# nginx的启动阶段 (30%)

## 概述 (100%)

nginx启动阶段指从nginx初始化直至准备好按最新配置提供服务的过程。

在不考虑nginx单进程工作的情况下，这个过程包含三种方式：

1. 启动新的nginx
2. reload配置
3. 热替换nginx代码

三种方式有共同的流程，下面这幅图向我们展现了这个流程：

图11-1

流程的开端是解析nginx配置、初始化模块，接着是初始化文件句柄，初始化共享内存，然后是监听端口，再后来创建worker子进程和其他辅助子进程，最后是worker初始化事件机制。以上步骤结束以后，nginx各个子进程开始各司其职，比如worker进程开始accept请求并按最新配置处理请求，cache-manager进程开始管理cache文件目录等等。

除了这些共同流程，这三种方式的差异也非常明显。第一种方式包含命令行解析的过程，同时输出有一段时间是输出到控制台。reload配置有两种形式，一种是使用nginx命令行，一种是向master进程发送HUP信号，前者表面上与第一种方式无异，但实际上差别很大，后者则完全不支持控制台输出，无法直接查看nginx的启动情况。而且reload配置时，nginx需要自动停止以往生成的子进程，所以还包含复杂的进程管理操作，这一点在启动新的nginx的方式中是不存在的。热替换nginx代码虽然使用上与reload配置的后一种形式相似，但在解析nginx配置方面，与reload配置的方式差距非常大。另外，热替换nginx代码时，对以往创建的子进程管理也不像reload配置那样，需要手工触发进行。所以，我们想弄懂nginx的启动阶段，就必须理解所有这三种方式下nginx都是如何工作的。

## 共有流程 (100%)

从概述中我们了解到，nginx启动分为三种方式，虽然各有不同，但也有一段相同的流程。在这一节中，我们对nginx启动阶段的共用流程进行讨论。

共有流程的代码主要集中在ngx\_cycle.c、ngx\_process.c、ngx\_process\_cycle.c和ngx\_event.c这四个文件中。我们这一节只讨论nginx的框架代码，而与http相关的模块代码，我们会在后面进行分析。

共有流程开始于解析nginx配置，这个过程集中在ngx\_init\_cycle函数中。ngx\_init\_cycle是nginx的一个核心函数，共有流程中与配置相关的几个过程都在这个函数中实现，其中包括解析nginx配置、初始化CORE模块，接着是初始化文件句柄，初始化错误日志，初始化共享内存，然后是监听端口。可以说共有流程80%都是现在ngx\_init\_cycle函数中。

在具体介绍以前，我们先解决一个概念问题------什么叫cycle？

cycle就是周期的意思，对应着一次启动过程。也就是说，不论发生了上节介绍的三种启动方式的哪一种，nginx都会创建一个新的cycle与这次启动对应。

### 配置解析接口 (100%)

ngx\_init\_cycle提供的是配置解析接口。接口是一个切入点，通过少量代码提供一个完整功能的调用。配置解析接口分为两个阶段，一个是准备阶段，另一个就是真正开始调用配置解析。准备阶段指什么呢？主要是准备三点：

1. 准备内存

nginx根据以往的经验（old\_cycle）预测这一次的配置需要分配多少内存。比如，我们可以看这段：

if (old\_cycle->shared\_memory.part.nelts) {  
 n = old\_cycle->shared\_memory.part.nelts;  
 for (part = old\_cycle->shared\_memory.part.next; part; part = part->next)  
 {  
 n += part->nelts;  
 }  
  
} else {  
 n = 1;  
}  
  
if (ngx\_list\_init(&cycle->shared\_memory, pool, n, sizeof(ngx\_shm\_zone\_t))  
 != NGX\_OK)  
{  
 ngx\_destroy\_pool(pool);  
 return NULL;  
}

这段代码的意思是遍历old\_cycle，统计上一次系统中分配了多少块共享内存，接着就按这个数据初始化当前cycle中共享内存的规模。

1. 准备错误日志

nginx启动可能出错，出错就要记录到错误日志中。而错误日志本身也是配置的一部分，所以不解析完配置，nginx就不能了解错误日志的信息。nginx通过使用上一个周期的错误日志来记录解析配置时发生的错误，而在配置解析完成以后，nginx就用新的错误日志替换旧的错误日志。具体代码摘抄如下，以说明nginx解析配置时使用old\_cycle的错误日志：

log = old\_cycle->log;  
pool->log = log;  
cycle->log = log;

