先上例子：此处是简化的配置，我们将在这个例子的基础上进行脚本引擎流程代码的梳理任务。

server {

if ( $request\_method = GET ) {

set $USER\_V1 "hello A";

}

location /abc {

set $USER\_V2 "B";

set $USER\_V3 $USER\_V1${USER\_V2}C;

echo $USER\_V3;

}

}

通过访问/abc将得到以下输出：

$ curl 192.168.124.129:8011/abc

hello ABC

这个例子，大家肯定都能预测到输出的结果是什么，但是这些指令它是怎样被nginx解析、处理和执行的？等等一系列的问题，初学者可能都会遇到这样的困惑。那么我们接下来将要分析：

1．Nginx如何解析这些指令

2．Nginx什么时候执行这些指令

一、我们将从nginx的指令解析开始码起。

Nginx解析的过程中会每行的逐一进行处理，每当遇到系统定义的指令时，将会调用相关处理函数。所以当遇到”if”这个系统定义的指令，将会停下来调用已经注册好的处理函数。代码中是这样定义的：

{ ngx\_string("if"),

NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_BLOCK|NGX\_CONF\_1MORE,

ngx\_http\_rewrite\_if,

NGX\_HTTP\_LOC\_CONF\_OFFSET,

0,

NULL }

所以将会调用 ngx\_http\_rewrite\_if()这个函数进行处理。接下来分析这个函数的处理流程。

static char \*

ngx\_http\_rewrite\_if(ngx\_conf\_t \*cf, ngx\_command\_t \*cmd, void \*conf)

{

…………………………………………………………..

…………………………………………………………..

// 类似于location指令的解析处理，申请一个存放各个模块的配置结构空间，nginx将if指令也当做一个location来处理（nginx为什么将if指令当做location来处理？？？）

ctx = ngx\_pcalloc(cf->pool, sizeof(ngx\_http\_conf\_ctx\_t));

if (ctx == NULL) {

return NGX\_CONF\_ERROR;

}

// 根据指令"if"所在位置的不同传入的cf->ctx指针也不同，如果"if"指令在server block{...}中ctx是server层的（即函数ngx\_http\_core\_server()申请的），如果在location blcok{...} 内为location层的。针对上边的例子来说，此时传入的cf->ctx是在server层的。（ctx中各个配置文件分层的作用？？？）

pctx = cf->ctx;

ctx->main\_conf = pctx->main\_conf; // 这里继承上一层的main\_conf、srv\_conf

ctx->srv\_conf = pctx->srv\_conf;

// 创建属于自己独立的loc\_conf（为什么要创建属于自己的ctx->loc\_conf？？？）

ctx->loc\_conf = ngx\_pcalloc(cf->pool, sizeof(void \*) \* ngx\_http\_max\_module);

if (ctx->loc\_conf == NULL) {

return NGX\_CONF\_ERROR;

}

for (i = 0; ngx\_modules[i]; i++) {

if (ngx\_modules[i]->type != NGX\_HTTP\_MODULE) {

continue;

}

module = ngx\_modules[i]->ctx;

if (module->create\_loc\_conf) {

mconf = module->create\_loc\_conf(cf);

if (mconf == NULL) {

return NGX\_CONF\_ERROR;

}

ctx->loc\_conf[ngx\_modules[i]->ctx\_index] = mconf;

}

}

// 获取上层的loc\_conf，针对上边例子是server层申请loc\_conf

pclcf = pctx->loc\_conf[ngx\_http\_core\_module.ctx\_index];

// 获取本层的loc\_conf

clcf = ctx->loc\_conf[ngx\_http\_core\_module.ctx\_index];

// 将”if”指令添加到locations队列做准备， 修改ngx\_http\_core\_loc\_conf\_t的字段loc\_conf 指向本层loc\_conf，在配置文件中每个location指令都对应一个ngx\_http\_core\_loc\_conf\_t结构

clcf->loc\_conf = ctx->loc\_conf;

// 将上层location的name 字段赋值给本层的name（举个例子：上层是location指令时，pclcf->name存放的是/abc，而此时if指令在server block中，所以pclcf->name=NULL）

clcf->name = pclcf->name;

// if block {...} 都被定义为noname类型

clcf->noname = 1;

