PHP程序员应该知道的Nginx(下)

Panda
https://wujunze.com

INDEX

- Nginx运行流程(深入到每个模块)
- Nginx与Lua(OpenResty)的核心原理
- OpenResty为Nginx插上翅膀(赋予Nginx更多可能)
- 开发一个Nginx模块
- 畅想Nginx与Web的未来

NGINX处理请求的几个阶段

- 读取完请求头后，nginx进入请求的处理阶段。简单的情况下，客户端发送过的统一资源定位符(url)对应服务器上某一路径上的资源，web服务器需要做的仅仅是将url映射到本地文件系统的路径，然后读取相应文件并返回给客户端。但这仅仅是最初的互联网的需求，而如今互联网出现了各种各样复杂的需求，要求web服务器能够处理诸如安全及权限控制，多媒体内容和动态网页等等问题。这些复杂的需求导致web服务器不再是一个短小的程序，而变成了一个必须经过仔细设计，模块化的系统。nginx良好的模块化特性体现在其对请求处理流程的多阶段划分当中，多阶段处理流程就好像一条流水线，一个nginx进程可以并发的处理处于不同阶段的多个请求。nginx允许开发者在处理流程的任意阶段注册模块，在启动阶段，nginx会把各个阶段注册的所有模块处理函数按序的组织成一条执行链。

- nginx实际把请求处理流程划分为了11个阶段，这样划分的原因是将请求的执行逻辑细分，各阶段按照处理时机定义了清晰的执行语义，开发者可以很容易分辨自己需要开发的模块应该定义在什么阶段，其定义在https://github.com/nginx/nginx/blob/master/src/http/ngx_http_core_module.h

- NGX_HTTP_POST_READ_PHASE:
  接收完请求头之后的第一个阶段，它位于uri重写之前，实际上很少有模块会注册在该阶段，默认的情况下，该阶段被跳过

- NGX_HTTP_SERVER_REWRITE_PHASE:
  server级别的uri重写阶段，也就是该阶段执行处于server块内，location块外的重写指令，在读取请求头的过程中nginx会根据host及端口找到对应的虚拟主机配置

- NGX_HTTP_FIND_CONFIG_PHASE:
  寻找location配置阶段，该阶段使用重写之后的uri来查找对应的location，值得注意的是该阶段可能会被执行多次，因为也可能有location级别的重写指令

