第六部分

OpenResty及Nginx源码

极客时间

扫码试看/订阅

《Nginx 核心知识100讲》

第三方模块源码的快速阅读方法

1. **分析模块提供的config文件**
   - 理解configure中添加第三方模块的顺序
   - 确认第三方模块的编译方式
2. **分析ngx_module_t模块**
   - 实现了哪些在进程启动、退出时的回调方法？
3. **分析ngx_command_t数组看看支持哪些配置指令**
   - 了解通用的指令值解析方法
4. **对于http模块，分析ngx_http_module_t中在http{}解析前后实现了哪些回调方法**
5. **根据第3、4步，找出该模块生效方式**
   - 在11个阶段中的哪一个阶段处理请求
   - 在过滤响应中生效吗？
   - 在负载均衡中生效吗？
   - 是否提供了新的变量？

config结构

极客时间

- **定义模块名称，用于显示执行结果**
  - 设定ngx_addon_name变量，例如：
    - adding module in /home/web/ngx_cache_purge/

      + ngx_http_cache_purge_module was configured
- **将HTTP模块添加至数组中**
  - 扩展HTTP_MODULES变量，例如
    - HTTP_MODULES="$HTTP_MODULES ngx_http_cache_purge_module"
- **将模块源代码文件添加至Nginx**
  - 扩展NGX_ADDON_SRCS变量，例如：
    - NGX_ADDON_SRCS="$NGX_ADDON_SRCS

      $ngx_addon_dir/ngx_cache_purge_module.c"

configure脚本分析

极客时间

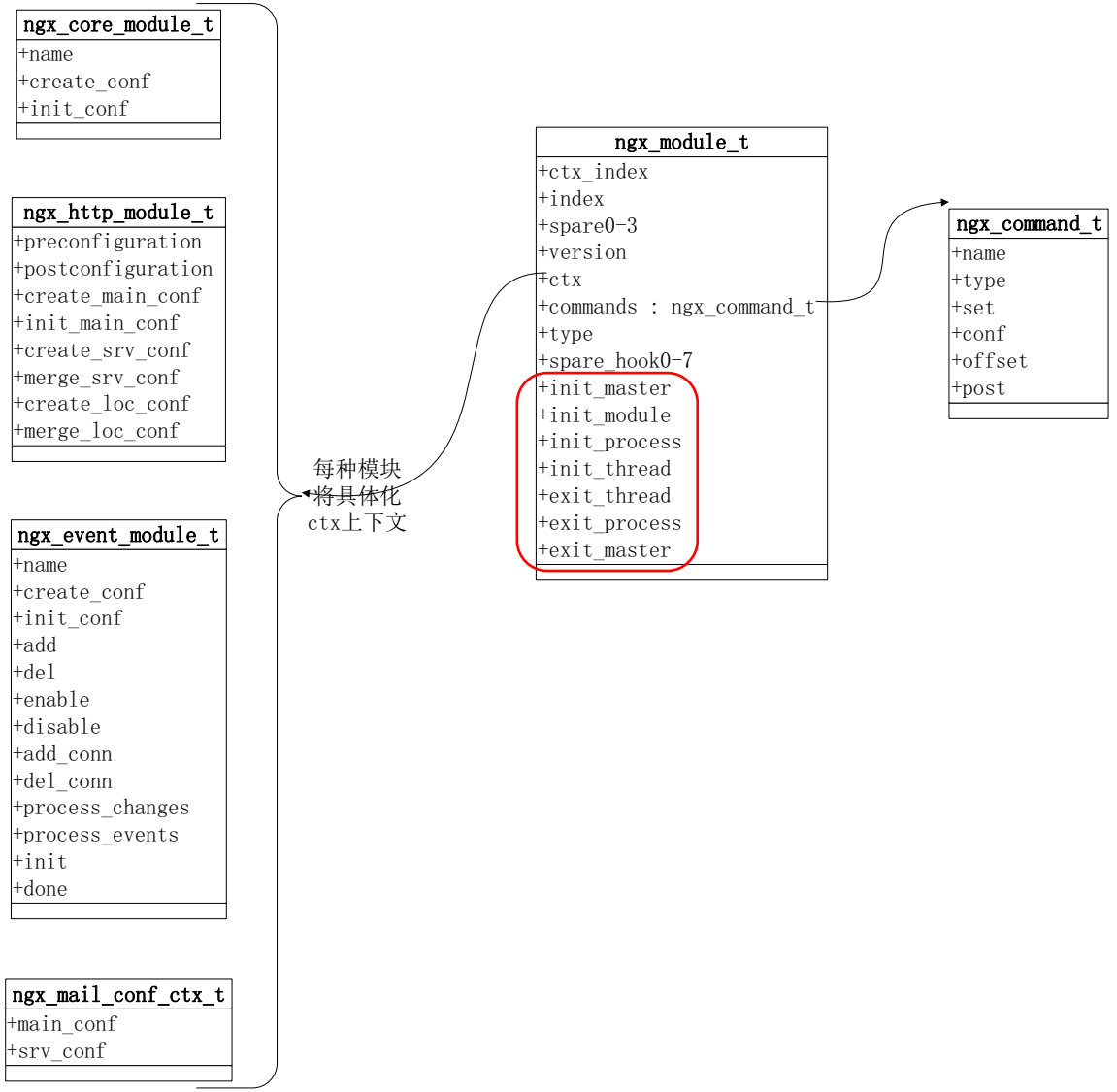
- 解析configure参数，生成编译参数：auto/options
  - 第三方模块通过--add-module参数会添加模块至NGX_ADDONS变量
- 针对不同操作系统、体系架构、编译器，选择特性（例如linux中的epoll或者windows中的iocp）及生成相应编译参数
- 根据所有模块生成ngx_modules.c及makefile
- 在屏幕上显示configure执行结果：auto/summary

configure执行结果

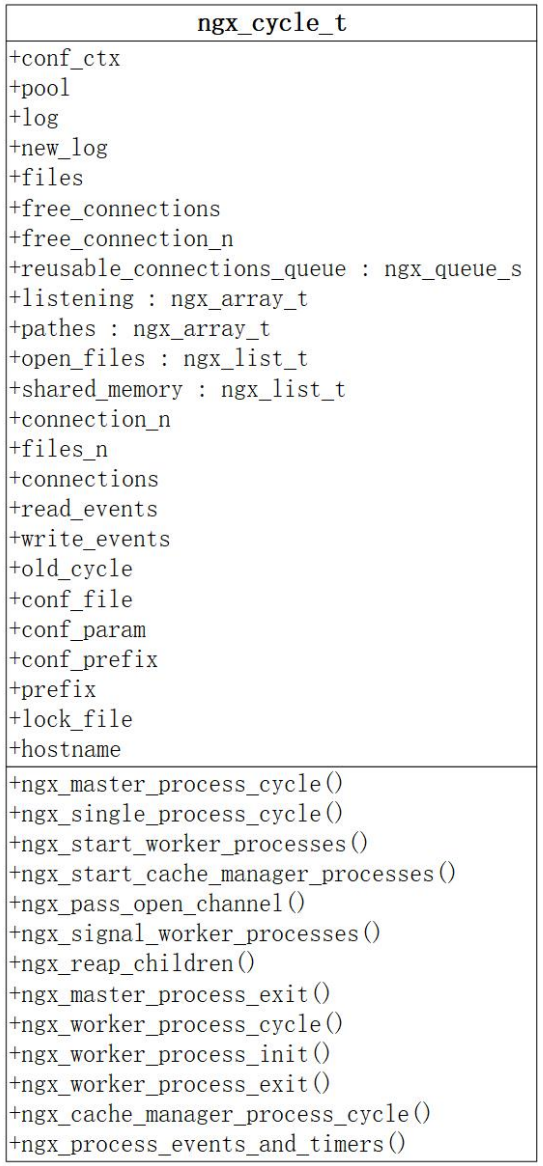

```
Configuration summary
  + using threads
  + using system PCRE library
  + using system OpenSSL library
  + using system zlib library

  nginx path prefix: "/usr/local/nginx"
  nginx binary file: "/usr/local/nginx/sbin/nginx"
  nginx modules path: "/usr/local/nginx/modules"
  nginx configuration prefix: "/usr/local/nginx/conf"
  nginx configuration file: "/usr/local/nginx/conf/nginx.conf"
  nginx pid file: "/usr/local/nginx/logs/nginx.pid"
  nginx error log file: "/usr/local/nginx/logs/error.log"
  nginx http access log file: "/usr/local/nginx/logs/access.log"
  nginx http client request body temporary files: "client_body_temp"
  nginx http proxy temporary files: "proxy_temp"
  nginx http fastcgi temporary files: "fastcgi_temp"
  nginx http uwsgi temporary files: "uwsgi_temp"
  nginx http scgi temporary files: "scgi_temp"
```

Nginx启动、退出时回调方法

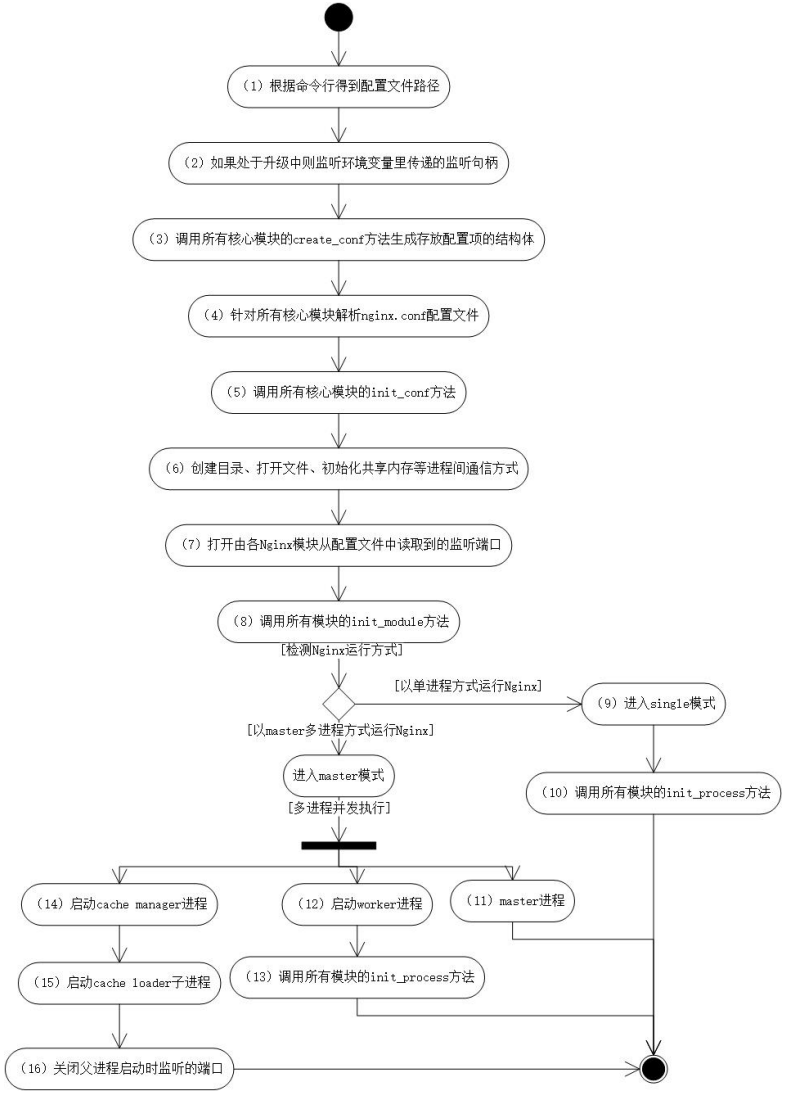
- ~~init_master~~
  - ~~目前版本取消该方法调用~~
- init_module
  - 在master进程中调用
- init_process
  - 在worker进程中调用
- ~~init_thread~~
  - ~~目前版本取消该方法调用~~
- ~~exit_thread~~
  - ~~目前版本取消该方法调用~~
- exit_process
  - 在worker进程退出时调用
- exit_master
  - master进程退出时调用



