nginx与lua的执行顺序和步骤说明

18,390 views

[2015年2月11日](http://www.mrhaoting.com/?p=157)[技术学习](http://www.mrhaoting.com/?cat=4)[lua](http://www.mrhaoting.com/?tag=lua)、[nginx](http://www.mrhaoting.com/?tag=nginx)

**一、nginx执行步骤**

nginx在处理每一个用户请求时，都是按照若干个不同的阶段依次处理的，与配置文件上的顺序没有关系，详细内容可以阅读《深入理解nginx:模块开发与架构解析》这本书，这里只做简单介绍；

1、post-read

读取请求内容阶段，nginx读取并解析完请求头之后就立即开始运行；

2、server-rewrite

server请求地址重写阶段；

3、find-config

配置查找阶段，用来完成当前请求与location配重块之间的配对工作；

4、rewrite

location请求地址重写阶段，当ngx\_rewrite指令用于location中，就是再这个阶段运行的；

5、post-rewrite

请求地址重写提交阶段，当nginx完成rewrite阶段所要求的内部跳转动作，如果rewrite阶段有这个要求的话；

6、preaccess

访问权限检查准备阶段，ngx\_limit\_req和ngx\_limit\_zone在这个阶段运行，ngx\_limit\_req可以控制请求的访问频率，ngx\_limit\_zone可以控制访问的并发度；

7、access

权限检查阶段，ngx\_access在这个阶段运行，配置指令多是执行访问控制相关的任务，如检查用户的访问权限，检查用户的来源IP是否合法；

8、post-access

访问权限检查提交阶段；

9、try-files

配置项try\_files处理阶段；

10、content

内容产生阶段，是所有请求处理阶段中最为重要的阶段，因为这个阶段的指令通常是用来生成HTTP响应内容的；

11、log

日志模块处理阶段；

**二、ngx\_lua运行指令**

ngx\_lua属于nginx的一部分，它的执行指令都包含在nginx的11个步骤之中了，不过ngx\_lua并不是所有阶段都会运行的；

**1、init\_by\_lua、init\_by\_lua\_file**

语法：init\_by\_lua <lua-script-str>

语境：http

阶段：loading-config

当nginx master进程在加载nginx配置文件时运行指定的lua脚本，通常用来注册lua的全局变量或在服务器启动时预加载lua模块：

init\_by\_lua 'cjson = require "cjson"';

server {

location = /api {

content\_by\_lua '

ngx.say(cjson.encode({dog = 5, cat = 6}))

'

}

}

或者初始化lua\_shared\_dict共享数据：

lua\_shared\_dict dogs 1m;

init\_by\_lua '

local dogs = ngx.shared.dogs;

dogs:set("Tom", 50)

'

server {

location = /api {

content\_by\_lua '

local dogs = ngx.shared.dogs;

ngx.say(dogs:get("Tom"))

'

}

}

但是，lua\_shared\_dict的内容不会在nginx reload时被清除。所以如果你不想在你的init\_by\_lua中重新初始化共享数据，那么你需要在你的共享内存中设置一个标志位并在init\_by\_lua中进行检查。

因为这个阶段的lua代码是在nginx forks出任何worker进程之前运行，数据和代码的加载将享受由操作系统提供的copy-on-write的特性，从而节约了大量的内存。

不要在这个阶段初始化你的私有lua全局变量，因为使用lua全局变量会照成性能损失，并且可能导致全局命名空间被污染。

这个阶段只支持一些小的LUA Nginx API设置：ngx.log和print、ngx.shared.DICT；

**2、init\_worker\_by\_lua、init\_worker\_by\_lua\_file**

语法：init\_worker\_by\_lua <lua-script-str>

语境：http

阶段：starting-worker

在每个nginx worker进程启动时调用指定的lua代码。如果master 进程不允许，则只会在init\_by\_lua之后调用。

这个hook通常用来创建每个工作进程的计时器(通过lua的ngx.timer API)，进行后端健康检查或者其它日常工作：

init\_worker\_by\_lua:

local delay = 3 -- in seconds

local new\_timer = ngx.timer.at

local log = ngx.log

local ERR = ngx.ERR

local check

check = function(premature)

if not premature then

-- do the health check other routine work

local ok, err = new\_timer(delay, check)

if not ok then

log(ERR, "failed to create timer: ", err)

return

end

end

end

local ok, err = new\_timer(delay, check)

if not ok then

log(ERR, "failed to create timer: ", err)

end

**3、set\_by\_lua、set\_by\_lua\_file**

语法：set\_by\_lua $res <lua-script-str> [$arg1 $arg2 …]

语境：server、server if、location、location if

阶段：rewrite

传入参数到指定的lua脚本代码中执行，并得到返回值到res中。<lua-script-str>中的代码可以使从ngx.arg表中取得输入参数(顺序索引从1开始)。

这个指令是为了执行短期、快速运行的代码因为运行过程中nginx的事件处理循环是处于阻塞状态的。耗费时间的代码应该被避免。

禁止在这个阶段使用下面的API：1、output api（ngx.say和ngx.send\_headers）；2、control api（ngx.exit）；3、subrequest api（ngx.location.capture和ngx.location.capture\_multi）；4、cosocket api（ngx.socket.tcp和ngx.req.socket）；5、sleep api（ngx.sleep）

