample-lua-bt抓取栈信息,并用 fix-lua-bt 工具处理；

```
./ngx-sample-lua-bt -p 9768 --luajit20 -t 5 > tmp.bt
./fix-lua-bt tmp.bt > a.bt
```

- 使用 stackcollapse-stap.pl 和；

```
./stackcollapse-stap.pl a.bt > a.cbt
./flamegraph.pl a.cbt > a.svg
```

- a.svg 即是火焰图，拖入浏览器即可



图片 9.3 Alt text

- 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”。

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/openresty-best-practice/