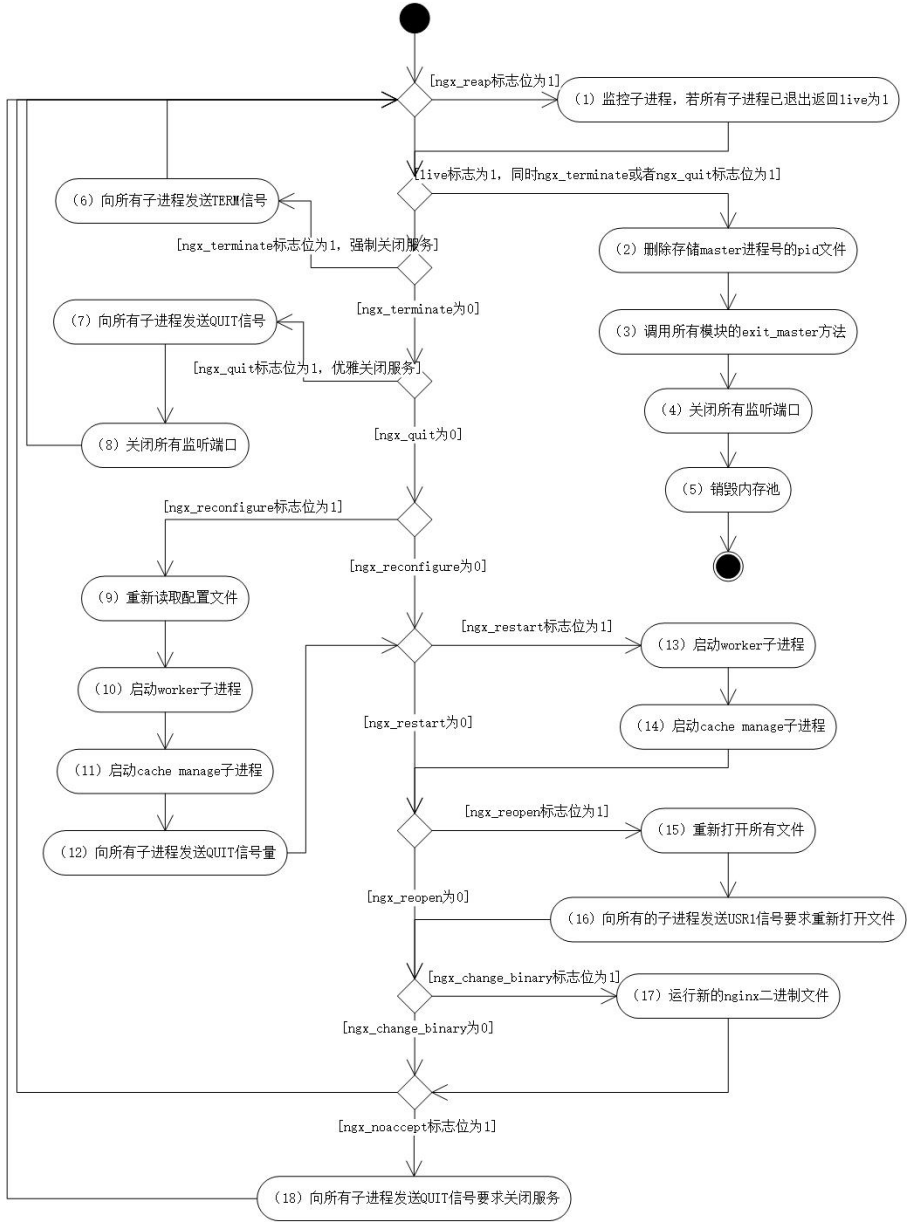
进程核心描述：ngx_cycle_t



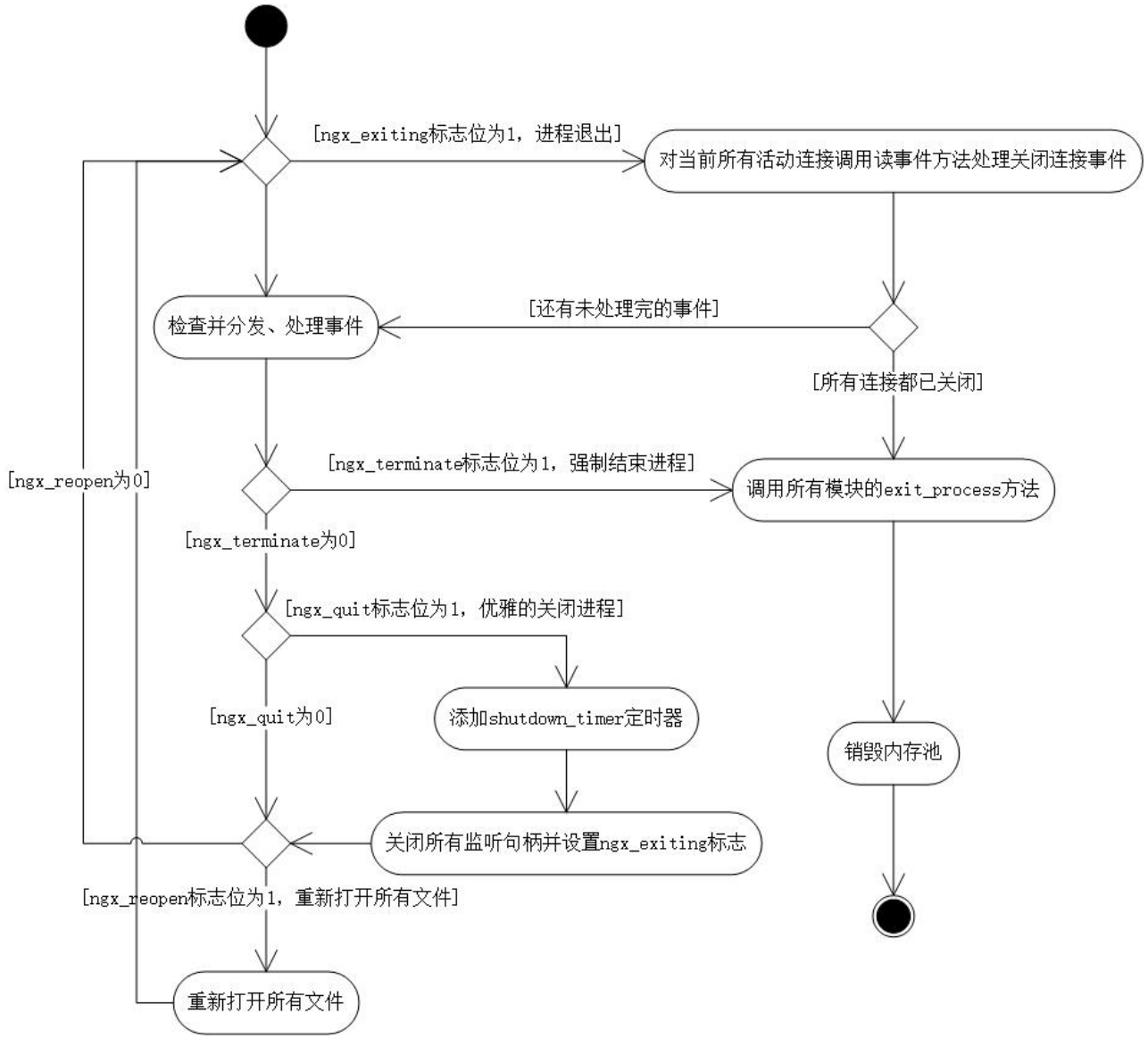
Nginx 启动流程


(1) 根据命令行得到配置文件路径
(2) 如果处于升级中则监听环境变量里传递的监听句柄
(3) 调用所有核心模块的create_conf方法生成存放配置项的结构体
(4) 针对所有核心模块解析nginx.conf配置文件
(5) 调用所有核心模块的init_conf方法
(6) 创建目录、打开文件、初始化共享内存等进程间通信方式
(7) 打开由各Nginx模块从配置文件中读取到的监听端口
(8) 调用所有模块的init_module方法
[检测Nginx运行方式]
[以单进程方式运行Nginx]
(9) 进入single模式
(10) 调用所有模块的init_process方法
[以master多进程方式运行Nginx]
进入master模式
[多进程并发执行]
(14) 启动cache manager进程
(12) 启动worker进程
(11) master进程
(15) 启动cache loader子进程
(13) 调用所有模块的init_process方法
(16) 关闭父进程启动时监听的端口

Master进程循环

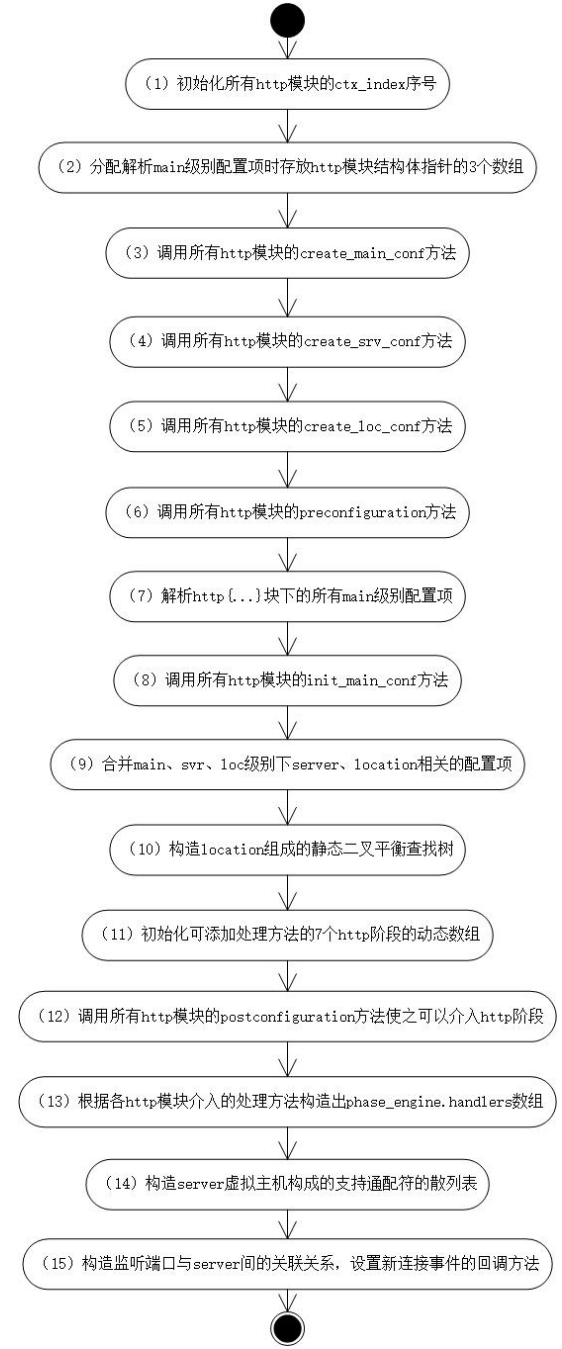

[ngx_reap标志位为1]
（1）监控子进程，若所有子进程已退出返回live为1
live标志为1，同时ngx_terminate或者ngx_quit标志位为1]
（6）向所有子进程发送TERM信号
[ngx_terminate标志位为1，强制关闭服务]
（2）删除存储master进程号的pid文件
[ngx_terminate为0]
（7）向所有子进程发送QUIT信号
（3）调用所有模块的exit_master方法
[ngx_quit标志位为1，优雅关闭服务]
（4）关闭所有监听端口
[ngx_quit为0]
（8）关闭所有监听端口
（5）销毁内存池
[ngx_reconfigure标志位为1]
[ngx_reconfigure为0]
（9）重新读取配置文件
[ngx_restart标志位为1]
（13）启动worker子进程
（10）启动worker子进程
[ngx_restart为0]
（14）启动cache manage子进程
（11）启动cache manage子进程
[ngx_reopen标志位为1]
（15）重新打开所有文件
（12）向所有子进程发送QUIT信号量
[ngx_reopen为0]
（16）向所有的子进程发送USR1信号要求重新打开文件
[ngx_change_binary标志位为1]
（17）运行新的nginx二进制文件
[ngx_change_binary为0]
[ngx_noaccept标志位为1]
（18）向所有子进程发送QUIT信号要求关闭服务

Worker进程循环


[ngx_exiting标志位为1，进程退出]
对当前所有活动连接调用读事件方法处理关闭连接事件
检查并分发、处理事件
[还有未处理完的事件]
[所有连接都已关闭]
[ngx_reopen为0]
[ngx_terminate标志位为1，强制结束进程]
调用所有模块的exit_process方法
[ngx_terminate为0]
[ngx_quit标志位为1，优雅的关闭进程]
[ngx_quit为0]
添加shutdown_timer定时器
销毁内存池
关闭所有监听句柄并设置ngx_exiting标志
[ngx_reopen标志位为1，重新打开所有文件]
重新打开所有文件

http模块初始化

```
typedef struct {
    ngx_int_t   (*preconfiguration)(ngx_conf_t *cf);
    ngx_int_t   (*postconfiguration)(ngx_conf_t *cf);

    void        *(*create_main_conf)(ngx_conf_t *cf);
    char        *(*init_main_conf)(ngx_conf_t *cf, void *conf);

    void        *(*create_srv_conf)(ngx_conf_t *cf);
    char        *(*merge_srv_conf)(ngx_conf_t *cf, void *prev, void *conf);

    void        *(*create_loc_conf)(ngx_conf_t *cf);
    char        *(*merge_loc_conf)(ngx_conf_t *cf, void *prev, void *conf);
} ngx_http_module_t;
```



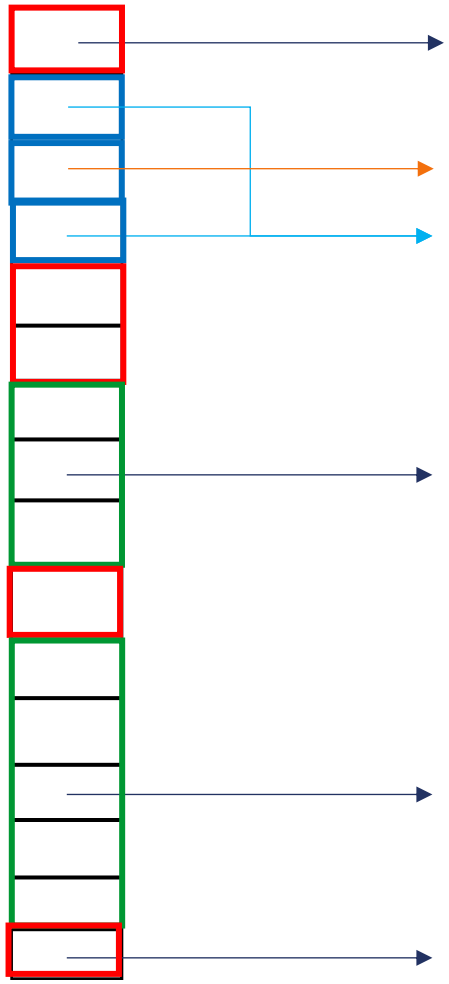
HTTP模块的11个阶段：初始化

极客时间

```
typedef enum {
    NGX_HTTP_POST_READ_PHASE = 0,

    NGX_HTTP_SERVER_REWRITE_PHASE,

    NGX_HTTP_FIND_CONFIG_PHASE,
    NGX_HTTP_REWRITE_PHASE,
    NGX_HTTP_POST_REWRITE_PHASE,

    NGX_HTTP_PREACCESS_PHASE,

    NGX_HTTP_ACCESS_PHASE,
    NGX_HTTP_POST_ACCESS_PHASE,

    NGX_HTTP_PRECONTENT_PHASE,

    NGX_HTTP_CONTENT_PHASE,

    NGX_HTTP_LOG_PHASE
} ngx_http_phases;
```

```
typedef struct {
    ngx_array_t             handlers;
} ngx_http_phase_t;

typedef struct {
    …… …
ngx_http_phase_t        phases[NGX_HTTP_LOG_PHASE
+ 1];
} ngx_http_core_main_conf_t;
```

```
typedef ngx_int_t (*ngx_http_handler_pt)(ngx_http_request_t *r);
```



static ngx_int_t ngx_http_realip_handler(ngx_http_request_t *r)

static ngx_int_t ngx_http_core_find_location(ngx_http_request_t *r)

static ngx_int_t ngx_http_rewrite_handler(ngx_http_request_t *r)

static ngx_int_t ngx_http_access_handler(ngx_http_request_t *r)

void ngx_http_cache_purge_handler(ngx_http_request_t *r)

static ngx_int_t ngx_http_log_handler(ngx_http_request_t *r)

添加模块至11个阶段的两种常用方式

极客时间

- **在HTTP模块解析完配置文件后**

```
cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);

h = ngx_array_push(&cmcf-
>phases[NGX_HTTP_ACCESS_PHASE].handlers);
if (h == NULL) {
    return NGX_ERROR;
}
*h = ngx_http_auth_basic_handler;
```

```
typedef struct {
    ngx_int_t   (*preconfiguration)(ngx_conf_t *cf);
    ngx_int_t   (*postconfiguration)(ngx_conf_t *cf);

    void        *(*create_main_conf)(ngx_conf_t *cf);
    char        *(*init_main_conf)(ngx_conf_t *cf, void *conf);

    void        *(*create_srv_conf)(ngx_conf_t *cf);
    char        *(*merge_srv_conf)(ngx_conf_t *cf, void *prev, void *conf);

    void        *(*create_loc_conf)(ngx_conf_t *cf);
    char        *(*merge_loc_conf)(ngx_conf_t *cf, void *prev, void *conf);
} ngx_http_module_t;
```

- **在HTTP模块解析配置文件，检测到某个开关被打开后**
  - 进入content阶段处理请求
  - 所有反向代理模块使用该方式，因其对其他content阶段模块具有排他性

```
clcf = ngx_http_conf_get_module_loc_conf(cf, ngx_http_core_module);
clcf->handler = ngx_http_proxy_handler;
```