// 加入到上层的locations队列中（即server层的）\*\*\*提示下：这里为什么需要加入到队列中，以及上边的ctx分层到底的作用和请求的过程中是怎样被调用的这些由于涉及的内容非常多，一篇文章很难讲清楚，所以后续的分析将会在这篇文章基础上进行讲解。

if (ngx\_http\_add\_location(cf, &pclcf->locations, clcf) != NGX\_OK) {

return NGX\_CONF\_ERROR;

}

// 解析条件表达式（即( $request\_method = GET )这部分内容），这里我们必须停止以下部分的解析来分析这个函数中做了哪些事情，分析完此函数后还需要回到这里进行以下代码的分析。：）

if (ngx\_http\_rewrite\_if\_condition(cf, lcf) != NGX\_CONF\_OK) {

return NGX\_CONF\_ERROR;

}

// 分析完上边的函数，这里就知道在做什么了。还是继续向lcf->codes数组中扔结构体。这里貌似换了一个类型ngx\_http\_script\_if\_code\_t，这个类型有三个字段。

if\_code = ngx\_array\_push\_n(lcf->codes, sizeof(ngx\_http\_script\_if\_code\_t));

if (if\_code == NULL) {

return NGX\_CONF\_ERROR;

}

if\_code->code = ngx\_http\_script\_if\_code;

// 备份ngx\_http\_rewrite\_loc\_conf\_t->codes, 在下边解析的过程中有可能会由于预申请的数组空间不足，重新申请数组空间的可能

elts = lcf->codes->elts;

/\* the inner directives must be compiled to the same code array \*/

// 这里将要解析if blcok {…}中的内容，将会遇到set指令的解析部分，代码分析下边会贴出来。

nlcf = ctx->loc\_conf[ngx\_http\_rewrite\_module.ctx\_index];

nlcf->codes = lcf->codes;

save = \*cf;

cf->ctx = ctx;

if (pclcf->name.len == 0) { // 如果if block {...}在server层，则上层name=NULL

if\_code->loc\_conf = NULL;

cf->cmd\_type = NGX\_HTTP\_SIF\_CONF; // 在server {...} block 内解析

} else { // 在location {...} block 内解析

if\_code->loc\_conf = ctx->loc\_conf; // 设置ngx\_http\_script\_if\_code\_t->loc\_conf 为当前上层的loc\_conf

cf->cmd\_type = NGX\_HTTP\_LIF\_CONF;

}

rv = ngx\_conf\_parse(cf, NULL);

\*cf = save;

if (rv != NGX\_CONF\_OK) {

return rv;

}

/\*

计算结构体ngx\_http\_script\_if\_code\_t在lcf->codes中的地址到当前if block {...}结尾的偏移量

如果在解析block{...}过程中数组大小不足被重新申请，将会重新计算if\_code在codes中的地址

\*/

if (elts != lcf->codes->elts) {

if\_code = (ngx\_http\_script\_if\_code\_t \*)

((u\_char \*) if\_code + ((u\_char \*) lcf->codes->elts - elts));

}

if\_code->next = (u\_char \*) lcf->codes->elts + lcf->codes->nelts

- (u\_char \*) if\_code;

/\* the code array belong to parent block \*/

nlcf->codes = NULL;

return NGX\_CONF\_OK;

}

由于函数体很长，我们将对部分重点内容进行分析，其余内容将贴出源码注释。

ngx\_http\_rewrite\_if\_condition(ngx\_conf\_t \*cf, ngx\_http\_rewrite\_loc\_conf\_t \*lcf)

{

*………………………………省略了一些检查操作……………………………………..*

*………………………………省略了一些检查操作……………………………………..*

// 函数中有一个大的if分支来说明“if”指令表达式的作用范围。If表达式共两种功能，一种是两个表达式的比较或匹配，另一种是检查表达式的属性。这里我们将对第一种进行分析。

if (len > 1 && p[0] == '$') {

*………………………………省略了一些检查操作……………………………………..*

// 将变量加入到变量的索引数组中，并记录变量在数组中的index，通过这个函数的执行在ngx\_http\_rewrite\_loc\_conf\_t的codes指向的数组中有了第一个结构ngx\_http\_script\_var\_code\_t，这个结构有两个字段，一个是处理函数的指针，另一个保存的是变量在索引变量数组中的index。（value[cur]是例子中的$request\_method）

if (ngx\_http\_rewrite\_variable(cf, lcf, &value[cur]) != NGX\_CONF\_OK) {

return NGX\_CONF\_ERROR;