1. 准备数据结构

主要是两个数据结果，一个是ngx\_cycle\_t结构，一个是ngx\_conf\_t结构。前者用于存放所有CORE模块的配置，后者则是用于存放解析配置的上下文信息。具体代码如下：

for (i = 0; ngx\_modules[i]; i++) {  
 if (ngx\_modules[i]->type != NGX\_CORE\_MODULE) {  
 continue;  
 }  
  
 module = ngx\_modules[i]->ctx;  
  
 if (module->create\_conf) {  
 rv = module->create\_conf(cycle);  
 if (rv == NULL) {  
 ngx\_destroy\_pool(pool);  
 return NULL;  
 }  
 cycle->conf\_ctx[ngx\_modules[i]->index] = rv;  
 }  
}  
  
conf.ctx = cycle->conf\_ctx;  
conf.cycle = cycle;  
conf.pool = pool;  
conf.log = log;  
conf.module\_type = NGX\_CORE\_MODULE;  
conf.cmd\_type = NGX\_MAIN\_CONF;

准备好了这些内容，nginx开始调用配置解析模块，其代码如下：

if (ngx\_conf\_param(&conf) != NGX\_CONF\_OK) {  
 environ = senv;  
 ngx\_destroy\_cycle\_pools(&conf);  
 return NULL;  
}  
  
if (ngx\_conf\_parse(&conf, &cycle->conf\_file) != NGX\_CONF\_OK) {  
 environ = senv;  
 ngx\_destroy\_cycle\_pools(&conf);  
 return NULL;  
}

第一个if解析nginx命令行参数\'-g\'加入的配置。第二个if解析nginx配置文件。好的设计就体现在接口极度简化，模块之间的耦合非常低。这里只使用区区10行完成了配置的解析。在这里，我们先浅尝辄止，具体nginx如何解析配置，我们将在后面的小节做细致的介绍。

## 配置解析

### 通用过程 (100%)

配置解析模块在ngx\_conf\_file.c中实现。模块提供的接口函数主要是ngx\_conf\_parse，另外，模块提供一个单独的接口ngx\_conf\_param，用来解析命令行传递的配置，当然，这个接口也是对ngx\_conf\_parse的包装。

ngx\_conf\_parse函数支持三种不同的解析环境：

1. parse\_file：解析配置文件；
2. parse\_block：解析块配置。块配置一定是由"{"和"}"包裹起来的；
3. parse\_param：解析命令行配置。命令行配置中不支持块指令。

我们先来鸟瞰nginx解析配置的流程，整个过程可参见下面示意图：

图11-2

这是一个递归的过程。nginx首先解析core模块的配置。core模块提供一些块指令，这些指令引入其他类型的模块，nginx遇到这些指令，就重新迭代解析过程，解析其他模块的配置。这些模块配置中又有一些块指令引入新的模块类型或者指令类型，nginx就会再次迭代，解析这些新的配置类型。比如上图，nginx遇到"events"指令，就重新调用ngx\_conf\_parse()解析event模块配置，解析完以后ngx\_conf\_parse()返回，nginx继续解析core模块指令，直到遇到"http"指令。nginx再次调用ngx\_conf\_parse()解析http模块配置的http级指令，当遇到"server"指令时，nginx又一次调用ngx\_conf\_parse()解析http模块配置的server级指令。

了解了nginx解析配置的流程，我们来看其中的关键函数ngx\_conf\_parse()。

ngx\_conf\_parse()解析配置分成两个主要阶段，一个是词法分析，一个是指令解析。

词法分析通过ngx\_conf\_read\_token()函数完成。指令解析有两种方式，其一是使用nginx内建的指令解析机制，其二是使用第三方自定义指令解析机制。自定义指令解析可以参见下面的代码：

if (cf->handler) {  
 rv = (\*cf->handler)(cf, NULL, cf->handler\_conf);  
 if (rv == NGX\_CONF\_OK) {  
 continue;  
 }  
  
 if (rv == NGX\_CONF\_ERROR) {  
 goto failed;  
 }  
  
 ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0, rv);  
  
 goto failed;  
}

这里注意cf->handler和cf->handler\_conf两个属性，其中handler是自定义解析函数指针，handler\_conf是conf指针。

下面着重介绍nginx内建的指令解析机制。本机制分为4个步骤：

1. 只有处理的模块的类型是NGX\_CONF\_MODULE或者是当前正在处理的模块类型，才可能被执行。nginx中有一种模块类型是NGX\_CONF\_MODULE，当前只有ngx\_conf\_module一种，只支持一条指令"include"。"include"指令的实现我们后面再进行介绍。

ngx\_modules[i]->type != NGX\_CONF\_MODULE && ngx\_modules[i]->type != cf->module\_type