过滤模块的单链表

极客时间

- **处理响应的方法：**

typedef ngx_int_t (*ngx_http_output_header_filter_pt)(ngx_http_request_t *r);
typedef ngx_int_t (*ngx_http_output_body_filter_pt) (ngx_http_request_t *r, ngx_chain_t *chain);

- **链表头：**

extern ngx_http_output_header_filter_pt ngx_http_top_header_filter;
extern ngx_http_output_body_filter_pt   ngx_http_top_body_filter;

- **链表指针：**

static ngx_http_output_header_filter_pt ngx_http_next_header_filter;
static ngx_http_output_body_filter_pt   ngx_http_next_body_filter;

添加过滤模块的方式

极客时间

- **定义本地指针**

```
static ngx_http_output_header_filter_pt  ngx_http_next_header_filter;
static ngx_http_output_body_filter_pt    ngx_http_next_body_filter;
```

- **在哪里插入链表**

```
typedef struct {
    ngx_int_t   (*preconfiguration)(ngx_conf_t *cf);
    ngx_int_t   (*postconfiguration)(ngx_conf_t *cf);

    void        *(*create_main_conf)(ngx_conf_t *cf);
    char        *(*init_main_conf)(ngx_conf_t *cf, void *conf);

    void        *(*create_srv_conf)(ngx_conf_t *cf);
    char        *(*merge_srv_conf)(ngx_conf_t *cf, void *prev, void *conf);

    void        *(*create_loc_conf)(ngx_conf_t *cf);
    char        *(*merge_loc_conf)(ngx_conf_t *cf, void *prev, void *conf);
} ngx_http_module_t;
```

- **如何插入链表**

```
ngx_http_next_header_filter = ngx_http_top_header_filter;
ngx_http_top_header_filter = ngx_http_ssi_header_filter;

ngx_http_next_body_filter = ngx_http_top_body_filter;
ngx_http_top_body_filter = ngx_http_ssi_body_filter;
```

rewrite的脚本指令

- set
- if
- break
- rewrite
- return

协程与rewrite脚本

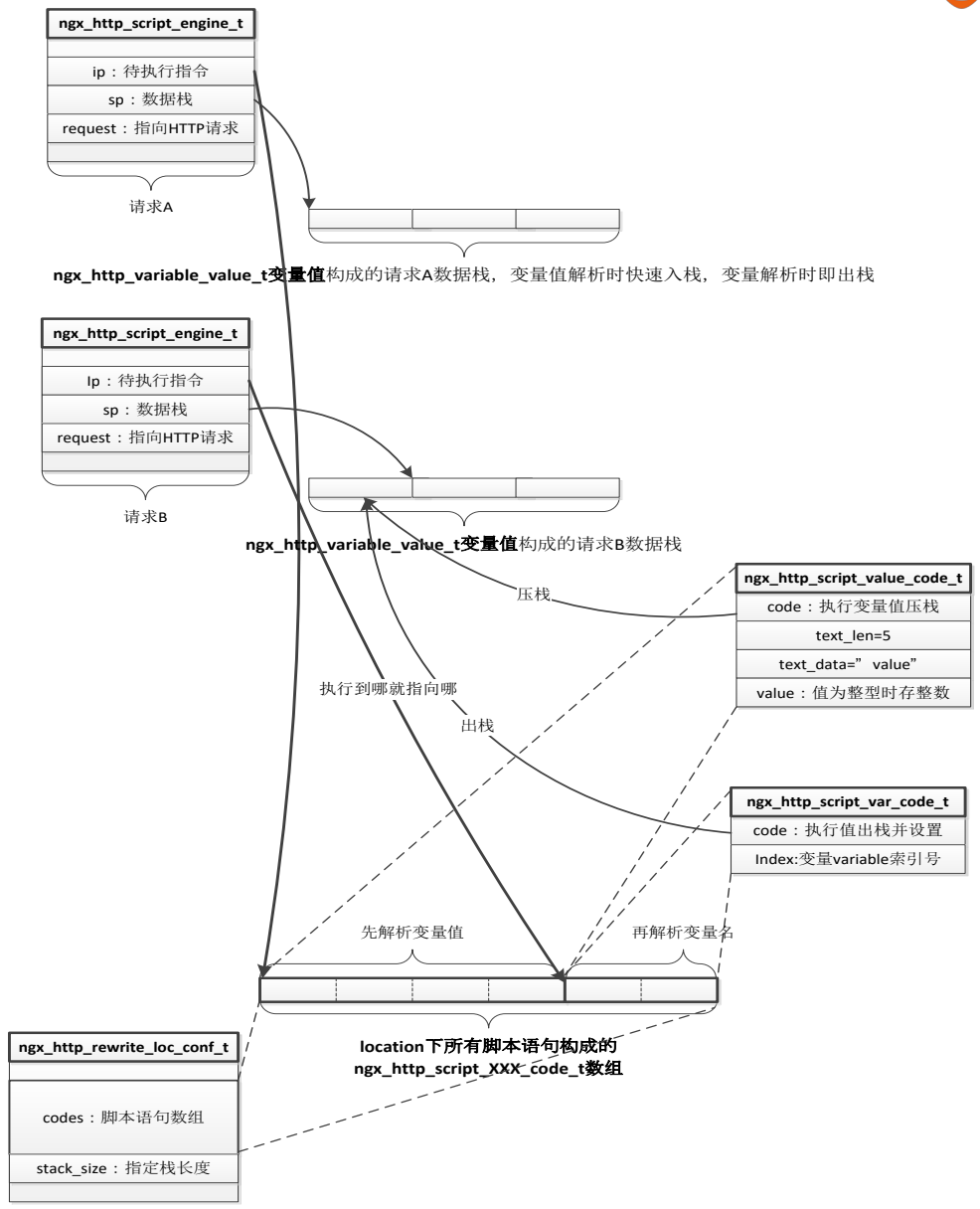
极客时间

- **上下文信息**
  - 指令集
  - 数据栈
  - 下一条指令

- **调度系统：切换任务**
  - 载入新任务
    - 载入数据栈
  - 重置下一条指令

rewrite脚本：设置变量

- **所有脚本指令**
  - ngx_http_rewrite_loc_conf_t->codes
- **请求的执行上下文**
  - ngx_http_script_engine_t
    - 当前指令
      - ip
    - 数据栈
      - sp



图中以set $variable value;作为示例，脚本由右向左解析为ngx_http_script_value_code_t、ngx_http_script_var_code_t

脚本式编程：指令的实现

极客时间

```
typedef void (*ngx_http_script_code_pt) (ngx_http_script_engine_t *e);

typedef struct {
    ngx_http_script_code_pt     code;
    … …
} ngx_http_script_..._code_t;
```

当if指令块连续出现时

极客时间

```
location /only-one-if
{
    set $true 1;
    if ($true) {
        add_header X-First 1;
    }
    if ($true) {
        add_header X-Second 2;
    }
    return 204;
}
```

- if为真时，其{}内的值指令会被赋予请求

```
void
ngx_http_script_if_code(ngx_http_script_engine_t *e)
{
    ngx_http_script_if_code_t  *code;

    code = (ngx_http_script_if_code_t *) e->ip;

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, e->request->connection->log, 0,
                   "http script if");

    e->sp--;

    if (e->sp->len && (e->sp->len != 1 || e->sp->data[0] != '0')) {
        if (code->loc_conf) {
            e->request->loc_conf = code->loc_conf;
            ngx_http_update_location_config(e->request);
        }

        e->ip += sizeof(ngx_http_script_if_code_t);
        return;
    }

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, e->request->connection->log, 0,
                   "http script if: false");

    e->ip += code->next;
} /* end ngx_http_script_if_code */
```

当if块内出现反向代理时

```
location /crash {
        set $true 1;
        if ($true) {
            proxy_pass  http://127.0.0.1:9000;
         }

         if ($true) {
                # no handler here
         }
}
```

```
void
ngx_http_update_location_config(ngx_http_request_t *r)
{
    ngx_http_core_loc_conf_t  *clcf;

    clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

    ... ...

    if (clcf->handler) {
        r->content_handler = clcf->handler;
    }
} ? end ngx_http_update_location_config ?
```

if指令出现问题的原因

极客时间

- if指令在rewrite阶段执行
- if {}块中的配置，会在if条件为真时，替换当前请求的配置
  - if {}同样向上继承父配置
  - 当在rewrite阶段顺序执行时，每次if为真都会替换当前请求的配置
- if {}中的配置，会影响rewrite阶段之后的阶段执行

```
void
ngx_http_script_if_code(ngx_http_script_engine_t *e)
{
    ngx_http_script_if_code_t  *code;

    code = (ngx_http_script_if_code_t *) e->ip;

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, e->request->connection->log, 0,
                   "http script if");

    e->sp--;

    if (e->sp->len && (e->sp->len != 1 || e->sp->data[0] != '0')) {
        if (code->loc_conf) {
            e->request->loc_conf = code->loc_conf;
            ngx_http_update_location_config(e->request);
        }

        e->ip += sizeof(ngx_http_script_if_code_t);
        return;
    }

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, e->request->connection->log, 0,
                   "http script if: false");

    e->ip += code->next;
} /* end ngx_http_script_if_code */
```

使用if指令的正确姿势

极客时间

- **理解if的执行流程**
  - 基本脚本编程的rewrite模块中的5个指令，在rewrite阶段执行，而此阶段后，还有7个阶段中的许多模块可能会执行
  - if指令一旦为真后，会替换当前请求的配置块，请务必确保if {}中的指令可以正确处理请求，不依赖if {}块外的指令
  - break会阻断后续rewrite阶段指令的执行

coredump核心转储文件

极客时间

Syntax:	**worker_rlimit_core** size;
Default:	—
Context:	main

Syntax:	**working_directory directory;**
Default:	—
Context:	main

gdb调试基本命令

极客时间

- **bt/backstrace**
  - 显示函数栈调用情况
- **f/frame**
  - 显示某一栈帧的详细情况
- **p/print**
  - 显示某一变量的值
- **l/list**
  - 显示当前行附近代码
- **x**
  - 显示内存中具体地址指向的值
- **i/info**
  - 显示信息，如i r为显示寄存器信息

启用多进程模式

极客时间

Syntax:	**daemon on** | off;
Default:	daemon on;
Context:	main

debug定位问题：出现错误时的应对方案

极客时间

Syntax:	**debug_points** abort | stop;
Default:	—
Context:	main

- **功能**
  - 某些Nginx模块可能存在代码Bug，在生产环境则忽略问题，打开debug_points后则遇到问题后停止服务，方便定位问题
- **工作方式**
  - abort
    - 生成核心转储coredump文件并结束进程
  - stop
    - 结束进程
- **生效时刻**
  - 任何Nginx模块调用ngx_debug_point()函数的位置，例如：
    - ssi模块中输出“the same buf was used in ssi”日志后
    - 共享内存分配出错，输出“ngx_slab_alloc(): page is busy”日志后

控制debug级别error.log日志的输出

极客时间

Syntax:	**debug_connection** address | CIDR | unix:;
Default:	—
Context:	events

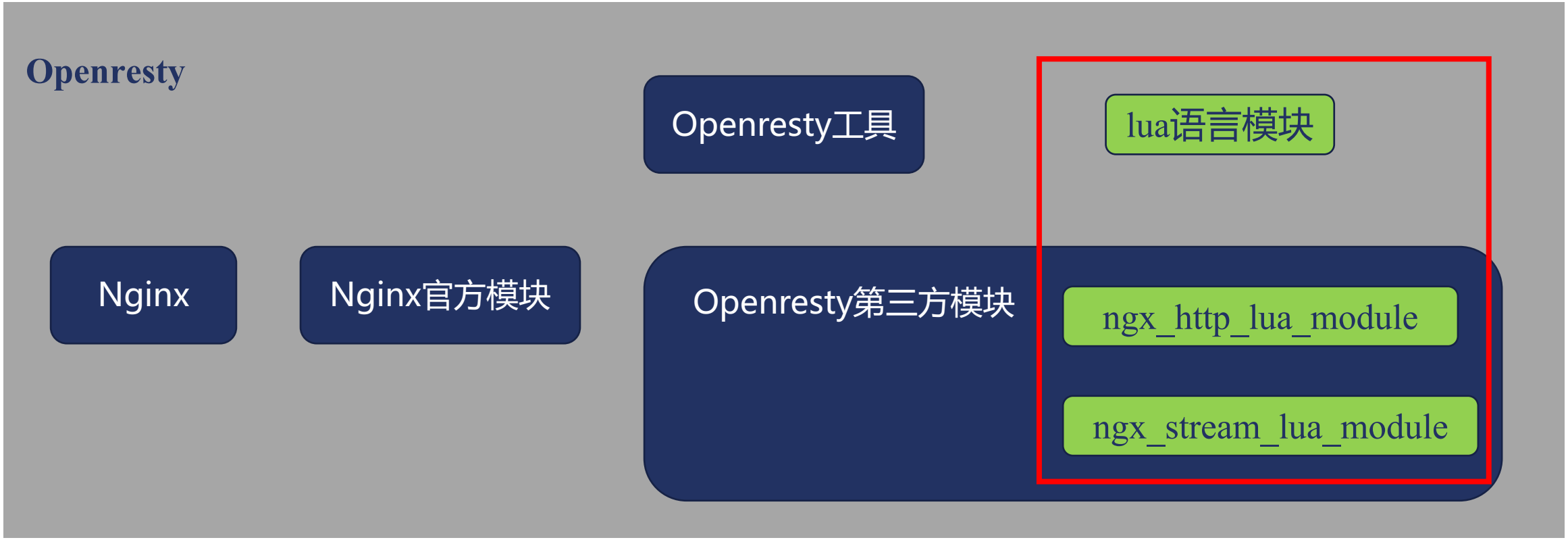
针对特定客户端打印DEBUG级别日志 ，其他日志仍然遵从error_log指令后设置的日志级别