- NGX_HTTP_REWRITE_PHASE:
  location级别的uri重写阶段，该阶段执行location基本的重写指令，也可能会被执行多次

- NGX_HTTP_POST_REWRITE_PHASE:
  location级别重写的后一阶段，用来检查上阶段是否有uri重写，并根据结果跳转到合适的阶段

- NGX_HTTP_PREACCESS_PHASE:
  访问权限控制的前一阶段，该阶段在权限控制阶段之前，一般也用于访问控制，比如限制访问频率，链接数等

- NGX_HTTP_ACCESS_PHASE:
  访问权限控制阶段，比如基于ip黑白名单的权限控制，基于用户名密码的权限控制等

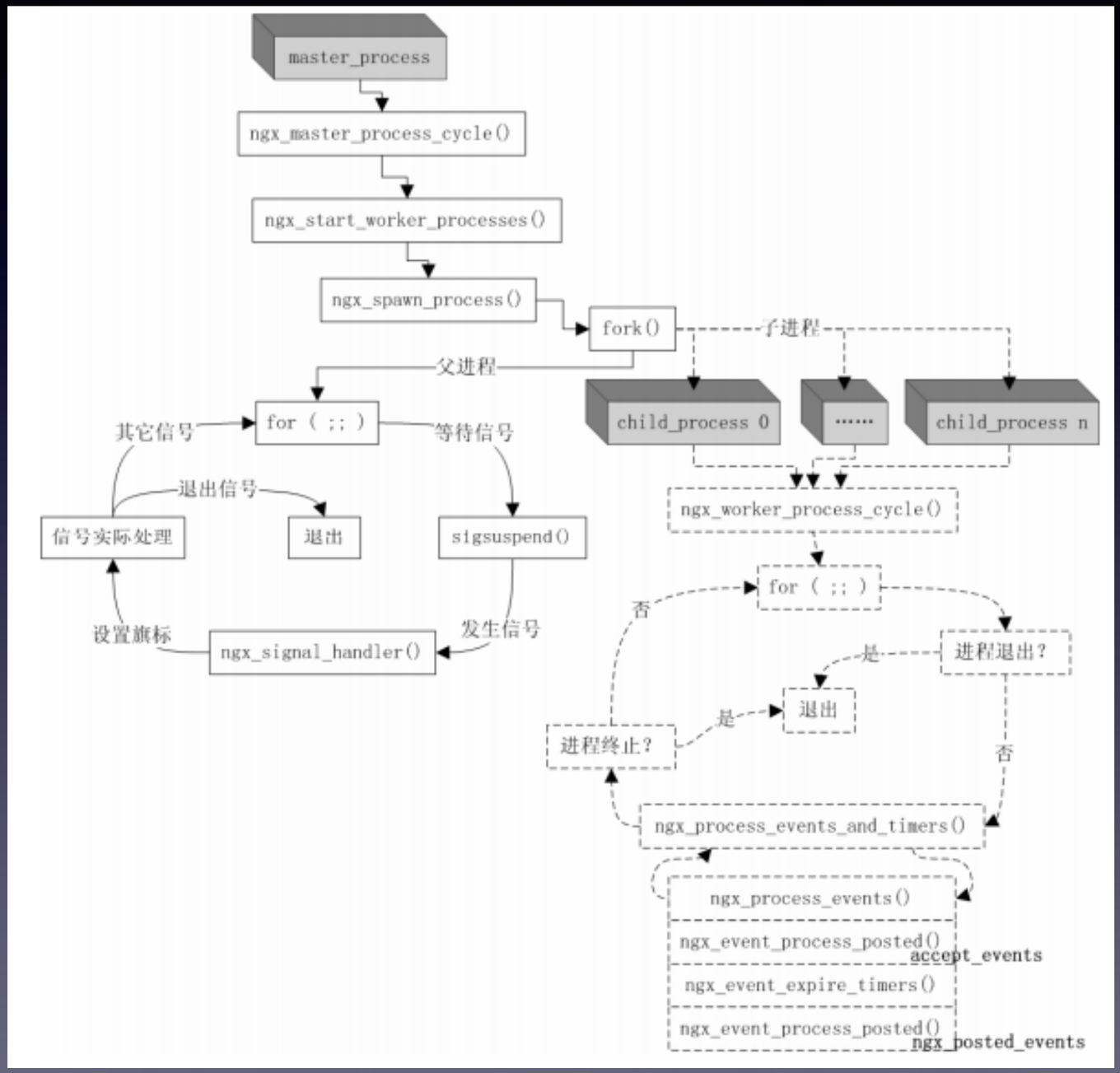
- NGX_HTTP_POST_ACCESS_PHASE:
  问权限控制的后一阶段，该阶段根据权限控制阶段的执行结果进行相应处理