1. 匹配指令名，判断指令用法是否正确。

a) 指令的Context必须当前解析Context相符；

!(cmd->type & cf->cmd\_type)

b) 非块指令必须以";"结尾；

!(cmd->type & NGX\_CONF\_BLOCK) && last != NGX\_OK

c) 块指令必须后接"{"；

(cmd->type & NGX\_CONF\_BLOCK) && last != NGX\_CONF\_BLOCK\_START

d) 指令参数个数必须正确。注意指令参数有最大值NGX\_CONF\_MAX\_ARGS，目前值为8。

if (!(cmd->type & NGX\_CONF\_ANY)) {  
  
 if (cmd->type & NGX\_CONF\_FLAG) {  
  
 if (cf->args->nelts != 2) {  
 goto invalid;  
 }  
  
 } else if (cmd->type & NGX\_CONF\_1MORE) {  
  
 if (cf->args->nelts < 2) {  
 goto invalid;  
 }  
  
 } else if (cmd->type & NGX\_CONF\_2MORE) {  
  
 if (cf->args->nelts < 3) {  
 goto invalid;  
 }  
  
 } else if (cf->args->nelts > NGX\_CONF\_MAX\_ARGS) {  
  
 goto invalid;  
  
 } else if (!(cmd->type & argument\_number[cf->args->nelts - 1])) {  
 goto invalid;  
 }  
}

1. 取得指令工作的conf指针。

if (cmd->type & NGX\_DIRECT\_CONF) {  
 conf = ((void \*\*) cf->ctx)[ngx\_modules[i]->index];  
  
} else if (cmd->type & NGX\_MAIN\_CONF) {  
 conf = &(((void \*\*) cf->ctx)[ngx\_modules[i]->index]);  
  
} else if (cf->ctx) {  
 confp = \*(void \*\*) ((char \*) cf->ctx + cmd->conf);  
  
 if (confp) {  
 conf = confp[ngx\_modules[i]->ctx\_index];  
 }  
}

a) NGX\_DIRECT\_CONF常量单纯用来指定配置存储区的寻址方法，只用于core模块。 b) NGX\_MAIN\_CONF常量有两重含义，其一是指定指令的使用上下文是main（其实还是指core模块），其二是指定配置存储区的寻址方法。所以，在代码中常常可以见到使用上下文是main的指令的cmd->type属性定义如下：

NGX\_MAIN\_CONF|NGX\_DIRECT\_CONF|...

表示指令使用上下文是main，conf寻址方式是直接寻址。

使用NGX\_MAIN\_CONF还表示指定配置存储区的寻址方法的指令有4个："events"、"http"、"mail"、"imap"。这四个指令也有共同之处------都是使用上下文是main的块指令，并且块中的指令都使用其他类型的模块（分别是event模块、http模块、mail模块和mail模块）来处理。

NGX\_MAIN\_CONF|NGX\_CONF\_BLOCK|...

后面分析ngx\_http\_block()函数时，再具体分析为什么需要NGX\_MAIN\_CONF这种配置寻址方式。

c) 除开core模块，其他类型的模块都会使用第三种配置寻址方式，也就是根据cmd->conf的值从cf->ctx中取出对应的配置。举http模块为例，cf->conf的可选值是NGX\_HTTP\_MAIN\_CONF\_OFFSET、NGX\_HTTP\_SRV\_CONF\_OFFSET、NGX\_HTTP\_LOC\_CONF\_OFFSET，分别对应"http{}"、"server{}"、"location{}"这三个http配置级别。