}

*………………………………省略了一些检查操作……………………………………..*

// 接下来是对几种运算符的分别解析，我们仅对上边的例子出现的情况进行分析

// 运算符是"="时

if (len == 1 && p[0] == '=') {

// 这里是对“if ( $request\_method = GET )”中的“GET”进行的处理，函数中向lcf->codes 数组中继续添加了对应的结构体类型(value[last]对应的是例子中的字符串”GET”)

if (ngx\_http\_rewrite\_value(cf, lcf, &value[last]) != NGX\_CONF\_OK) {

return NGX\_CONF\_ERROR;

}

// ngx\_http\_script\_start\_code()函数就是在内存池上申请空间的函数

code = ngx\_http\_script\_start\_code(cf->pool, &lcf->codes,

sizeof(uintptr\_t));

if (code == NULL) {

return NGX\_CONF\_ERROR;

}

// 与上边ngx\_http\_rewrite\_value ()函数加入到lcf->codes中不同的是，这里就是一个函数，而ngx\_http\_rewrite\_variable()函数中加入的是一个结构体，稍后会画出lcf->codes中内存布局图。

\*code = ngx\_http\_script\_equal\_code;

return NGX\_CONF\_OK;

}

typedef struct {

// codes中存放的是一个指针数组

ngx\_array\_t \*codes; /\* uintptr\_t \*/

ngx\_uint\_t stack\_size;

ngx\_flag\_t log;

ngx\_flag\_t uninitialized\_variable\_warn;

} ngx\_http\_rewrite\_loc\_conf\_t;

// 这个函数的作用是将变量存放到cmcf->variables的数组中并保存此变量在数组中的index，同时根据也就知道了去哪里获取变量的值，变量的值存放在ngx\_http\_request\_t结构体中variables数组中

static char \*

ngx\_http\_rewrite\_variable(ngx\_conf\_t \*cf, ngx\_http\_rewrite\_loc\_conf\_t \*lcf,

ngx\_str\_t \*value)

{

ngx\_int\_t index;

ngx\_http\_script\_var\_code\_t \*var\_code;

// 获取到字符串” request\_method”

value->len--;

value->data++;

// 获取request\_method变量在索引变量数组中的index。如果未在索引变量数组中，将添加到数组中并返回index.通过这个调用变量” request\_method”将在cmcf->variables数组中有了自己的位置，cmcf->variables与ngx\_http\_request\_t结构体中variables数组是一一对应的，ngx\_http\_request\_t结构体中variables中存放的是变量的值。也就是说cmcf->variables中下标是N的变量在ngx\_http\_request\_t结构体中variables中下标为N的位置必然存有它的变量值，由于变量的值与请求相关，每个请求变量的值都不一样，所以在解析”if”指令的时候都会重新获取一次变量的值，并存放在ngx\_http\_request\_t结构体中variables中对应的位置上，然后在进行字符串比较。（变量的相关内容这里仅作简单的介绍，具体处理过程此处不进行讲解）

index = ngx\_http\_get\_variable\_index(cf, value);

if (index == NGX\_ERROR) {

return NGX\_CONF\_ERROR;

}

// 在ngx\_http\_rewrite\_loc\_conf\_t的codes字段上申请一块连续的内容空间，存放ngx\_http\_script\_var\_code\_t结构体。从这里可以看出，定义的变量的脚本解析处理函数对应的函数是

ngx\_http\_script\_var\_code（）

var\_code = ngx\_http\_script\_start\_code(cf->pool, &lcf->codes,

sizeof(ngx\_http\_script\_var\_code\_t));

if (var\_code == NULL) {

return NGX\_CONF\_ERROR;

}

// 对ngx\_http\_script\_var\_code\_t结构体的两个字段进行赋值

var\_code->code = ngx\_http\_script\_var\_code;

var\_code->index = index;

return NGX\_CONF\_OK;

}

/\*

\* 用于设置set指令和if\_condition的表达式

\* 表达式可以是常量字符串和内部变量

\*/

static char \*

ngx\_http\_rewrite\_value(ngx\_conf\_t \*cf, ngx\_http\_rewrite\_loc\_conf\_t \*lcf,

ngx\_str\_t \*value)

{

ngx\_int\_t n;

ngx\_http\_script\_compile\_t sc;

ngx\_http\_script\_value\_code\_t \*val;

ngx\_http\_script\_complex\_value\_code\_t \*complex;