注意：configure中必须加--with-debug编译选项

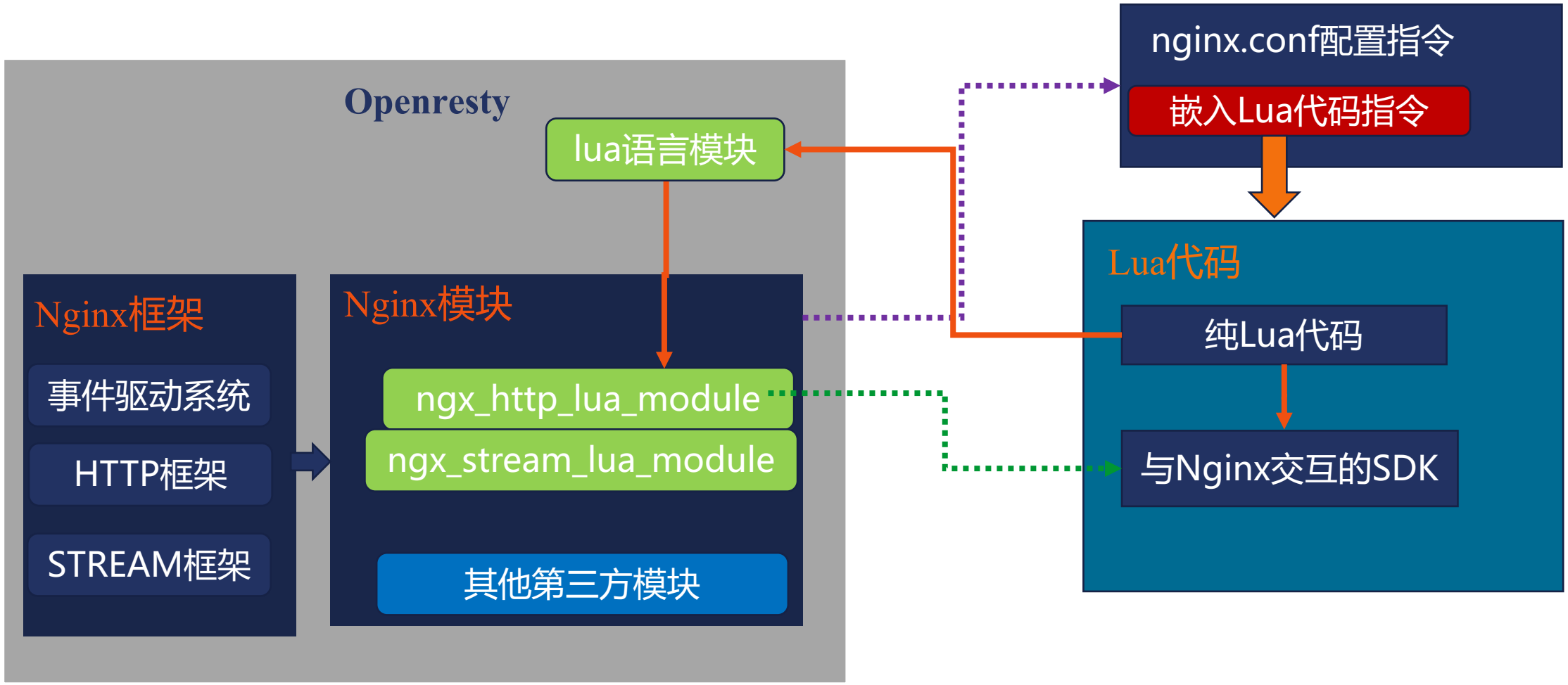
DEBUG日志

- **建立连接**
  - SSL握手
- **接收请求的HTTP头部**
  - 请求行解析
  - 请求头部解析
- **11个阶段的处理**
  - 寻找处理请求的location
  - 反向代理构造上游请求
  - 接收客户端请求包体
- **构造出的响应头部**
- **发送响应**
  - 过滤模块

Openresty的主要组成



Openresty的运行机制



Openresty中的SDK

极客时间

- cosocket通讯
  - udp
  - tcp
- 基于共享内存的字典shared.DICT
- 定时器
- 基于协程的并发编程
- 获取客户端请求与响应的信息
- 修改客户端请求与响应，包括发送响应
- 子请求
- 工具类
  - 正则表达式
  - 日志
  - 系统配置
  - 编解码
  - 时间

Openresty的使用要点

极客时间

- 不破坏Nginx的事件驱动体系，不使用任何会阻塞Nginx进程进行事件调度的方法
  - 不调用会导致Nginx进程主动休眠的方法
    - 谨慎调用第三方库，若不是通过cosocket实现（例如Lua标准网络库或者第三方服务提供的SDK库），就无法融入Nginx的事件驱动体系中，那么对TCP消息的处理通常会导致Nginx进程进入Sleep状态
  - 不调用会长时间占用CPU的方法
    - 避免Lua代码块执行密集计算指令

- 不破坏Nginx的低内存消耗优点，避免对每个请求分配过大内存
  - 对会导致分配内存的Lua SDK，谨慎分配内存。考虑用状态机多次分批处理。

- 理解Lua代码块如何嵌入Nginx中执行
- 理解Lua SDK详细用法，及它们如何集成在Nginx中处理请求
- 保持Lua代码的高效

Openresty的Nginx模块：核心模块

极客时间

- **ndk_http_module**
  - Openresty的基础模块，全名为Nginx Development Kit，顾名思议是一个开发工具包，为如ngx_http_lua_module等模块提供通用功能
  - 通过--without-ngx_devel_kit_module禁用模块
- **ngx_http_lua_module：**
  - Openresty提供HTTP服务lua编程能力的核心模块，在所有阶段、过滤、负载均衡等允许用lua语言处理请求
  - 通过--without-http_lua_module禁用模块
- **ngx_http_lua_upstream_module**
  - 作为ngx_http_lua_module模块的补充，为lua编程提供更多的关于upstream管理的API
  - 通过--without-http_lua_upstream_module禁用模块
- **ngx_stream_lua_module**
  - Openresty提供四层TCP/UDP服务lua编程能力的核心模块，允许用lua语言处理请求
  - 通过--without-stream_lua_module禁用模块

Openresty的Nginx模块：反向代理模块

极客时间

- **ngx_http_memc_module：**
  - content阶段的反向代理模块，上游为Memcached，相比Nginx官方的memcached模块（仅提供GET命令），它提供了全部的命令
  - 通过--without-http_memc_module禁用模块
- **ngx_http_redis2_module ：**
  - content阶段的反向代理模块，上游为Redis服务
  - 通过--without-http_redis2_module禁用模块
- **ngx_http_redis_module：**
  - content阶段的反向代理模块，上游为Redis服务。这两个redis模块皆为Nginx模块，使用nginx.conf语法，皆可被Lua语言版本的lua-resty-redis取代（它使用ngx_http_lua_module的tcp sdk）
  - 通过--without-http_redis_module禁用模块
- **ngx_http_postgres_module**
  - content阶段的反向代理模块，上游为Postgres数据库。
  - 通过--with-http_postgres_module启用该模块
- **ngx_http_drizzle_module**
  - content阶段的反向代理模块，上游为Mysql或者Drizzle数据库
  - 通过--with-http_drizzle_module启用该模块

Openresty的Nginx模块：工具模块（1）

极客时间

- **ngx_http_headers_more_filter_module：**
  - rewrite阶段处理请求、过滤模块，修改请求、响应的HTTP头部
  - 通过--without-http_headers_more_module禁用模块
- **ngx_http_rds_json_filter_module：**
  - 过滤模块，将RDS格式响应转换为JSON格式
  - 通过--without-http_rds_json_module禁用模块
- **ngx_http_xss_filter_module**
  - 过滤模块，使Nginx支持ajax跨站请求
  - 通过--without-http_xss_module禁用模块
- **ngx_http_rds_csv_filter_module：**
  - 过滤模块，将RDS格式响应转换为CSV格式
  - 通过--without-http_rds_csv_module禁用模块
- **ngx_http_srcache_filter_module**
  - 在access阶段、响应过滤时生效，针对location在内存中缓存响应
  - 通过--without-http_srcache_module禁用模块
- **ngx_coolkit_module**
  - 提供一些小功能集合，例如从Basic HTTP Authentication协议中取出密码作为变量值，等等
  - 通过--without-http_coolkit_module禁用模块
- **ngx_http_iconv_module**
  - 过滤模块，可以将变量或者HTTP响应从一种编码（例如GBK）转换为另一种编码（例如UTF8）
  - 通过--with-http_iconv_module启用模块

Openresty的Nginx模块：工具模块（2）

极客时间

- **ngx_http_echo_module：**
  - 在content阶段、响应过滤时皆会生效，生成或者修改返回客户端的响应。
  - 通过--without-http_echo_module禁用模块
- **ngx_http_set_misc_module：**
  - 扩展了ngx_http_rewrite_module模块的set指令
  - 通过--without-http_set_misc_module禁用模块
- **ngx_http_encrypted_session_module**
  - 使用AES256算法对变量的值进行加密和解密
  - 通过--without-http_encrypted_session_module禁用模块
- **ngx_http_form_input_module：**
  - rewrite阶段读取请求包体，将POST和PUT请求中的FORM表单内容，解析成nginx变量供其他模块使用
  - 通过--without-http_form_input_module禁用模块
- **ngx_http_array_var_module**
  - 将变量中数组类型的值，以固定分隔符的形式取出生成数组变量，并可将数组变量再通过固定分隔符组合生成新的字符串变量
  - 通过--without-http_array_var_module禁用模块

Openresty的官方Lua模块（1）

极客时间

- **lua-redis-parser**
  - 将原始的redis响应解析为Lua语言的数据结构
  - 通过--without-lua_redis_parser禁用模块
- **lua-rds-parser**
  - 将原始的Drizzle、Mysql、PostGres数据库响应解析为Lua语言的数据结构
  - 通过--without-lua_rds_parser禁用模块
- **lua-resty-dns**
  - 基于cosocket实现DNS协议的通讯
  - 通过--without-lua_resty_dns禁用模块
- **lua-resty-memcached**
  - 基于ngx.socket.tcp实现的memcached客户端
  - 通过--without-lua_resty_memcached禁用模块
- **lua-resty-redis**
  - 基于ngx.socket.tcp实现的redis客户端
  - 通过--without-lua_resty_redis禁用模块

Openresty的官方Lua模块（2）

极客时间

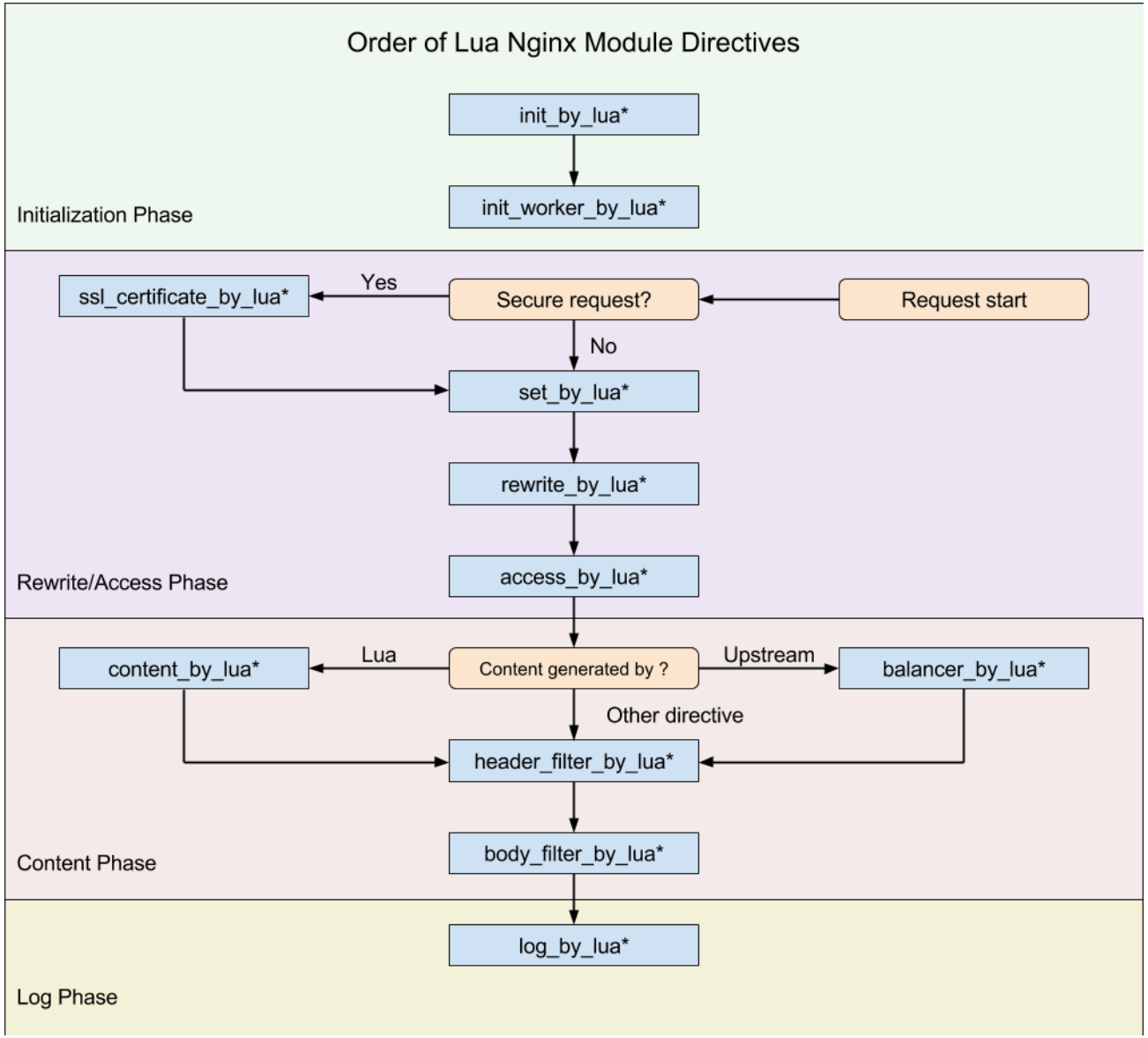
- **lua-resty-mysql**
  - 基于ngx.socket.tcp实现的mysql客户端
  - 通过--without-lua_resty_mysql禁用模块
- **lua-resty-upload**
  - 实现了读取请求中上传文件内容的功能
  - 通过--without-lua_resty_upload禁用模块
- **lua-resty-upstream-healthcheck**
  - 通过向上游服务发送心跳检查上游服务的健康状态，以踢除有问题的节点
  - 通过--without-lua_resty_upstream_healthcheck禁用模块
- **lua-resty-string**
  - 提供了hash函数（如SHA1、MD5等）、字符串转换函数等工具
  - 通过--without-lua_resty_string禁用模块
- **lua-cjson**
  - 提供Lua语言与Json数据结构之间的编码、解码工具库
  - 通过--without-lua_cjson禁用模块

Openresty的官方Lua模块（3）

极客时间

- **lua-resty-websocket**
  - 基于ngx_http_lua_module实现了websocket协议的客户端及服务器端
  - 通过--without-lua_resty_websocket禁用Lua库
- **lua-resty-limit-traffic**
  - 基于Lua语言实现了多种限速方案
  - 通过--without-lua_resty_limit_traffic禁用Lua库
- **lua-resty-lock**
  - 提供基于共享内存的非阻塞互斥锁，基于ngx_http_lua_module中的ngx.sleep方法实现
  - 通过--without-lua_resty_lock禁用Lua库
- **lua-resty-lrucache**
  - 基于内存实现的LRU（Least Recently Used）缓存，故不能跨Nginx worker进程使用
  - 通过--without-lua_resty_lrucache禁用Lua库
- **lua-resty-core**
  - ngx_http_lua_module模块提供的SDK皆基于老的C API实现，而本模块使用ffi方式重新实现了ngx_http_lua_module模块中的SDK
  - 通过--without-lua_resty_core禁用Lua库