- NGX_HTTP_TRY_FILES_PHASE:
  try_files指令的处理阶段，如果没有配置try_files指令，则该阶段被跳过

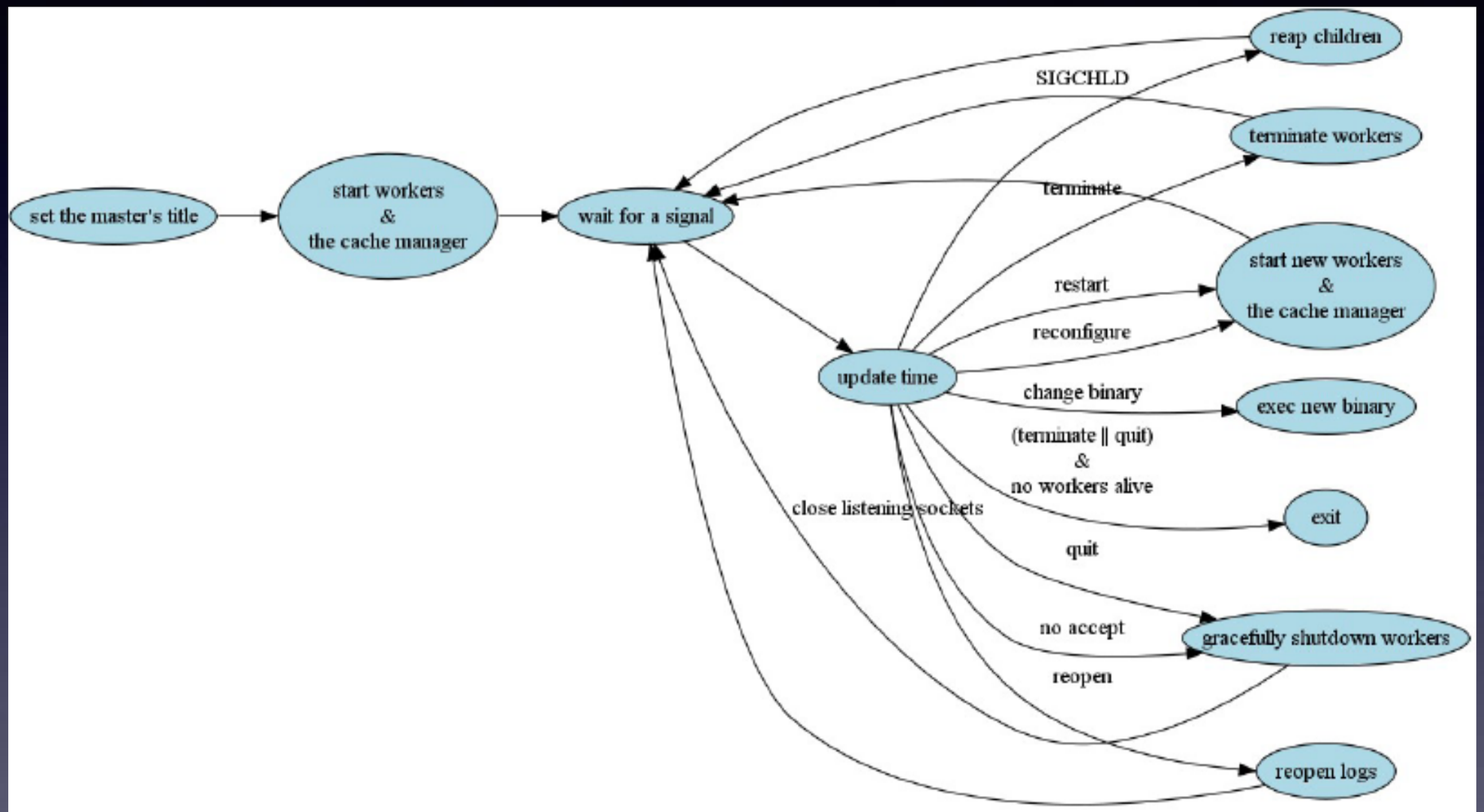
- NGX_HTTP_CONTENT_PHASE:
  内容生成阶段，该阶段产生响应，并发送到客户端

- NGX_HTTP_LOG_PHASE:
  日志记录阶段，该阶段记录访问日志
  请求初始化处理是在http/ngx_http.c里的ngx_http_block()函数里执行，然后会依次创建初始化各个阶段的配置(main_conf,server_conf,loc_conf),初始化各个阶段的handler,ngx_http_init_phase_handlers()