此外注意，这个指令只能一次写出一个nginx变量，但是使用ngx.var接口可以解决这个问题：

location /foo {

set $diff '';

set\_by\_lua $num '

local a = 32

local b = 56

ngx.var.diff = a - b; --写入$diff中

return a + b; --返回到$sum中

'

echo "sum = $sum, diff = $diff";

}

这个指令可以自由的使用HttpRewriteModule、HttpSetMiscModule和HttpArrayVarModule所有的方法。所有的这些指令都将按他们出现在配置文件中的顺序进行执行。

**4、rewrite\_by\_lua、rewrite\_by\_lua\_file**

语法：rewrite\_by\_lua <lua-script-str>

语境：http、server、location、location if

阶段：rewrite tail

作为rewrite阶段的处理，为每个请求执行指定的lua代码。注意这个处理是在标准HtpRewriteModule之后进行的：

location /foo {

set $a 12;

set $b "";

rewrite\_by\_lua 'ngx.var.b = tonumber(ngx.var.a) + 1';

echo "res = $b";

}

如果这样的话将不会按预期进行工作：

location /foo {

set $a 12;

set $b '';

rewrite\_by\_lua 'ngx.var.b = tonumber(ngx.var.a) + 1';

if($b = '13') {

rewrite ^ /bar redirect;

break;

}

echo "res = $b"

}

因为if会在rewrite\_by\_lua之前运行，所以判断将不成立。正确的写法应该是这样：

location /foo {

set $a 12;

set $b '';

rewrite\_by\_lua '

ngx.var.b = tonumber(ngx.var.a) + 1

if tonumber(ngx.var.b) == 13 then

return ngx.redirect("/bar");

end

'

echo "res = $b";

}

注意ngx\_eval模块可以近似于使用rewite\_by\_lua，例如：

location / {

eval $res {

proxy\_pass http://foo,com/check-spam;

}

if($res = 'spam') {

rewrite ^ /terms-of-use.html redirect;

}

fastcgi\_pass .......

}

可以被ngx\_lua这样实现：

location = /check-spam {

internal;

proxy\_pass http://foo.com/check-spam;

}

location / {

rewrite\_by\_lua '

local res = ngx.location.capture("/check-spam")

if res.body == "spam" then

return ngx.redirect("terms-of-use.html")

'

fastcgi\_pass .......

}

和其它的rewrite阶段的处理程序一样，rewrite\_by\_lua在subrequests中一样可以运行。

请注意在rewrite\_by\_lua内调用ngx.exit(ngx.OK)，nginx的请求处理流程将继续进行content阶段的处理。从rewrite\_by\_lua终止当前的请求，要调用ngx.exit返回status大于200并小于300的成功状态或ngx.exit(ngx.HTTP\_INTERNAL\_SERVER\_ERROR)的失败状态。

如果HttpRewriteModule的重写指令被用来改写URI和重定向，那么任何rewrite\_by\_lua和rewrite\_by\_lua\_file的代码将不会执行，例如：

location /foo {

rewrite ^ /bar;

rewrite\_by\_lua 'ngx.exit(503)'

}

location /bar {

.......

}

在这个例子中ngx.exit(503)将永远不会被执行，因为rewrite修改了location，请求已经跳入其它location中了。

**5、access\_by\_lua，access\_by\_lua\_file**

语法：access\_by\_lua <lua-script-str>

语境：http,server,location,location if

阶段：access tail

为每个请求在访问阶段的调用lua脚本进行处理。主要用于访问控制，能收集到大部分的变量。

注意access\_by\_lua和rewrite\_by\_lua类似是在标准HttpAccessModule之后才会运行，看一个例子：

location / {

deny 192.168.1.1;

allow 192.168.1.0/24;

allow 10.1.1.0/16;

deny all;

access\_by\_lua '

local res = ngx.location.capture("/mysql", {...})

....

'

}

如果client ip在黑名单之内，那么这次连接会在进入access\_by\_lua调用的mysql之前被丢弃掉。

ngx\_auth\_request模块和access\_by\_lua的用法类似：

location / {

auth\_request /auth;

}

可以用ngx\_lua这么实现：

location / {

access\_by\_lua '

local res = ngx.location.capture("/auth")

if res.status == ngx.HTTP\_OK then

return

end

if res.status == ngx.HTTP\_FORBIDDEN then

ngx.exit(res.status)

end

ngx.exit(ngx.HTTP\_INTERNAL\_SERVER\_ERROR)

'

}

和其它access阶段的模块一样，access\_by\_lua不会在subrequest中运行。

请注意在access\_by\_lua内调用ngx.exit(ngx.OK)，nginx的请求处理流程将继续进行后面阶段的处理。从rewrite\_by\_lua终止当前的请求，要调用ngx.exit返回status大于200并小于300的成功状态或ngx.exit(ngx.HTTP\_INTERNAL\_SERVER\_ERROR)的失败状态。