// 统计参数value中的"$"符号个数，也就是出现变量的个数

// e.g. set $USER\_V2 "B"; 返回0

// set $USER\_V3 $USER\_V1${USER\_V2}C; 返回2

n = ngx\_http\_script\_variables\_count(value);

// 这里将会分两种情况，第一个种value中没有变量出现，仅有常量字符串（e.g. 类似于例子中set $USER\_V2 "B";其中仅有字符串”B”）。第二种value中有变量出现，既有变量又有常量字符串（e.g. 类似于例子中set $USER\_V3 $USER\_V1${USER\_V2}C;其中变量的个数等于2）。

if (n == 0) {

// 第一种情况，我们将向lcf->codes中继续塞结构体，这回不同的是塞的结构体类型变ngx\_http\_script\_value\_code\_t啦，这样我就知道啦，常量字符串的处理对应结构体是ngx\_http\_script\_value\_code\_t，结构体中会保存处理常量字符串函数的指针

val = ngx\_http\_script\_start\_code(cf->pool, &lcf->codes,

sizeof(ngx\_http\_script\_value\_code\_t));

if (val == NULL) {

return NGX\_CONF\_ERROR;

}

// 尝试转换value到整数，不能转换成整数设置n =0

n = ngx\_atoi(value->data, value->len); 0

if (n == NGX\_ERROR) {

n = 0;

}

// 对结构体ngx\_http\_script\_value\_code\_t进行赋值，保存常量字符串的值。

val->code = ngx\_http\_script\_value\_code;

val->value = (uintptr\_t) n;

val->text\_len = (uintptr\_t) value->len;

val->text\_data = (uintptr\_t) value->data;

return NGX\_CONF\_OK;

}

// 这里是复合类型的value，就是上边说到的第二种情况，稍后会介绍这种情况，设计到的函数比较多

complex = ngx\_http\_script\_start\_code(cf->pool, &lcf->codes,

sizeof(ngx\_http\_script\_complex\_value\_code\_t));

if (complex == NULL) {

return NGX\_CONF\_ERROR;

}

complex->code = ngx\_http\_script\_complex\_value\_code;

complex->lengths = NULL;

// 这里设置lcf->codes 存放变量的value和常量字符串的值，ngx\_http\_script\_compile（）函数中设置

// 的函数会在ngx\_http\_script\_complex\_value\_code（）函数之后执行

ngx\_memzero(&sc, sizeof(ngx\_http\_script\_compile\_t));

sc.cf = cf;

sc.source = value;

sc.lengths = &complex->lengths;

sc.values = &lcf->codes;

sc.variables = n;

sc.complete\_lengths = 1;

if (ngx\_http\_script\_compile(&sc) != NGX\_OK) {

return NGX\_CONF\_ERROR;

}

return NGX\_CONF\_OK;

}

static char \*

ngx\_http\_rewrite\_set(ngx\_conf\_t \*cf, ngx\_command\_t \*cmd, void \*conf)

{

*………………………………省略了一些检查操作……………………………………..*

// 增加自定义变量到变量的hash数组中(cmcf->variables\_keys->keys)，这里的NGX\_HTTP\_VAR\_CHANGEABLE标记说明变量是可被覆盖的

v = ngx\_http\_add\_variable(cf, &value[1], NGX\_HTTP\_VAR\_CHANGEABLE);

if (v == NULL) {

return NGX\_CONF\_ERROR;

}

// 在将变量增加到索引变量数组中（cmcf->variables）

index = ngx\_http\_get\_variable\_index(cf, &value[1]);

if (index == NGX\_ERROR) {

return NGX\_CONF\_ERROR;

}

*………………………………省略一些操作……………………………………..*

// 这里先处理表达式是有原因的，当在lcf->codes表中注册处理结构时，由于set的表达式处理函数中需要使用表达式的value，所以运行lcf->codes表中的函数时，需要先将具体的值填充好

if (ngx\_http\_rewrite\_value(cf, lcf, &value[2]) != NGX\_CONF\_OK) {

return NGX\_CONF\_ERROR;