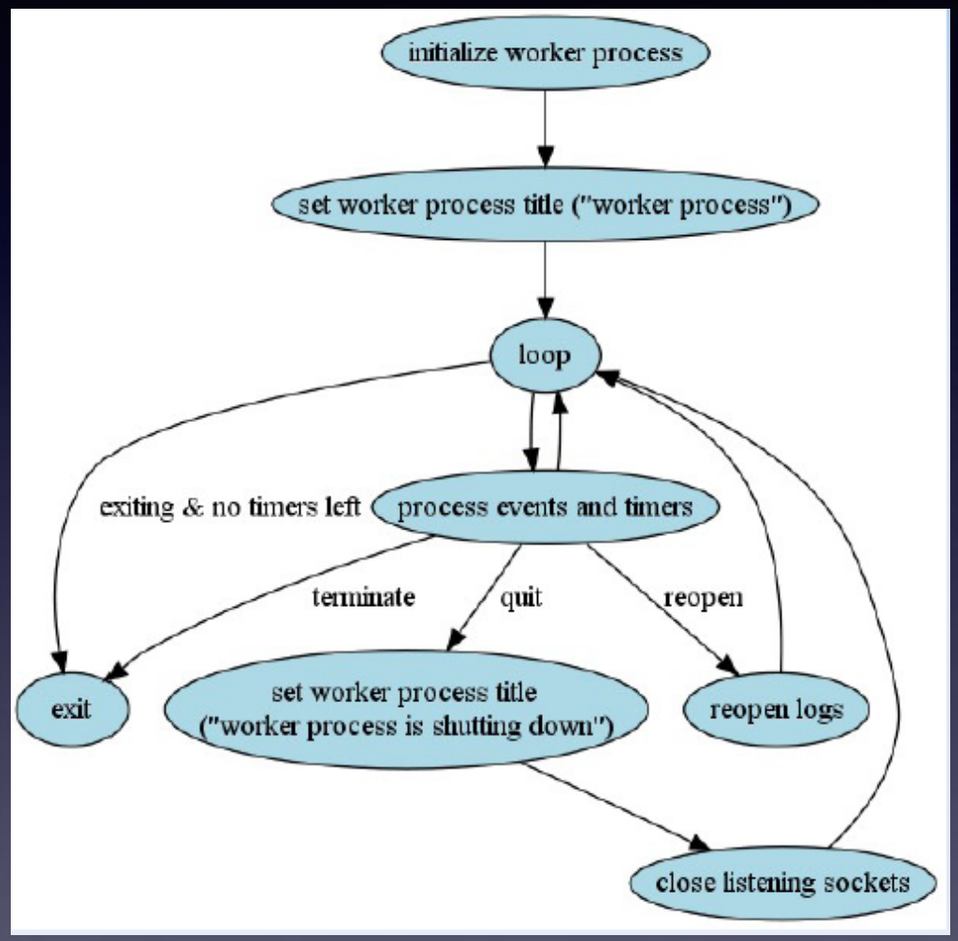
Nginx运行流程


master_process
ngx_master_process_cycle()
ngx_start_worker_processes()
ngx_spawn_process()
fork()
子进程
父进程
for (;;)
其它信号
等待信号
退出信号
信号实际处理
退出
sigsuspend()
设置旗标
ngx_signal_handler()
发生信号
child_process 0
……
child_process n
ngx_worker_process_cycle()
for (;;)
否
进程退出?
是
退出
进程终止?
是
否
ngx_process_events_and_timers()
ngx_process_events()
ngx_event_process_posted()
accept_events
ngx_event_expire_timers()
ngx_event_process_posted()
ngx_posted_events

Nginx信号处理


set the master's title
start workers
&
the cache manager
wait for a signal
reap children
SIGCHLD
terminate workers
terminate
start new workers
&
the cache manager
restart
reconfigure
update time
change binary
exec new binary
(terminate || quit)
&
no workers alive
exit
close listening sockets
quit
no accept
gracefully shutdown workers
reopen
reopen logs