Lua代码嵌入指令


Order of Lua Nginx Module Directives
init_by_lua*
init_worker_by_lua*
Initialization Phase
ssl_certificate_by_lua*
Yes
Secure request?
Request start
No
set_by_lua*
rewrite_by_lua*
access_by_lua*
Rewrite/Access Phase
content_by_lua*
Lua
Content generated by ?
Upstream
balancer_by_lua*
Other directive
header_filter_by_lua*
body_filter_by_lua*
Content Phase
log_by_lua*
Log Phase

在Nginx启动过程中嵌入Lua代码

极客时间

- **init_by_lua/init_by_lua_block/init_by_lua_file**
  - context：http
  - 在Nginx解析配置文件（Master进程）时在Lua VM层面立即调用的Lua代码

- **init_worker_by_lua/init_worker_by_lua_block/init_worker_by_lua_file**
  - context: http
  - 在Nginx worker进程启动时调用，实际实现了ngx_module_t中的init_process方法

在11个HTTP阶段中嵌入Lua代码

- set_by_lua/set_by_lua_block/set_by_lua_file
  - **context**: server, server if, location, location if
  - 将Lua代码添加到Nginx官方ngx_http_rewrite_module 模块中的脚本指令中，依赖ngx_devel_kit模块
- rewrite_by_lua/rewrite_by_lua_block/rewrite_by_lua_file
  - **context**: http, server, location, location if
  - 将Lua代码添加到11个阶段中的rewrite阶段中，作为独立模块执行
- access_by_lua/access_by_lua_block/access_by_lua_file
  - **context**: http, server, location, location if
  - 将Lua代码添加到11个阶段中的access阶段中执行
- content_by_lua/content_by_lua_block/content_by_lua_file
  - **context**: location, location if
  - 在11个阶段的content阶段以独占方式（参见各反向代理模块，以r->content_handler方式拒绝其他content模块执行）执行Lua代码
- log_by_lua/log_by_lua_block/log_by_lua_file
  - **context**: http, server, location, location if
  - 将Lua代码添加到11个阶段中的log阶段中执行

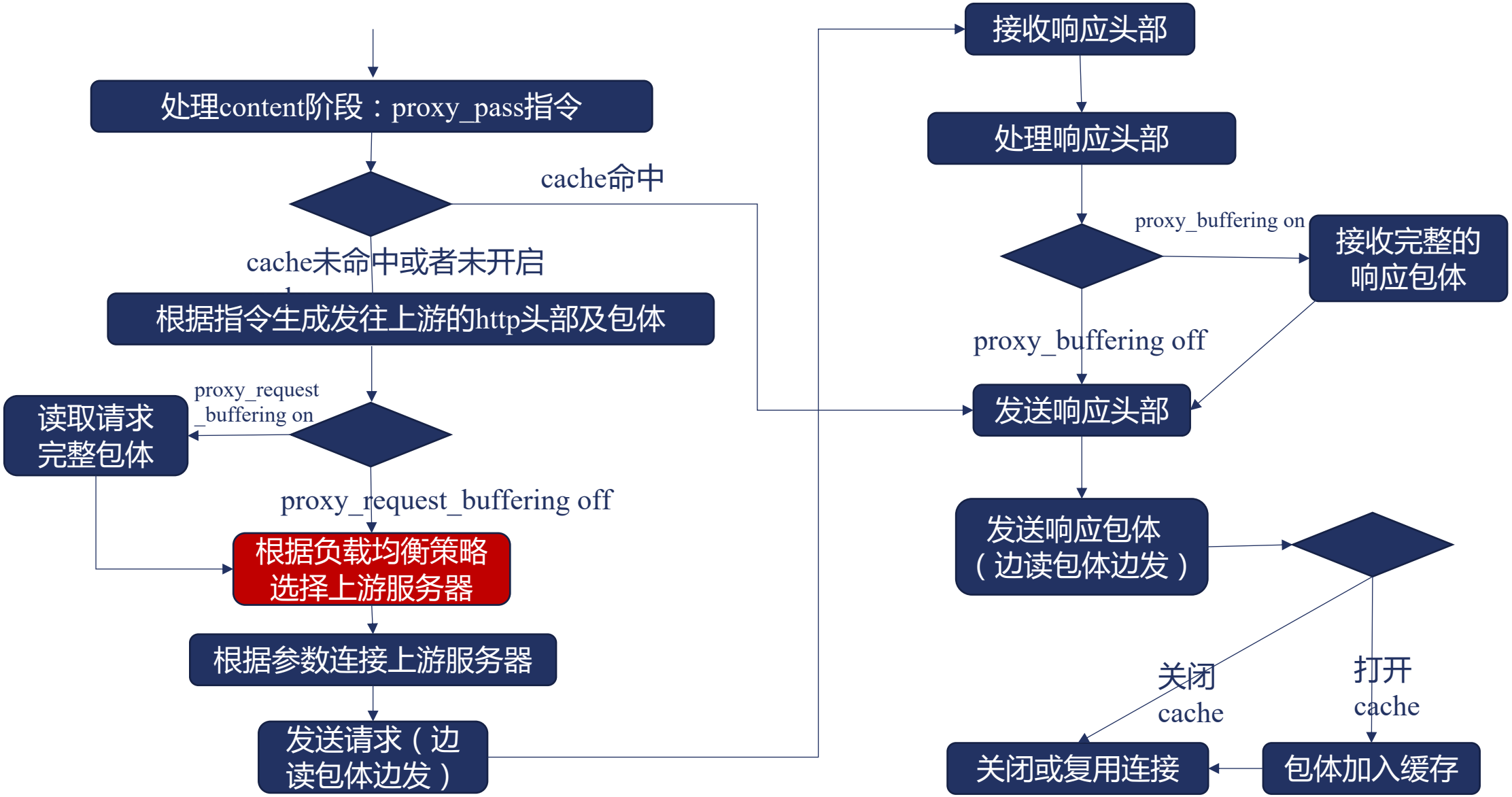
POST_READ	realip
SERVER_REWRITE	rewrite
FIND_CONFIG	
REWRITE	rewrite
POST_REWRITE	
PREACCESS	limt_conn, limit_req
ACCESS	auth_basic, access, auth_request
POST_ACCESS	
PRECONTENT	try_files
CONTENT	index, autoindex, concat
LOG	access_log

控制rewrite/access是否延迟执行Lua代码

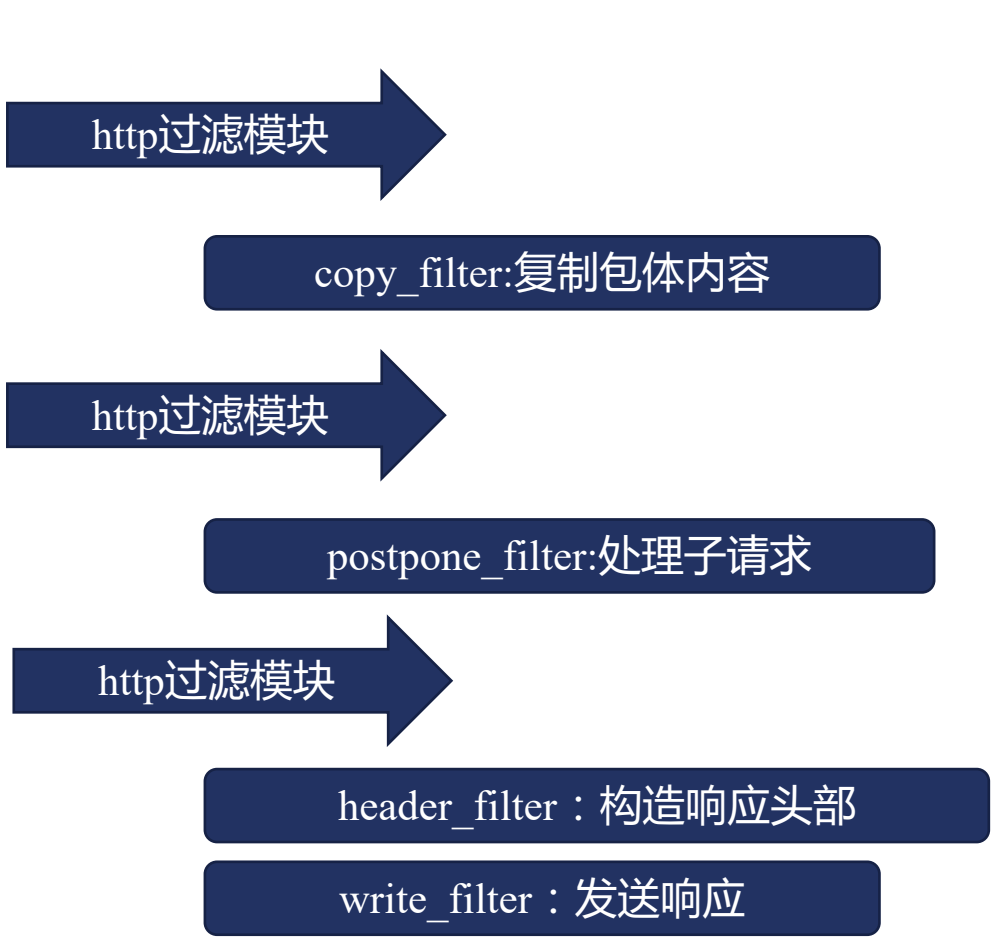
极客时间

- **rewrite_by_lua_no_postpone**
  - **context**: http
  - 控制是否在rewrite阶段延迟至最后再执行Lua代码，默认关闭
- **access_by_lua_no_postpone**
  - **context**: http
  - 控制是否在access阶段延迟至最后再执行Lua代码，默认关闭

HTTP 反向代理流程


处理content阶段：proxy_pass指令
cache命中
cache未命中或者未开启
根据指令生成发往上游的http头部及包体
proxy_request
_buffering on
读取请求
完整包体
proxy_request_buffering off
根据负载均衡策略
选择上游服务器
根据参数连接上游服务器
发送请求（边
读包体边发）
接收响应头部
处理响应头部
proxy_buffering on
接收完整的
响应包体
proxy_buffering off
发送响应头部
发送响应包体
（边读包体边发）
关闭
cache
打开
cache
关闭或复用连接
包体加入缓存

返回响应-加工响应内容




```
&ngx_http_write_filter_module,
&ngx_http_header_filter_module,
&ngx_http_chunked_filter_module,
&ngx_http_v2_filter_module,
&ngx_http_range_header_filter_module,
&ngx_http_gzip_filter_module,
&ngx_http_postpone_filter_module,
&ngx_http_ssi_filter_module,
&ngx_http_charset_filter_module,
&ngx_http_sub_filter_module,
&ngx_http_addition_filter_module,
&ngx_http_userid_filter_module,
&ngx_http_headers_filter_module,
&ngx_http_echo_module,
&ngx_http_xss_filter_module,
&ngx_http_srcache_filter_module,
&ngx_http_lua_module,
&ngx_http_headers_more_filter_module,
&ngx_http_rds_json_filter_module,
&ngx_http_rds_csv_filter_module,
&ngx_http_copy_filter_module,
&ngx_http_range_body_filter_module,
&ngx_http_not_modified_filter_module,
```

在过滤响应、负载均衡时嵌入Lua代码

极客时间

- **header_filter_by_lua/header_filter_by_lua_block/header_filter_by_lua_file**
  - context: http, server, location, location if
  - 将Lua代码嵌入到响应头部过滤阶段中，注意所属模块ngx_http_lua_module的次序
- **body_filter_by_lua/body_filter_by_lua_block/body_filter_by_lua_file**
  - context: http, server, location, location if
  - 将Lua代码嵌入到响应包体过滤阶段中，注意所属模块ngx_http_lua_module的次序
- **balancer_by_lua_block/balancer_by_lua_file**
  - context: upstream
  - 将Lua代码添加到反向代理模块、生成上游服务地址的init_upstream回调方法中

在Openssl处理SSL协议时嵌入Lua代码

极客时间

- ssl_certificate_by_lua_block/ssl_certificate_by_lua_file
  - **context**: server
  - 利用Openssl库（要求1.0.2e版本以上）的SSL_CTX_set_cert_cb特性，将Lua代码添加到验证下游客户端SSL证书的代码前
- ssl_session_fetch_by_lua_block/ssl_session_fetch_by_lua_file
  - **context**: http
  - 利用Openssl库（要求1.0.2h版本以上）的SSL_CTX_sess_set_get_cb特性，将Lua代码添加到获取客户端Session时
- ssl_session_store_by_lua_block/ssl_session_store_by_lua_file
  - **context**: http
  - 利用Openssl库（要求1.0.2h版本以上）的SSL_CTX_sess_set_new_cb特性，将Lua代码添加到保存客户端Session时

在Lua代码中获取当前阶段

极客时间

- ngx.get_phase
  - **context**: init_by_lua*, init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, balancer_by_lua*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*, ssl_session_store_by_lua*
  - **init** for the context of init_by_lua*.
  - **init_worker** for the context of init_worker_by_lua*.
  - **ssl**_cert for the context of ssl_certificate_by_lua*.
  - **ssl_session_fetch** for the context of ssl_session_fetch_by_lua*.
  - **ssl_session_store** for the context of ssl_session_store_by_lua*.
  - **set for** the context of set_by_lua*.
  - **rewrite** for the context of rewrite_by_lua*.
  - **balancer** for the context of balancer_by_lua*.
  - **access** for the context of access_by_lua*.
  - **content** for the context of content_by_lua*.
  - **header_filter** for the context of header_filter_by_lua*.
  - **body_filter** for the context of body_filter_by_lua*.
  - **log** for the context of log_by_lua*.
  - **timer** for the context of user callback functions for ngx.timer.*.