1. 执行指令解析回调函数

rv = cmd->set(cf, cmd, conf);

cmd是词法分析得到的结果，conf是上一步得到的配置存贮区地址。

### http的解析

http是作为一个core模块被nginx通用解析过程解析的，其核心就是"http"块指令回调，它完成了http解析的整个功能，从初始化到计算配置结果。

因为这是本书第一次提到块指令，所以在这里对其做基本介绍。

块指令的流程是：

1. 创建并初始化上下文环境；
2. 调用通用解析流程解析；
3. 根据解析结果进行后续合并处理；
4. 善后工作。

下面我们以"http"指令为例来介绍这个流程：

#### 创建并初始化上下文环境

ctx = ngx\_pcalloc(cf->pool, sizeof(ngx\_http\_conf\_ctx\_t));  
  
\*(ngx\_http\_conf\_ctx\_t \*\*) conf = ctx;  
  
...  
  
ctx->main\_conf = ngx\_pcalloc(cf->pool,  
 sizeof(void \*) \* ngx\_http\_max\_module);  
  
ctx->srv\_conf = ngx\_pcalloc(cf->pool, sizeof(void \*) \* ngx\_http\_max\_module);  
  
ctx->loc\_conf = ngx\_pcalloc(cf->pool, sizeof(void \*) \* ngx\_http\_max\_module);  
  
for (m = 0; ngx\_modules[m]; m++) {  
 if (ngx\_modules[m]->type != NGX\_HTTP\_MODULE) {  
 continue;  
 }  
  
 module = ngx\_modules[m]->ctx;  
 mi = ngx\_modules[m]->ctx\_index;  
  
 if (module->create\_main\_conf) {  
 ctx->main\_conf[mi] = module->create\_main\_conf(cf);  
 }  
  
 if (module->create\_srv\_conf) {  
 ctx->srv\_conf[mi] = module->create\_srv\_conf(cf);  
 }  
  
 if (module->create\_loc\_conf) {  
 ctx->loc\_conf[mi] = module->create\_loc\_conf(cf);  
 }  
}  
  
pcf = \*cf;  
cf->ctx = ctx;  
  
for (m = 0; ngx\_modules[m]; m++) {  
 if (ngx\_modules[m]->type != NGX\_HTTP\_MODULE) {  
 continue;  
 }  
  
 module = ngx\_modules[m]->ctx;  
  
 if (module->preconfiguration) {  
 if (module->preconfiguration(cf) != NGX\_OK) {  
 return NGX\_CONF\_ERROR;  
 }  
 }  
}

http模块的上下文环境ctx（注意我们在通用解析流程中提到的ctx是同一个东西）非常复杂，它是由三个指针数组组成的：main\_conf、srv\_conf、loc\_conf。根据上面的代码可以看到，这三个数组的元素个数等于系统中http模块的个数。想想我们平时三四十个http模块的规模，大家也应该可以理解这一块结构的庞大。nginx还为每个模块分别执行对应的create函数分配空间。我们需要注意后面的这一句"cf->ctx = ctx;"，正是这一句将解析配置的上下文切换成刚刚建立的ctx。最后一段代码通过调用各个http模块的preconfiguration回调函数完成了对应模块的预处理操作，其主要工作是创建模块用到的变量。

#### 调用通用解析流程解析

cf->module\_type = NGX\_HTTP\_MODULE;  
cf->cmd\_type = NGX\_HTTP\_MAIN\_CONF;  
rv = ngx\_conf\_parse(cf, NULL);

基本上所有的块指令都类似上面的三行语句（例外是map，它用的是cf->handler），改变通用解析流程的工作状态，然后调用通用解析流程。

#### 根据解析结果进行后续合并处理

for (m = 0; ngx\_modules[m]; m++) {  
 if (module->init\_main\_conf) {  
 rv = module->init\_main\_conf(cf, ctx->main\_conf[mi]);  
 }  
  
 rv = ngx\_http\_merge\_servers(cf, cmcf, module, mi);  
}  
  
for (s = 0; s < cmcf->servers.nelts; s++) {  
  
 if (ngx\_http\_init\_locations(cf, cscfp[s], clcf) != NGX\_OK) {  
 return NGX\_CONF\_ERROR;  
 }  
  
 if (ngx\_http\_init\_static\_location\_trees(cf, clcf) != NGX\_OK) {  
 return NGX\_CONF\_ERROR;  
 }  
}  
  
if (ngx\_http\_init\_phases(cf, cmcf) != NGX\_OK) {  
 return NGX\_CONF\_ERROR;  
}  
  