**6、content\_by\_lua，content\_by\_lua\_file**

语法：content\_by\_lua <lua-script-str>

语境：location，location if

阶段：content

作为“content handler”为每个请求执行lua代码，为请求者输出响应内容。

不要将它和其它的内容处理指令在同一个location内使用如proxy\_pass。

**7、header\_filter\_by\_lua，header\_filter\_by\_lua\_file**

语法：header\_filter\_by\_lua <lua-script-str>

语境：http，server，location，location if

阶段：output-header-filter

一般用来设置cookie和headers，在该阶段不能使用如下几个API：

1、output API（ngx.say和ngx.send\_headers）

2、control API（ngx.exit和ngx.exec）

3、subrequest API(ngx.location.capture和ngx.location.capture\_multi)

4、cosocket API（ngx.socket.tcp和ngx.req.socket）

有一个例子是 在你的lua header filter里添加一个响应头标头：

location / {

proxy\_pass http://mybackend;

header\_filter\_by\_lua 'ngx.header.Foo = "blah"';

}

**8、body\_filter\_by\_lua，body\_filter\_by\_lua\_file**

语法：body\_filter\_by\_lua <lua-script-str>

语境：http，server，location，location if

阶段：output-body-filter

输入的数据时通过ngx.arg[1](作为lua的string值)，通过ngx.arg[2]这个bool类型表示响应数据流的结尾。

基于这个原因，‘eof’只是nginx的链接缓冲区的last\_buf（对主requests）或last\_in\_chain（对subrequests）的标记。

运行以下命令可以立即终止运行接下来的lua代码：

return ngx.ERROR

这会将响应体截断导致无效的响应。lua代码可以通过修改ngx.arg[1]的内容将数据传输到下游的nginx output body filter阶段的其它模块中去。例如，将response body中的小写字母进行反转，我们可以这么写：

location / {

proxy\_pass http://mybackend;

body\_filter\_by\_lua 'ngx.arg[1] = string.upper(ngx.arg[1])'

}

当将ngx.arg[1]设置为nil或者一个空的lua string时，下游的模块将不会收到数据了。

同样可以通过修改ngx.arg[2]来设置新的”eof“标记，例如：

location /t {

echo hello world;

echo hiya globe;

body\_filter\_by\_lua '

local chunk = ngx.arg[1]

if string.match(chunk, "hello") then

ngx.arg[2] = true --new eof

return

end

--just throw away any remaining chunk data

ngx.arg[1] = nil

'

}

那么GET /t的请求只会回复：hello world

这是因为，当body filter看到了一块包含”hello“的字符块后立即将”eof“标记设置为了true，从而导致响应被截断了但仍然是有效的回复。

当lua代码中改变了响应体的长度时，应该要清除content-length响应头部的值，例如：

location /foo {

header\_filter\_by\_lua 'ngx.header.content\_length = nil'

body\_filter\_by\_lua 'ngx.arg[1] = string.len(ngx.arg[1]) .. "\\n"'

}

在该阶段不能使用如下几个API：

1、output API（ngx.say和ngx.send\_headers）

2、control API（ngx.exit和ngx.exec）

3、subrequest API(ngx.location.capture和ngx.location.capture\_multi)

4、cosocket API（ngx.socket.tcp和ngx.req.socket）

nginx output filters可能会在一次请求中被多次调用，因为响应体可能是以chunks方式传输的。因此这个指令一般会在一次请求中被调用多次。

9、log\_by\_lua，log\_by\_lua\_file

语法：log\_by\_lua <lua-script-str>

语境：http，server，location，location if

阶段：log

在log阶段调用指定的lua脚本，并不会替换access log，而是在那之后进行调用。

在该阶段不能使用如下几个API：

1、output API（ngx.say和ngx.send\_headers）

2、control API（ngx.exit和ngx.exec）

3、subrequest API(ngx.location.capture和ngx.location.capture\_multi)

4、cosocket API（ngx.socket.tcp和ngx.req.socket）

一个收集upstream\_response\_time的平均数据的例子：

lua\_shared\_dict log\_dict 5M

server{

location / {

proxy\_pass http;//mybackend

log\_by\_lua '

local log\_dict = ngx.shared.log\_dict

local upstream\_time = tonumber(ngx.var.upstream\_response\_time)

local sum = log\_dict:get("upstream\_time-sum") or 0

sum = sum + upstream\_time

log\_dict:set("upsteam\_time-sum", sum)

local newval, err = log\_dict:incr("upstream\_time-nb", 1)

if not newval and err == "not found" then

log\_dict:add("upstream\_time-nb", 0)

log\_dict:incr("upstream\_time-nb", 1)

end

'

}

location = /status {

content\_by\_lua '

local log\_dict = ngx.shared.log\_dict

local sum = log\_dict:get("upstream\_time-sum")

local nb = log\_dict:get("upstream\_time-nb")

if nb and sum then

ngx.say("average upstream response time: ", sum/nb, " (", nb, " reqs)")

else

ngx.say("no data yet")

end

'

}

}