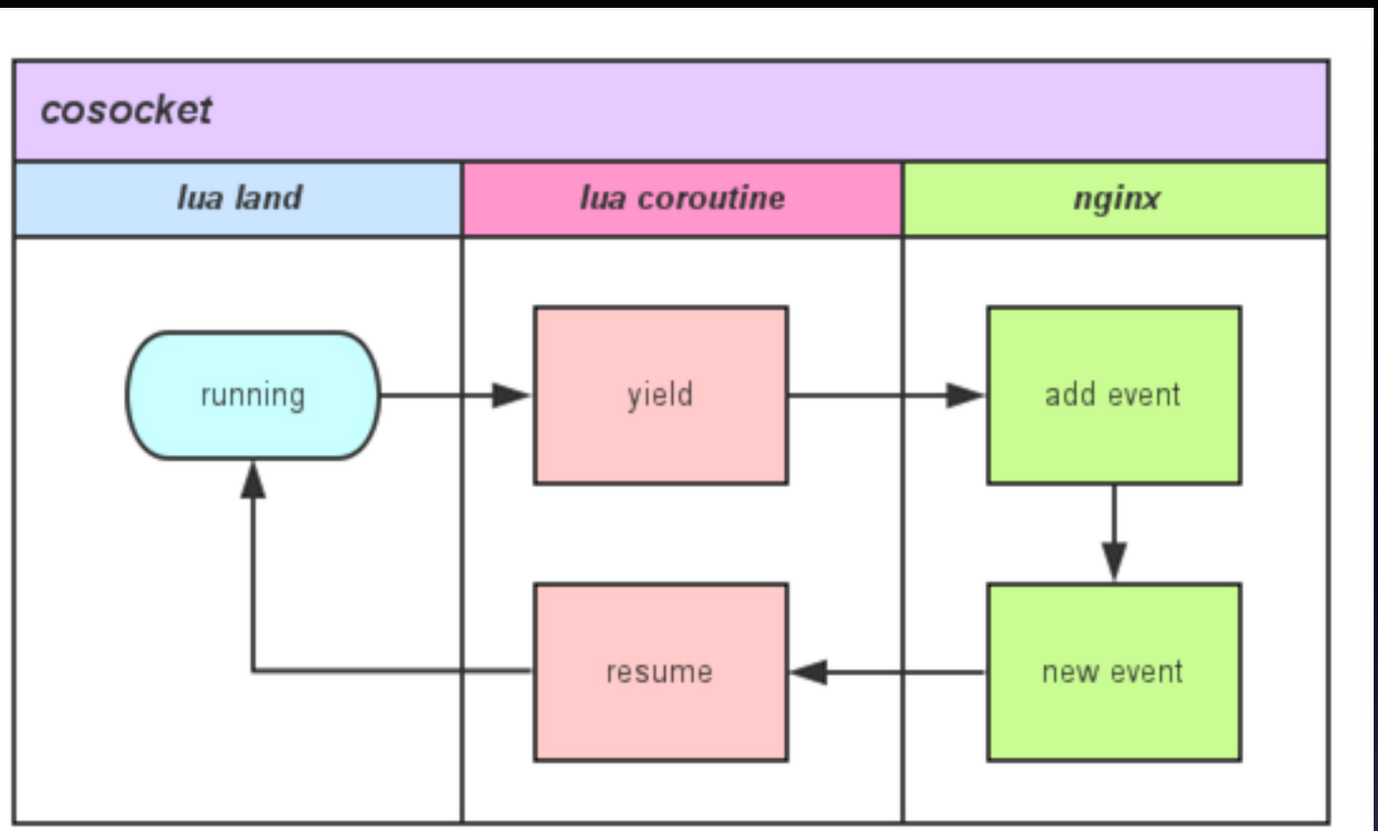
Nginx干活儿进程


initialize worker process
set worker process title ("worker process")
loop
exiting & no timers left
process events and timers
terminate
quit
reopen
exit
set worker process title
("worker process is shutting down")
reopen logs
close listening sockets

Nginx与Lua(OpenResty)的核心原理

- cosocket 是 OpenResty 世界中技术、实用价值最高部分。让我们可以用非常低廉的成本，优雅的姿势，比传统 socket 编程效率高好几倍的方式进行网络编程。无论资源占用、执行效率、并发能力都非常出色。