Lua FFI

极客时间

- **提供一种Lua语言调用C语言函数的功能**
  - ffi.cdef声明C语言中的函数或者结构体
  - ffi.C调用声明说的方法
  - tonumber为Lua语言标准库函数
  - _M为Lua的数据结构：表
  - #s是Lua语法，表示取得字符串s长度
  - ngx_atoi是Nginx中的工具函数，用于将字符串转换为数字

lua-resty-string: string.lua

```
local ffi = require "ffi"
local C = ffi.C
local tonumber = tonumber

local _M = { _VERSION = '0.11' }

ffi.cdef[[
intptr_t ngx_atoi(const unsigned char *line, size_t n);
]]

function _M.atoi(s)
    return tonumber(C.ngx_atoi(s, #s))
end

return _M
```

```
typedef intptr_t        ngx_int_t;
#define u_char          unsigned char
ngx_int_t  ngx_atoi(u_char *line, size_t n)  { …}
```

系统级配置指令

极客时间

- **lua_malloc_trim**
  - **context**: http
  - 每N个请求使用C库的malloc_trim方法，将缓存的空闲内存归还操作系统。该功能是在11个阶段中的log阶段判断是否需要执行的。值为0时表示关闭该功能
- **lua_code_cache**
  - **context**: http, server, location, location if
  - 值为on表示Lua VM由所有请求共享，值为off表示每个请求使用独立的Lua VM
- **lua_package_path**
  - **context**: http
  - 设置调用Lua模块的路径地址
- **lua_package_cpath**
  - **context**: http
  - 设置Lua调用C模块的路径地址

取得配置参数SDK

极客时间

- **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, init_by_lua*, init_worker_by_lua*
- ngx.config.subsystem
  - 获取Nginx子模块，例如http{}下会返回http，而stream{}下会返回stream
- ngx.config.debug
  - 以布尔值返回当前Nginx编译执行configure时是否加入--with-debug
- ngx.config.prefix
  - 返回Nginx执行时的prefix路径，由命令行的-p或者configure --prefix指定的路径
- ngx.config.nginx_version
  - 以整型返回Nginx版本，例如1.13.6会返回1013006
- ngx.config.nginx_configure
  - **context:** *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, init_by_lua**
  - 返回编译时执行configure命令的参数
- ngx.config.ngx_lua_version
  - **context:** *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, init_by_lua**
  - 以整型返回ngx_http_lua_module版本号，例如10013（ngx_http_lua_version宏）

读取、修改变量的SDK

- **ngx.var.VAR_NAME**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*
  - 根据变量名VAR_NAME读取Nginx中的变量值（包括set定义的变量，以及Nginx框架生成的变量，例如http_HEADER或者arg_NAME）。也可修改变量的值，但必须是已经存在的变量。

客户端提前关闭连接

极客时间

- **lua_check_client_abort**
  - **context**: http, server, location, location-if
  - 是否检查客户端关闭连接，且发现该事件后回调ngx.on_abort指定的Lua方法。默认关闭

- ok, err = ngx.on_abort(callback)
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 当下游客户端过早关闭连接时，调用callback函数

获取请求头部的SDK

- **ngx.req.get_headers**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*
  - 以Lua table的方式返回当前请求中的所有请求头部。最多返回100个头部
- **ngx.req.raw_header（no_request_line）**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*
  - 返回网络中接收到的原始HTTP请求头部（包括请求行）。如果不包含请求行参数传递true

```
no_request_line=false
GET /t HTTP/1.1
Host: localhost
Connection: close
Foo: bar
```

```
no_request_line=true

Host: localhost
Connection: close
Foo: bar
```

获取请求URL的SDK

- **ngx.req.get_method**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, balancer_by_lua*, log_by_lua*
  - 以字符串形式返回当前请求的方法名，例如“GET”
- **ngx.req.http_version**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*
  - 返回请求中的HTTP版本号，例如1.1、2.0等
- **ngx.req.get_uri_args**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, balancer_by_lua*
  - 以Lua table的方式返回当前请求的全部URI参数，若URI参数只有名没有值，则该值为true。最多返回100个参数
- **ngx.arg[index]**
  - **context**: set_by_lua*, body_filter_by_lua*
  - 按index索引号读取到用户请求中的参数

获取请求包体的SDK

极客时间

•指令：lua_need_request_body

- **context**: http, server, location, location if
- Nginx在HTTP11个阶段中都不确保读取到完整的请求包体，而该指令可以强制要求在以上阶段Lua代码执行前读取完整的请求包体。默认关闭

SDK：

- **ngx.req.get_post_args**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*
  - 以Lua table的方式返回POST请求中的表单数据，最多返回100个成员
- **ngx.req.read_body**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 以同步的方式读取当前请求的包体。当请求没有包体、包体已经被读取或者包体被丢弃时，该方法会立即返回。读取到的包体，需要通过get_body_data或者get_body_file获取到内容
- **ngx.req.get_body_data**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*, log_by_lua*
  - 以Lua字符串形式返回内存中的包体。当请求包体未被读取、请求包体被存放在磁盘文件中、请求包体不存在时，返回nil
- **ngx.req.get_body_file**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 返回磁盘中存放请求包体的文件名，当包体未被读取或者未被存放至文件时，返回nil

HTTP请求方法名常量

- ngx.HTTP_GET
- ngx.HTTP_HEAD
- ngx.HTTP_PUT
- ngx.HTTP_POST
- ngx.HTTP_DELETE
- ngx.HTTP_OPTIONS (added in the v0.5.0rc24 release)
- ngx.HTTP_MKCOL (added in the v0.8.2 release)
- ngx.HTTP_COPY (added in the v0.8.2 release)
- ngx.HTTP_MOVE (added in the v0.8.2 release)
- ngx.HTTP_PROPFIND (added in the v0.8.2 release)
- ngx.HTTP_PROPPATCH (added in the v0.8.2 release)
- ngx.HTTP_LOCK (added in the v0.8.2 release)
- ngx.HTTP_UNLOCK (added in the v0.8.2 release)
- ngx.HTTP_PATCH (added in the v0.8.2 release)
- ngx.HTTP_TRACE (added in the v0.8.2 release)

修改请求信息的SDK

极客时间

- **ngx.req.set_method(method_id)**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*
  - 以HTTP方法变量的方式设置当前请求的方法
- **ngx.req.set_header(name , value)**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*
  - 修改或者删除当前请求的请求头部。当value为nil时，表示删除该头部
- **ngx.req.clear_header(header_name)**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*
  - 删除当前请求的某个请求头部，与ngx.req.set_header中值为nil效果相同
- **ngx.req.set_uri_args(args)**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*
  - 设置当前请求的URI参数，可传入字符串（如ngx.req.set_uri_args(“a=3&b=hello%20world”) ）或者传入Lua table（如ngx.req.set_uri_args({ a = 3, b = "hello world" }) ）

修改请求包体的SDK

极客时间

- **ngx.req.discard_body**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 丢弃接收到的请求包体
- **ngx.req.set_body_data**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 设置当前请求的请求包体内容。必须先读取完请求包体
- **ngx.req.set_body_file(filename, autoclean)**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 以磁盘文件来设置请求中的包体。必须先读取完请求包体。若autoclean值为true，则请求执行完后会删除这个磁盘文件，该值默认为false
- **ngx.req.init_body(buffer_size)**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 初始化一个空缓冲区，用于append_body和finish_body使用，创建用户请求包体。若buffer_size没有传递，则使用Nginx官方提供的client_body_buffer_size值作为缓冲区大小。
- **ngx.req.append_body**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 向缓冲区中写入请求包体内容。若添加的包体大小超出缓冲区，则将包体写入临时文件中
- **ngx.req.finish_body**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 使用init_body和append_body添加完包体后，必须调用该方法

修改URI并跳转的SDK

极客时间

- **ngx.req.is_internal**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*
  - 以布尔值返回当前请求是否为内部跳转过来的请求
- **ngx.req.set_uri(uri, jump)**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*
  - 修改当前请求的URI。jump默认为false，若为true且在rewrite_by_lua*上下文中时则类似rewrite指令进行location跳转
- **ngx.exec**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 更改当前请求的URI并执行location内部跳转

HTTP响应状态码常量

极客时间

- **HTTP响应状态码常量**
  - **context**: init_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, balancer_by_lua*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*, ssl_session_store_by_lua*
  - HTTP响应的状态码

- value = ngx.HTTP_CONTINUE (100) (first added in the v0.9.20 release)
- value = ngx.HTTP_SWITCHING_PROTOCOLS (101) (first added in the v0.9.20 release)
- value = ngx.HTTP_OK (200)
- value = ngx.HTTP_CREATED (201)
- value = ngx.HTTP_ACCEPTED (202) (first added in the v0.9.20 release)
- value = ngx.HTTP_NO_CONTENT (204) (first added in the v0.9.20 release)
- value = ngx.HTTP_PARTIAL_CONTENT (206) (first added in the v0.9.20 release)
- value = ngx.HTTP_SPECIAL_RESPONSE (300)
- value = ngx.HTTP_MOVED_PERMANENTLY (301)
- value = ngx.HTTP_MOVED_TEMPORARILY (302)
- value = ngx.HTTP_SEE_OTHER (303)
- value = ngx.HTTP_NOT_MODIFIED (304)
- value = ngx.HTTP_TEMPORARY_REDIRECT (307) (first added in the v0.9.20 release)
- value = ngx.HTTP_PERMANENT_REDIRECT (308)
- value = ngx.HTTP_BAD_REQUEST (400)
- value = ngx.HTTP_UNAUTHORIZED (401)
- value = ngx.HTTP_PAYMENT_REQUIRED (402) (first added in the v0.9.20 release)
- value = ngx.HTTP_FORBIDDEN (403)
- value = ngx.HTTP_NOT_FOUND (404)
- value = ngx.HTTP_NOT_ALLOWED (405)
- value = ngx.HTTP_NOT_ACCEPTABLE (406) (first added in the v0.9.20 release)
- value = ngx.HTTP_REQUEST_TIMEOUT (408) (first added in the v0.9.20 release)
- value = ngx.HTTP_CONFLICT (409) (first added in the v0.9.20 release)
- value = ngx.HTTP_GONE (410)
- value = ngx.HTTP_UPGRADE_REQUIRED (426) (first added in the v0.9.20 release)
- value = ngx.HTTP_TOO_MANY_REQUESTS (429) (first added in the v0.9.20 release)
- value = ngx.HTTP_CLOSE (444) (first added in the v0.9.20 release)
- value = ngx.HTTP_ILLEGAL (451) (first added in the v0.9.20 release)
- value = ngx.HTTP_INTERNAL_SERVER_ERROR (500)
- value = ngx.HTTP_METHOD_NOT_IMPLEMENTED (501)
- value = ngx.HTTP_BAD_GATEWAY (502) (first added in the v0.9.20 release)
- value = ngx.HTTP_SERVICE_UNAVAILABLE (503)
- value = ngx.HTTP_GATEWAY_TIMEOUT (504) (first added in the v0.3.1rc38 release)
- value = ngx.HTTP_VERSION_NOT_SUPPORTED (505) (first added in the v0.9.20 release)
- value = ngx.HTTP_INSUFFICIENT_STORAGE (507) (first added in the v0.9.20 release)

读取、修改响应值

极客时间

- **ngx.status**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*
  - 读取响应值，在响应头部发出以前可以修改响应值

修改发送响应的头部

极客时间

- **ngx.header.HEADER或者ngx.header[‘HEADER’]**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*
  - 增、删、改、查当前请求中的响应头部。注意lua_transform_underscores_in_response_headers指令是默认开启中的。
  - 其值为nil时（或者值为{}）表示删除该头部。当头部不存在时，则返回nil

•指令：lua_transform_underscores_in_response_headers

•**context**: http, server, location, location-if

•当我们通过SDK中的ngx.header系列API去设置、获取头部时，是否自动将头部名称（不包括值）中的’_’更换为’-’，默认是开启的

发送响应的SDK

- **ngx.redirect(uri, status?)**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 向客户端返回重定向请求，status默认为302
- **ngx.send_headers**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 显式的向客户端发送响应头部，返回1表示成功，返回nil表示失败
- **ngx.headers_sent**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 返回布尔值标识响应头部是否已经发送

请求响应类指令

极客时间

- **lua_use_default_type**
  - **context**: http, server, location, location if
  - 指定是否使用Nginx框架提供的default_type中定义的Content-Type类型，作为Lua代码的响应类型。默认开启。
- **lua_http10_buffering**
  - **context**: http, server, location, location-if
  - 控制对于Lua生成的、HTTP/1.0协议的响应是否先缓存再发送，当Lua代码中显示设定了Content-Length头部时会自动关闭该缓存功能

发送响应的SDK

- **ngx.print**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 通过写入响应包体向下游客户端发送响应。该方法是异步的，它会立刻返回而不等待所有响应发送完毕，在其后使用ngx.flush(true)可以同步等待发送成功
- **ngx.say**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 与ngx.print相似，但在包体最后会加入换行符
- **ngx.flush(wait?)**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 将响应发向客户端，wait参数默认为false，此时flush方法是异步的，即它返回时所有响应未必都已经传递给内核的发送缓冲区；当值为true时，flush方法是同步，它返回时要么响应发送完毕，要么超时
- **ngx.exit(status)**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, ngx.timer.*, balancer_by_lua*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*, ssl_session_store_by_lua*
  - 当status>=200时，exit会终止当前请求的Lua代码，通过向Nginx返回响应码来发送响应
  - 当status==0时，exit会终止当前阶段（参见HTTP11个阶段）的执行，继续后续阶段的执行
- **ngx.eof**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 显式的指定响应内容已经结束。对于HTTP/1.1（RFC2616）中定义的chunked编码来说，eof会指定Nginx发出“last chunk”来指明响应结束

Nginx函数返回值常量

极客时间

- Nginx函数返回值常量
  - ngx.OK (0)
  - ngx.ERROR (-1)
  - ngx.AGAIN (-2)
  - ngx.DONE (-4)
  - ngx.DECLINED (-5)

取worker进程信息SDK

- **ngx.worker.exiting**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, init_by_lua*, init_worker_by_lua*
  - 以布尔值返回当前Nginx worker进程是否正在退出
- **ngx.worker.pid**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, init_by_lua*, init_worker_by_lua*
  - 返回worker进程ID，由于不依赖变量，所以比ngx.var.pid应用范围更广
- **ngx.worker.count**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, init_by_lua*, init_worker_by_lua*
  - 返回nginx.conf中配置的Nginx worker进程的数量
- **ngx.worker.id**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, init_worker_by_lua*
  - 返回所属worker进程的序号（从0到N-1）

日志级别常量

极客时间

- **日志级别常量**
  - **context**: init_by_lua*, init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, balancer_by_lua*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*, ssl_session_store_by_lua*
    - ngx.STDERR
    - ngx.EMERG
    - ngx.ALERT
    - ngx.CRIT
    - ngx.ERR
    - ngx.WARN
    - ngx.NOTICE
    - ngx.INFO
    - ngx.DEBUG

写入日志的SDK

极客时间

- **print(…)**
  - **context**: init_by_lua*, init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, balancer_by_lua*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*, ssl_session_store_by_lua*
  - 使用notice级别将日志写入error.log文件中
- **ngx.log(log_level, …)**
  - **context**: init_by_lua*, init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, balancer_by_lua*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*, ssl_session_store_by_lua*
  - 以指定的日志级别log_level来写入日志至error.log文件中

获取error.log日志内容的SDK

极客时间

- **lua_capture_error_log**
  - **context**: http
  - 基于Nginx提供的ngx_cycle_t->ngx_log_intercept_pt机制，使用API中的get_logs（由lua-resty-core模块提供）直接在内存中获取到Nginx所有输出至error.log中的日志内容。该指令定义了缓存大小

- **get_logs**

```
local errlog = require "ngx.errlog“
local res = errlog.get_logs(10)
```

建立上下文信息的SDK

- **ngx.ctx**
  - **context**: init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, balancer_by_lua*
  - 针对每个HTTP请求在内存中建立上下文字典，可通过名称设置、读取值。子请求、内部跳转后皆不能使用ctx上下文