}

*………………………………省略一些操作……………………………………..*

// 增加变量的解析结构体到lcf->codes 中，

// ngx\_http\_script\_var\_code\_t->code 赋值为ngx\_http\_script\_set\_var\_code

// ngx\_http\_script\_var\_code\_t->index 赋值为变量在索引数组中的index

vcode = ngx\_http\_script\_start\_code(cf->pool, &lcf->codes,

sizeof(ngx\_http\_script\_var\_code\_t));

if (vcode == NULL) {

return NGX\_CONF\_ERROR;

}

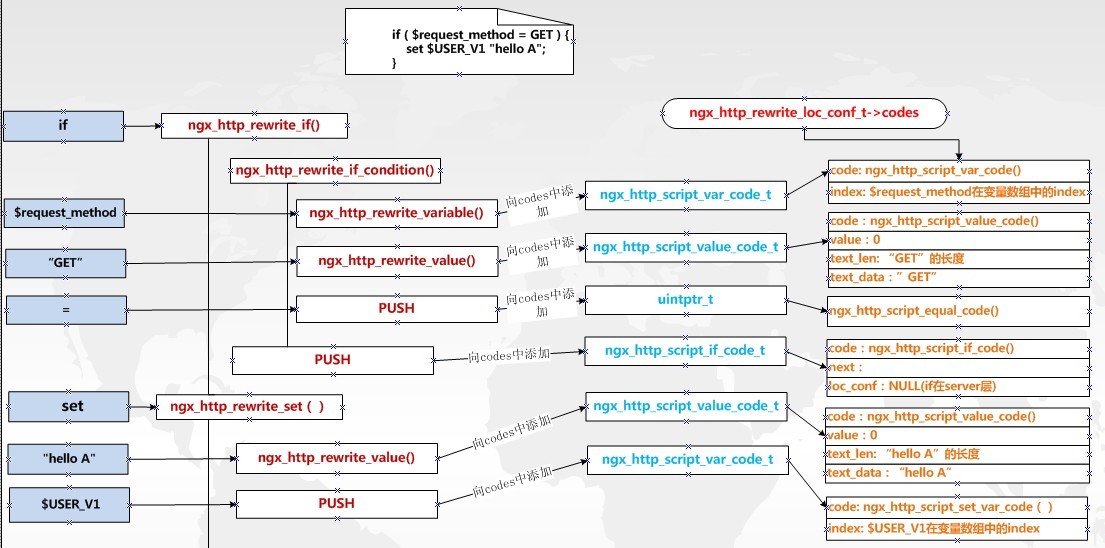
vcode->code = ngx\_http\_script\_set\_var\_code;

vcode->index = (uintptr\_t) index;

return NGX\_CONF\_OK;

}

执行完ngx\_http\_rewrite\_if（）函数，ngx\_http\_rewrite\_loc\_conf\_t的codes指针数组将指向的内存空间布局格式如下。



复合类型变量的解析

static ngx\_int\_t

ngx\_http\_script\_add\_var\_code(ngx\_http\_script\_compile\_t \*sc, ngx\_str\_t \*name)

{

ngx\_int\_t index, \*p;

ngx\_http\_script\_var\_code\_t \*code;

// 添加name参数到索引变量数组中，单独加入索引变量的数组，会在ngx\_http\_block（）-> ngx\_http\_variables\_init\_vars（）

// 中判断在hash过的变量数组中是否存在，如果存在于使用hash过的数组中，将会使用hash过的数组中的变量字段赋值给索引变量数组中的变量

index = ngx\_http\_get\_variable\_index(sc->cf, name);

if (index == NGX\_ERROR) {

return NGX\_ERROR;

}

if (sc->flushes) { // 将索引变量加入到sc->flushes 指向的数组中

p = ngx\_array\_push(\*sc->flushes);

if (p == NULL) {

return NGX\_ERROR;

}

\*p = index;

}

code = ngx\_http\_script\_add\_code(\*sc->lengths,

sizeof(ngx\_http\_script\_var\_code\_t), NULL);

if (code == NULL) {

return NGX\_ERROR;

}

code->code = (ngx\_http\_script\_code\_pt) ngx\_http\_script\_copy\_var\_len\_code;

code->index = (uintptr\_t) index;

code = ngx\_http\_script\_add\_code(\*sc->values,

sizeof(ngx\_http\_script\_var\_code\_t),

&sc->main);

if (code == NULL) {

return NGX\_ERROR;

}

code->code = ngx\_http\_script\_copy\_var\_code;

code->index = (uintptr\_t) index;

return NGX\_OK;

}