- cosocket = coroutine + socket
  coroutine：协同程序（后面简称：协程）
  socket：网络套接字



从该图中我们可以看到，用户的 Lua 脚本每触发一个网络操作，都会有协程的 yield 以及 resume，因为请求的 Lua 脚本实际上都运行在独享协程之上，可以在任何需要的时候暂停自己（yield），也可以在任何需要的时候被唤醒（resume）。

OpenResty为Nginx插上翅膀

- 什么是Openresty
  OpenResty® 是一个基于 Nginx 与 Lua 的高性能 Web 平台，其内部集成了大量精良的 Lua 库、第三方模块以及大多数的依赖项。用于方便地搭建能够处理超高并发、扩展性极高的动态 Web 应用、Web 服务和动态网关。

详见官网介绍: https://openresty.org/cn/

典型应用场景

- 其实官网 wiki 已经列了出来：

- 在lua中混合处理不同nginx模块输出（proxy, drizzle, postgres, redis, memcached等）。
- 在请求真正到达上游服务之前，lua中处理复杂的准入控制和安全检查。
- 比较随意的控制应答头（通过Lua）。
- 从外部存储中获取后端信息，并用这些信息来实时选择哪一个后端来完成业务访问。
- 在内容handler中随意编写复杂的web应用，同步编写异步访问后端数据库和其他存储。
- 在rewrite阶段，通过Lua完成非常复杂的处理。
- 在Nginx子查询、location调用中，通过Lua实现高级缓存机制。
- 对外暴露强劲的Lua语言，允许使用各种Nginx模块，自由拼合没有任何限制。该模块的脚本有充分的灵活性，同时提供的性能水平与本地C语言程序无论是在CPU时间方面以及内存占用差距非常小。所有这些都要求LuaJIT 2.x是启用的。其他脚本语言实现通常很难满足这一性能水平。

目前不擅长的应用场景

- 有长时间阻塞调用的过程
- 例如通过 Lua 完成系统命令行调用
- 使用阻塞的Lua API完成相应操作
- 单个会话处理逻辑复杂，尤其是需要和请求方多次交互的长连接场景
- Nginx的内存池 pool 是每次新申请内存存放数据
- 所有的内存释放都是在会话退出的时候统一释放
- 如果单个会话处理过于复杂，将会有过多内存无法及时释放
- 内存占用高的处理
- 受制于Lua VM的最大使用内存 1G 的限制
- 这个限制是单个Lua VM，也就是单个Nginx worker
- 两个会话之间有交流的场景
- 例如你做个在线聊天，要完成两个用户之间信息的传递
- 当前OpenResty还不具备这个通讯能力（后面可能会有所完善）
- 与行业专用的组件对接
- 最好是 TCP 协议对接，不要是 API 方式对接，防止里面有阻塞 TCP 处理
- 由于OpenResty必须要使用非阻塞 API ，所以传统的阻塞 API ，我们是没法直接使用的
- 获取 TCP 协议，使用 cosocket 重写（重写后的效率还是很赞的）
- 每请求开启的 light thread 过多的场景
- 虽然已经是light thread，但它对系统资源的占用相对是比较大的
- 这些适合、不适合信息可能在后面随着 OpenResty 的发展都会有新的变化，大家拭目以待

开发一个原生Nginx模块

- 1. 写了一个Demo　大家一起看看 https://github.com/wujunze/nginx-http-echo-module

畅想Web的未来

- 1. Web1.0 网络—>人 (信息提供者,单向性)
- 2. Web2.0 人->人 (平台)
- 3. Web3.0 人—>网络—>人 (人工智能 关联数据 语义网络)
- 4. future ??

参考文档

- https://github.com/bingbo/blog
- http://blog.csdn.net/heiyeshuwu/
- http://nginx.org/en/docs/contributing_changes.html
- https://www.kancloud.cn/allanyu/openresty-best-practices

QA

Thanks