编解码

极客时间

- **context:** init_by_lua*, init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, balancer_by_lua*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*, ssl_session_store_by_lua*
- ngx.escape_uri
  - 将字符串进行URL编码
- ngx.unescape_uri
  - 将以URL编码的字符串解码
- ngx.encode_args
  - 将Lua table以URL格式中参数的形式编码，如{foo = 3, [“b r”] = “hello world”}编码为foo=3&b%20r=hello%20world
- ngx.decode_args
  - 将URL格式的参数解码为Lua table
- ngx.encode_base64
  - 将字符串编码为base64格式
- ngx.decode_base64
  - 将base64格式的字符串解码为原字符串
- ngx.quote_sql_str
  - 将SQL语句字符串转换为Mysql格式

哈希编码SDK

极客时间

- **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, balancer_by_lua*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*, ssl_session_store_by_lua*
- ngx.crc32_short(str)
  - 计算字符串的CRC32编码，该方法适用于较短的字符串（小于30~60字节）
- ngx.crc32_long
  - 计算字符串的CRC32编码，该方法适用于较长的字符串（大于30~60字节）
- ngx.hmac_sha1(secret_key, str)
  - 返回字符串的SHA1摘要值（依赖Openssl库）
- ngx.md5(str)
  - 返回字符串的MD5摘要值
- ngx.md5_bin
  - 返回字符串的二进制格式的MD5摘要值
- ngx.sha1_bin
  - 返回字符串的二进制格式的SHA1摘要值

正则表达式SDK（1）

极客时间

- **ngx.re.match**
  - **context**: init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, balancer_by_lua*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*, ssl_session_store_by_lua*
  - 进行正则表达式匹配，返回匹配中的子字符串
- **ngx.re.find**
  - **context**: init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, balancer_by_lua*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*, ssl_session_store_by_lua*
  - 进行正则表达式匹配，返回匹配中的子字符串中的第1个、最后1个字符的索引。未创建新字符串
- **ngx.re.gmatch**
  - **context**: init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, balancer_by_lua*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*, ssl_session_store_by_lua*
  - 进行正则表达式匹配，但返回的是迭代器，需要通过迭代器取得所有匹配的子字符串

正则表达式SDK（2）

极客时间

- **ngx.re.sub**
  - **context**: init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, balancer_by_lua*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*, ssl_session_store_by_lua*
  - 进行正则表达式匹配，并可将匹配中的子字符串替换为新的字符串
- **ngx.re.gsub**
  - **context**: init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, balancer_by_lua*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*, ssl_session_store_by_lua*
  - 进行正则表达式匹配，并以全局方式将匹配中的子字符串替换为新的字符串

正则表达式相关指令

极客时间

- **lua_regex_match_limit**
  - **context**: http
  - 限制ngx.re正则表达式类API匹配中的最大组数，默认值0表示使用PCRE库编译时设定的最大值
- **lua_regex_cache_max_entries**
  - **context**: http
  - 对于以下API：ngx.re.match, ngx.re.gmatch, ngx.re.sub, ngx.re.gsub所生成的正则表达式缓存至内存中，该指令定义了缓存的最大条目数。当达到最大数目后，新的正则表达式不会被缓存。默认值1024

cosocket相关指令

极客时间

- **context**: http, server, location
- lua_socket_connect_timeout
  - 设置SDK中的cosocket的连接超时时间，默认60秒，可以携带所有时间类单位
- lua_socket_send_timeout
  - 设置SDK中的cosocket的两次写操作间的超时时间，默认60秒，可以携带所有时间类单位
- lua_socket_read_timeout
  - 设置SDK中的cosocket的两次读操作间的超时时间，默认60秒，可以携带所有时间类单位
- lua_socket_buffer_size
  - 设置cosocket中的读缓冲区大小，默认为4K/8K
- lua_socket_pool_size
  - 设置每个cosocket连接池中的最大连接数（每个worker进程），超出后使用LRU算法关闭、淘汰连接
- lua_socket_keepalive_timeout
  - 设置每个cosocket连接的最大空闲时间（类似HTTP的Keepalive），默认60秒
- lua_socket_log_errors
  - 控制cosocket出错时是否记录错误日志至Nginx的error.log文件中

TCP cosocket（1）

极客时间

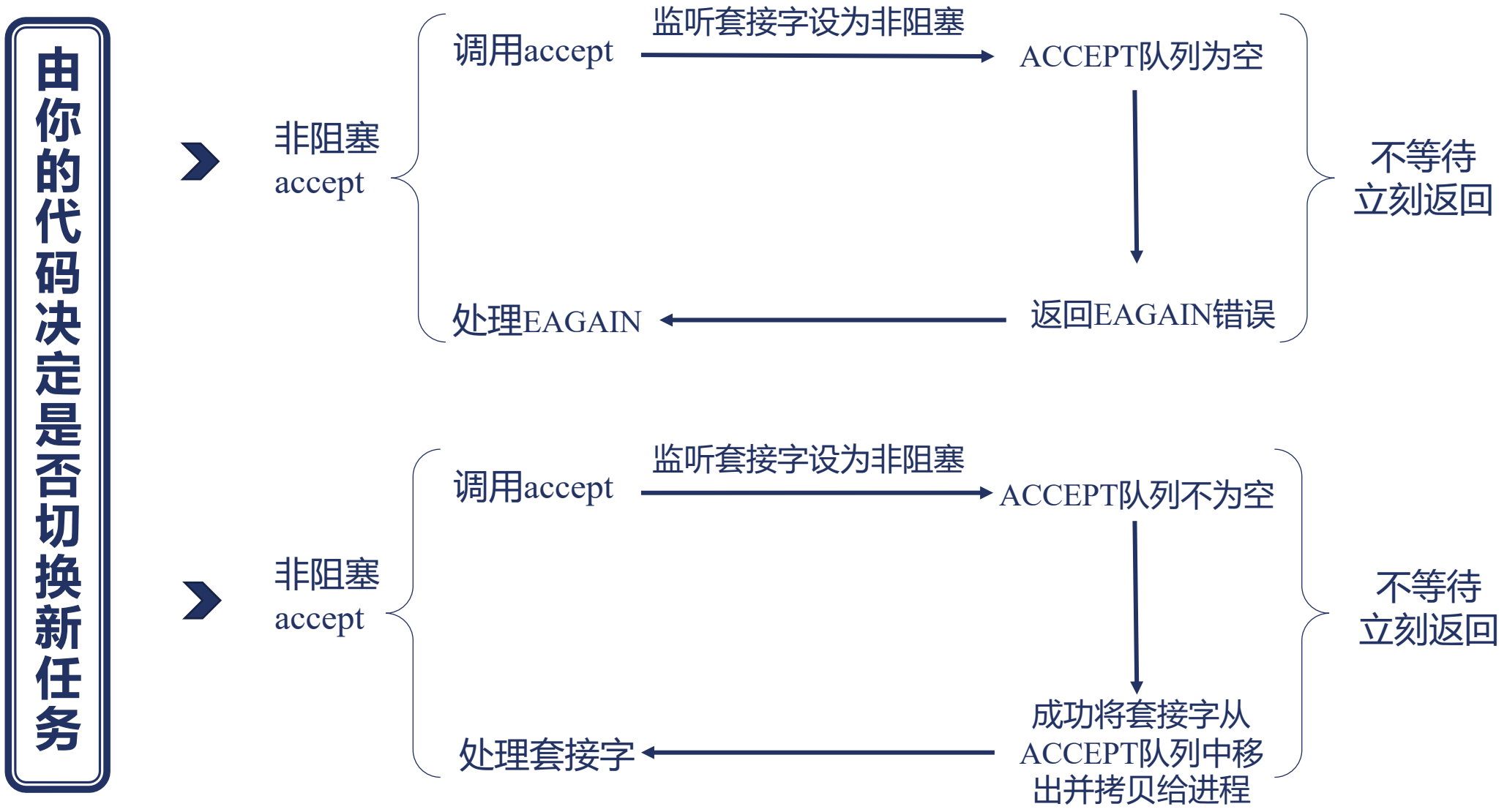
- **ngx.socket.tcp**
  - rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*
  - 创建TCP协议的cosocket对象。当没有显示关闭cosocket或者把它放入连接池，则当前阶段结束时会自动关闭对象
  - connect
    - 连接上游服务（IP端口或者unix地址）
  - sslhandshake
    - 在已经建立好的TCP连接上进行SSL握手
  - send
    - 将消息发送到对端。在send方法返回时，它已经将用户态的字节流拷贝到内核socket缓冲区中
  - receive(size)或者receive(pattern?)
    - 自cosocket中接收size个字节的消息，未接收足时则不会返回，直到满足超时条件（由settimeout、settimeouts方法或者lua_socket_read_timeout指令定义）
    - pattern为*a表示一直接收消息 ，直到连接关闭
    - pattern为*l表示接收一整行消息 ，直到收到LF为止。没有参数时等同*l
  - receiveany(max)
    - 自cosocket中接收到任意数据即返回，最多接收max字节（若内核读缓冲区数据超过max，仍只返回max字节数据）

TCP cosocket（2）

极客时间

- **ngx.socket.tcp**
  - receiveuntil(pattern, options?)
    - 返回用于读取消息的Lua迭代器方法，直到pattern匹配到接收到的字节流
  - close
    - 关闭TCP连接
  - settimeout(microseconds)
    - 设置超时时间，该时间同时应用在connect、send、receive方法上，参数单位为毫秒
  - settimeouts(connect_timeout, send_timeout, read_timeout)
    - 分别设置connect、send、receive方法的超时时间，单位毫秒
  - setkeepalive(timeout?, size?)
    - 关闭cosocket对象，并将当前TCP连接放入keepalive连接池，最长空闲时间为timeout（单位毫秒），设为0表示永不过期，若忽略该参数则使用lua_socket_keepalive_timeout指令的值。
    - size指定了连接池中对当前上游服务的最大连接数（仅第一次创建该上游服务的连接池时有效），忽略该参数时使用lua_socket_pool_size指令的值。达到size上限后（每worker进程内），按LRU关闭较老的空闲连接
    - 若内核读缓冲区仍有数据，则该方法会返回错误信息“connection in dubious state”
  - getreusedtimes
    - 返回当前TCP连接复用了多少次

非阻塞调用


由你的代码决定是否切换新任务
非阻塞accept
调用accept
监听套接字设为非阻塞
ACCEPT队列为空
返回EAGAIN错误
处理EAGAIN
不等待立刻返回
非阻塞accept
调用accept
监听套接字设为非阻塞
ACCEPT队列不为空
成功将套接字从ACCEPT队列中移出并拷贝给进程
处理套接字
不等待立刻返回

非阻塞调用下的同步与异步

极客时间

Openresty的同步调用代码

```
local client = redis:new()
 client:set_timeout(30000)
 local ok,err = client:connect(ip,port)
 if not ok then
        ngx.say("failed: ",err)
        return
 end
```

这是标准的异步调用

```
rc = ngx_http_read_request_body ( r, ngx_http_upstream_init) ;
if ( rc > = NGX_HTTP_SPECIAL_RESPONSE) {
    return rc ;
}
```

这个方法执行完时调用post_handler异步方法

```
ngx_int_t
ngx_http_read_client_request_body (ngx_http_request_t  * r,
    ngx_http_client_body_handler_pt  post_handler)
```

最终读取完body后调用ngx_http_upstream_init方法

```
void
ngx_http_upstream_init (ngx_http_request_t  * r )
{
```

获取客户端的TCP cosocket

极客时间

- **ngx.req.socket**
  - context: rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 将与下游客户端间的socket连接，以只读cosocket对象的形式封装后返回（仅支持receive和receiveuntil方法）

SSL相关指令

极客时间

- **lua_ssl_ciphers**
  - **context**: http, server, location
  - 定义cosocket中所使用的SSL安全套件
- **lua_ssl_crl**
  - **context**: http, server, location
  - 指定SSL中的CRL吊销证书列表
- **lua_ssl_protocols**
  - **context**: http, server, location
  - 指定cosocket中的SSL协议版本号
- **lua_ssl_trusted_certificate**
  - **context**: http, server, location
  - 指定验证cosocket中对端证书的可信证书文件
- **lua_ssl_verify_depth**
  - 指定验证cosocket中对端证书的证书链深度

UDP cosocket

极客时间

- **ngx.socket.udp**
  - rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*
  - 创建UDP协议的cosocket对象，支持5个方法
  - setpeername
    - 设置对端节点的地址（IP端口或者unix地址）
  - send
    - 将消息通过UDP协议的cosocket发送到对端
  - receive(size?)
    - 自cosocket中接收最多size个字节的消息，默认size为8192字节。若在规定的超时时间内（由settimeout方法或者lua_socket_read_timeout指令定义）未接收到，则返回nil
  - close
    - 关闭cosocket对象
  - settimeout(microseconds)
    - 设置receive读取方法的超时时间，参数单位为毫秒

协程SDK（1）

极客时间

- **co = ngx.thread.spawn(func, arg1, arg2, ...)**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*
  - 生成轻量级线程并由nginx及ngx_http_lua_module模块立刻调度执行。
  - 入参func为线程中执行的函数，arg为func需要的参数，返回对象为Lua thread
- **ok, res1, res2, ... = ngx.thread.wait(thread1, thread2, ...)**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*
  - 等待1个或者多个轻量级线程执行完毕后返回，ok表示所有线程是否正常结束 ，接下来的res返回值按照次序是每个func的返回值
- **ok, err = ngx.thread.kill(thread)**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.*
  - 杀死正在运行的轻量级线程，仅父线程可以执行

协程SDK（2）

极客时间

- **coroutine**
  - context: rewrite_by_lua*, access_by_lua*, content_by_lua*, init_by_lua*, ngx.timer.*, header_filter_by_lua*, body_filter_by_lua*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*, ssl_session_store_by_lua*
  - **create**
    - 创建Lua协程，它不会自动执行
  - **resume**
    - 执行已创建还未执行的协程，或者恢复执行被yield挂起的协程
  - **yield**
    - 挂起当前执行的协程
  - **wrap**
    - 创建Lua协程，可通过调用返回函数实现在协程中执行传入的方法（类似resume）
  - **running**
    - 返回正在执行的协程对象
  - **status**
    - 返回协程的状态，包括running、suspended、normal、dead

同步非阻塞的sleep方法及lua-resty-lock锁

极客时间

- **ngx.sleep(seconds)**
  - context: rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*
  - 对当前Lua代码块睡眠一段时间，seconds可以精确到0.001（即毫秒）。它是通过Nginx中的定时器实现，不会阻塞Nginx运行

lua-resty-lock模块的方法

- **obj, err = lock:new(dict_name, opts)**
  - 自共享内存字典中新建锁对象
  - opts选项包括
    - exptime：设置锁的过期时间，到期后自动释放锁。默认30秒，精确到毫秒
    - timeout：lock方法的最大等待时间，默认5秒，精确到毫秒
    - step：锁是由ngx.sleep方法实现的，step定义了第一次sleep的时间。默认0.001秒
    - ratio：定义每轮sleep时间的增长率，默认为2
    - max_step：定义了最大sleep时间，默认为0.5秒
- **elapsed, err = obj:lock(key)**
  - 获取关于key的锁，返回等待的时间，如果未获取到锁则返回nil
- **ok, err = obj:unlock()**
  - 释放锁
- **ok, err = obj:expire(timeout)**
  - 重置new方法传入的timeout选项

定时器SDK

极客时间

- **hdl, err = ngx.timer.at(delay, callback, user_arg1, user_arg2, ...)**
  - context: init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, balancer_by_lua*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*, ssl_session_store_by_lua*
  - 新增定时器，定时器触发后执行callback函数，user_arg是函数的参数。
  - delay是延迟执行时间，单位为秒，精确到毫秒。0表示立刻在后台的轻量级线程中执行callback
  - callback的第一个参数是布尔值的premature，它表示定时器是否过早被触发，例如Nginx worker进程在关闭时并不会等待定时器触发，而是直接调用callback，此时premature值为true，接下来才是user_arg等参数
  - callback由于没有当前请求，故不能使用子请求类API（如ngx.location.capture），也不能获取当前请求的参数，如ngx.req.*
- **ngx.timer.every**
  - 类似at，相当于每daley秒就调用一次callback
- **ngx.timer.running_count**
  - 当前正在执行的定时器数量
- **ngx.timer.pending_count**
  - 等待执行的定时器数量

定时器相关指令

极客时间

- **lua_max_pending_timers**
  - **context**: http
  - 指定API：ngx.timer.at中等待执行的定时器的最大数目，达到后新增的定时器直接返回nil，默认值1024
- **lua_max_running_timers**
  - **context**: http
  - 指定API：ngx.timer.at中正在执行回调方法的定时器最大数目，达到后新增的定时器的回调方法不会被执行，在error.log中出现N lua_max_running_timers are not enough字样的日志，默认256

获取请求的处理时间SDK

- **ngx.req.start_time**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*
  - 返回开始处理当前请求经过的时间

时间类SDK

极客时间

- **context**: init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, balancer_by_lua*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*, ssl_session_store_by_lua*
- **ngx.today**
  - 从Nginx中取当前时间的日期（格式为yyyy-mm-dd）
- **ngx.time**
  - 取得自epoch时间（1970年1月1日0点）到现在的秒数
- **ngx.now**
  - 取得自epoch时间（1970年1月1日0点）到现在的秒数，精度到0.001（毫秒）
- **ngx.update_time**
  - 更新Nginx的缓存时间
- **ngx.localtime**
  - 返回本地时区下的当前时间（格式为yyyy-mm-dd hh:mm:ss）
- **ngx.utctime**
  - 返回UTC下的当前时间（格式为yyyy-mm-dd hh:mm:ss）
- **ngx.cookie_time(sec)**
  - 将epoch时间秒数转换为cookie中可以识别的时间格式，例如ngx.cookie_time(1547466623）返回Mon, 14-Jan-19 11:50:23 GMT
- **ngx.http_time(sec)**
  - 将epoch时间秒数转换为HTTP头部中的标准时间格式，例如ngx.http_time(1547466623）返回Mon, 14 Jan 2019 11:50:23 GMT:2019-01-14
- **ngx.parse_http_time(str)**
  - 将HTTP头部标准时间格式的字符串转换为epoch时间秒数

基于共享内存的字典shared_dict

极客时间

- **指令**
  - lua_shared_dict
    - **context**: http
    - 基于Nginx的共享内存（使用Slab管理器）实现的跨worker进程字典容器，支持LRU淘汰功能。由于reload不会清除共享内存中的内容，故reload后shared_dict值仍存在
- **SDK**
  - 共享内存的所有方法都是原子的、线程安全的
  - context: init_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, balancer_by_lua*, ssl_certificate_by_lua*, ssl_session_fetch_by_lua*, ssl_session_store_by_lua*

shared_dict的SDK（1）

极客时间

- **value, flags = ngx.shared.DICT:get(key)**
  - 返回字典中未过期key的值，返回值如下：
    - value：返回关键字的值，如果不存在或者已经过期，则返回nil
    - flags：若set设置时flag为0则返回nil，否则返回设置时的flags
- **value, flags, stale = ngx.shared.DICT:get_stale(key)**
  - 返回字典中key的值（无论是否过期），返回值如下：
    - value：返回关键字的值，如果不存在则返回nil，过期仍然返回其值
    - flags：若set设置时flag为0则返回nil，否则返回设置时的flags
    - stale：如果关键字已经过期，则stale为true，否则为false
- **success, err, forcible = ngx.shared.DICT:set(key, value, exptime?, flags?)**
  - 向字典中设置键值对，当内存不足时会按过期时间淘汰老的元素。当连续淘汰30个（当前版本）元素后仍然无法分配出足够内存，则返回失败
    - key,value：键值对
    - exptime：过期时间，单位为秒，可精确到毫秒。默认为0表示永不过期（会被LRU淘汰）
    - flags：用于区别不同的键。默认为0，以无符号32位整型存储
    - success：以布尔值返回成功或者失败
    - err：错误信息
    - forcible：若是由于LRU强制淘汰较老的键才为本次设置成功分配内存，则为true
- **ok, err = ngx.shared.DICT:safe_set(key, value, exptime?, flags?)**
  - 类似set方法，但不会在内存不足时根据LRU强行淘汰未过期的键。内存不足时，返回nil和”no memory”

shared_dict的SDK（2）

极客时间

- **success, err, forcible = ngx.shared.DICT:add(key, value, exptime?, flags?)**
  - 类似set，但仅向字典中添加新的键值对，若key已经存在则返回false,"exists",false
- **ok, err = ngx.shared.DICT:safe_add(key, value, exptime?, flags?)**
  - 类似add，但不会在内存不足时根据LRU强行淘汰未过期的键。内存不足时，返回nil和"no memory"
- **success, err, forcible = ngx.shared.DICT:replace(key, value, exptime?, flags?)**
  - 类似set，但仅能覆盖字典中已经存在的键值。若key不存在 ，则返回false,"not found",false
- **ngx.shared.DICT:delete(key)**
  - 从字典中删除key及对应的值，等价于set(key, nil)
- **newval, err, forcible? = ngx.shared.DICT:incr(key, value, init?, init_ttl?)**
  - 如果字典中key对应的值存在 ，则将其值加上value，并返回相加后的值newval。若key在字典中对应的值不是数字，则返回nil,"not a number",false
    - init：当key不存在时，使用init+value作为该key的值添加至字典
    - init_ttl：当init生效时，指定key的过期时间
    - forcible：当init未设置时，由于不会添加新键值，故其值始终为nil

shared_dict的SDK（3）

- **length, err = ngx.shared.DICT:lpush(key, value)**
  - 设置key对应的值为队列，从队列的头部插入元素value。返回length为插入后队列中的所有元素个数。它不会引发LRU淘汰
  - 如果key不存在，则插入一个仅含有元素value的队列。
  - 如果key存在，但其值不是一个队列，则返回nil,"value not list"
- **length, err = ngx.shared.DICT:rpush(key, value)**
  - 类似lpush，但却是从队列的尾部插入元素
- **val, err = ngx.shared.DICT:lpop(key)**
  - 从key对应的队列头部取出1个元素
- **val, err = ngx.shared.DICT:rpop(key)**
  - 从key对应的队列尾部取出1个元素
- **len, err = ngx.shared.DICT:llen(key)**
  - 查询key对应的队列的元素个数。若key对应的值不是队列，则返回nil,"value not a list"

shared_dict的SDK（4）

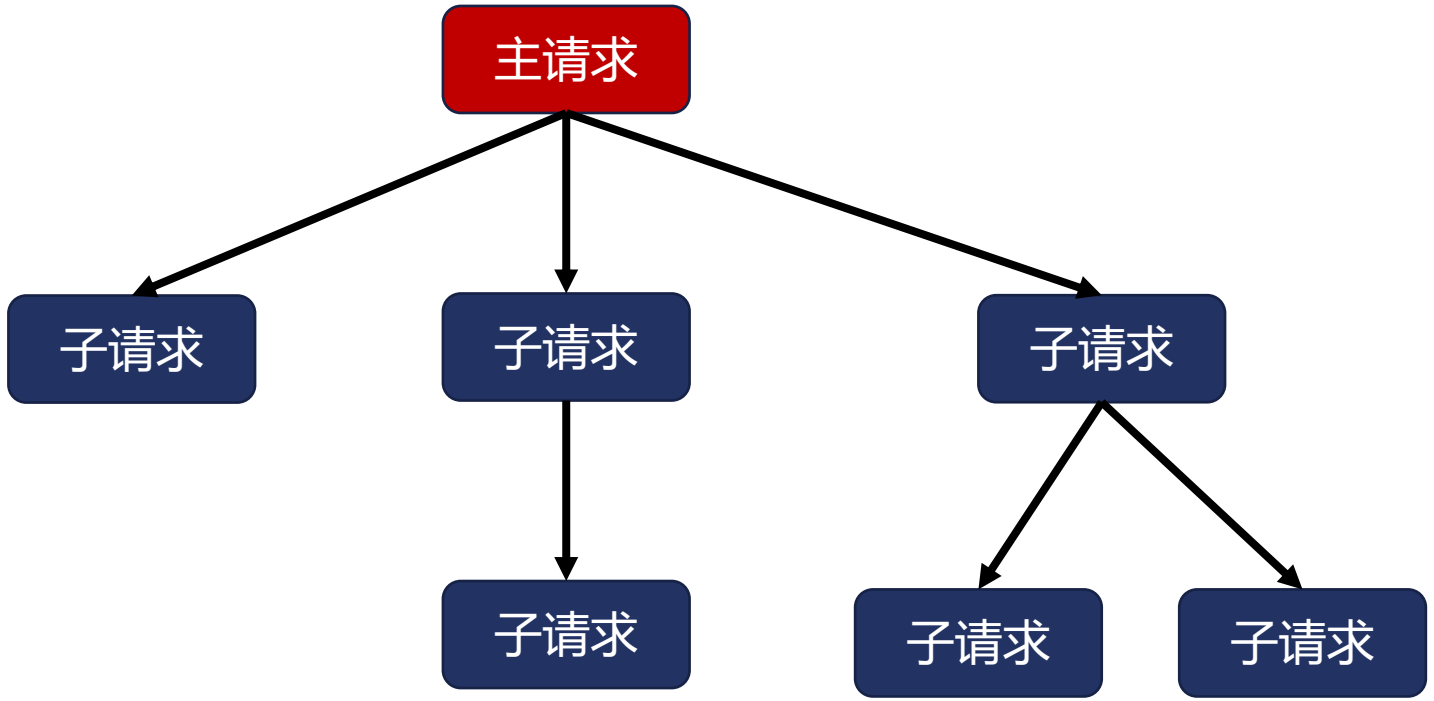
极客时间

- **ttl, err = ngx.shared.DICT:ttl(key)**
  - 返回key对应的元素距离过期时间的剩余秒数，如果是永不过期类型则返回ttl为0
- **success, err = ngx.shared.DICT:expire(key, exptime)**
  - 设置key对应元素的过期时间，如果key对应元素不存在则返回nil,”not found”。exptime为0则表示永不过期
- **ngx.shared.DICT:flush_all()**
  - 标记字典中所有元素皆为过期
- **flushed = ngx.shared.DICT:flush_expired(max_count?)**
  - 将字典中最多max_count(0表示不做限制)个过期元素释放内存
- **keys = ngx.shared.DICT:get_keys(max_count?)**
  - 将字典中最多max_count个键取出，max_count默认值1024，设置为0时表示取出所有的键
- **capacity_bytes = ngx.shared.DICT:capacity()**
  - 以字节数返回共享内存大小。须require "resty.core.shdict"
- **free_page_bytes = ngx.shared.DICT:free_space()**
  - 以字节数返回共享内存slab管理器中空闲页的大小。须require "resty.core.shdict"

Nginx中的主请求与子请求

由Nginx框架为模块开发提供的机制（《深入理解Nginx：模块开发与架构解析》第5章）

- **子请求的生命周期**
  - 依赖父请求
- **主请求与父子请求间的关系**
  - 主请求即客户端发来的请求
  - 主请求可派生多个子请求
  - 子请求也可派生多个子请求
  - 子请求完成时会激活父请求
- **子请求的响应如何处理？**
  - 作为父请求响应
    - postpone_filter过滤模块
  - 放置内存中等待模块处理




```
typedef struct ngx_http_request_s     ngx_http_request_t;
struct ngx_http_request_s {
                ngx_http_request_t            *main;//主请求
                ngx_http_request_t            *parent;//父请求
                。。。
}
```

Nginx中的主请求与子请求

极客时间

```
typedef struct ngx_http_request_s     ngx_http_request_t;
struct ngx_http_request_s {
            ngx_http_request_t          *main;//主请求
            ngx_http_request_t          *parent;//父请求
            。。。
}
```

location.capture子请求

极客时间

- **ngx.location.capture(uri, option)**
  - rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 生成Nginx子请求，且接下来的Lua代码会以同步的方式等待子请求返回后再执行。因此，该方法执行前应读取完整的请求包体，防止读取请求包体超时发生。子请求继承当前请求所有头部。
  - 返回值依赖子请求的响应，且响应都会放置在内存中，故子请求返回的响应不应过大，对于大文件响应可以使用cosocket。
    - 返回值为Lua table，包含4个成员：
      - status：响应码
      - header：以Lua table方式保存的响应头部。如果某个头部值属于多值头部，则其值返回Lua table，并以数组的方式按出现次序返回成员
      - body：响应包体，可能被截断
      - truncated：如果响应包体被截断，则为True
  - option选项
    - method：请求方法名，须传递HTTP请求方法名常量，默认为GET方法
    - body：设置请求包体，仅接受string类型的参数
    - args：设置URI参数
    - ctx：以Lua table方式设置子请求API：ngx.ctx中可使用的上下文字典
    - vars：以Lua table方式设置子请求中的变量
    - copy_all_vars：是否拷贝当前请求中的变量至子请求中
    - share_all_vars：是否共享当前请求中的变量至子请求中，修改子请求变量值会影响到当前请求
    - always_forward_body：若body选项未设置，其值为真时将当前请求包体作为子请求包体

location.capture子请求

极客时间

- **ngx.location.capture_multi({{uri, option},{uri, option},…})**
  - **context**: rewrite_by_lua*, access_by_lua*, content_by_lua*
  - 支持并行发起多个子请求，返回值代表的子请求响应与参数中的uri顺序相同
  - 仅所有子请求皆获得响应后，当前方法所属的代码段才会继续执行
- **ngx.is_subrequest**
  - **context**: set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*
  - 以布尔值返回当前请求是否为子请求

WAF(Web Application Firewall)防火墙

极客时间

- https://github.com/loveshell/ngx_lua_waf

```
lua_package_path "/usr/local/nginx/conf/waf/?.lua";
lua_shared_dict limit 10m;
init_by_lua_file /usr/local/nginx/conf/waf/init.lua;
access_by_lua_file /usr/local/nginx/conf/waf/waf.lua;
```

极客时间

扫码试看/订阅

《Nginx 核心知识100讲》