if (ngx\_http\_init\_headers\_in\_hash(cf, cmcf) != NGX\_OK) {  
 return NGX\_CONF\_ERROR;  
}  
  
for (m = 0; ngx\_modules[m]; m++) {  
 if (module->postconfiguration) {  
 if (module->postconfiguration(cf) != NGX\_OK) {  
 return NGX\_CONF\_ERROR;  
 }  
 }  
}  
  
if (ngx\_http\_variables\_init\_vars(cf) != NGX\_OK) {  
 return NGX\_CONF\_ERROR;  
}  
  
if (ngx\_http\_init\_phase\_handlers(cf, cmcf) != NGX\_OK) {  
 return NGX\_CONF\_ERROR;  
}  
  
if (ngx\_http\_optimize\_servers(cf, cmcf, cmcf->ports) != NGX\_OK) {  
 return NGX\_CONF\_ERROR;  
}

以上是http配置处理最重要的步骤。首先，在这里调用了各个模块的postconfiguration回调函数完成了模块配置过程。更重要的是，它为nginx建立了一棵完整的配置树（叶子节点为location，包含location的完整配置）、完整的location搜索树、一张变量表、一张完成的阶段处理回调表(phase handler)、一张server对照表和一张端口监听表。下面我们将分别介绍这些配置表的生成过程。

##### location配置树

介绍这部分以前，先说明一个nginx的公理

公理11-1：所有存放参数为NGX\_HTTP\_SRV\_CONF\_OFFSET的配置，配置仅在请求匹配的虚拟主机(server)上下文中生效，而所有存放参数为NGX\_HTTP\_LOC\_CONF\_OFFSET的配置，配置仅在请求匹配的路径(location)上下文中生效。

正因为有公理11-1，所以nginx需要调用merge\_XXX回调函数合并配置。具体的原因是很多配置指令可以放在不同配置层级，比如access\_log既可以在http块中配置，又可以在server块中配置，还可以在location块中配置。 但是因为公理11-1，access\_log指令配置只有在路径(location)上下文中生效，所以需要将在http块中配置的access\_log指令的配置向路径上下文做两次传递，第一次从HTTP(http)上下文到虚拟主机(server)上下文，第二次从虚拟主机上下文到路径上下文。

可能有人会疑惑，为什么需要传递和合并呢？难道它们不在一张表里么？对，在创建并初始化上下文环境的过程中，大家已经看到，nginx为HTTP上下文创建了main\_conf，为虚拟主机上下文创建了srv\_conf，为路径上下文创建了loc\_conf。但是，这张表只是用于解析在http块但不包含server块中定义的指令。而后面我们会看到，在server块指令中，同样建立了srv\_conf和loc\_conf，用于解析在server块但不含location块中定义的指令。所以nginx其实维护了很多张配置表，因此nginx必须将配置在这些表中从顶至下不断传递。

前面列出的

for (m = 0; ngx\_modules[m]; m++) {  
 if (module->init\_main\_conf) {  
 rv = module->init\_main\_conf(cf, ctx->main\_conf[mi]);  
 }  
  
 rv = ngx\_http\_merge\_servers(cf, cmcf, module, mi);  
}

就是初始化HTTP上下文，并且完成两步配置合并操作：从HTTP上下文合并到虚拟主机上下文，以及从虚拟主机上下文合并到路径上下文。其中，合并到路径上下问的操作是在ngx\_http\_merge\_servers函数中进行的，见

if (module->merge\_loc\_conf) {  
  
 /\* merge the server{}'s loc\_conf \*/  
  
 /\* merge the locations{}' loc\_conf's \*/  
  
}

大家注意观察ngx\_http\_merge\_servers函数中的这段，先将HTTP上下文中的location配置合并到虚拟主机上下文，再将虚拟主机上下文中的location配置合并到路径上下文。

##### location搜索树

公理11-2：nginx搜索路径时，正则匹配路径和其他的路径分开搜。

公理11-3：nginx路径可以嵌套。

所以，nginx存放location的有两个指针，分别是

struct ngx\_http\_core\_loc\_conf\_s {  
  
 ...  
  
 ngx\_http\_location\_tree\_node\_t \*static\_locations;  
#if (NGX\_PCRE)  
 ngx\_http\_core\_loc\_conf\_t \*\*regex\_locations;  
#endif  
  
 ...  
}

通过这段代码，大家还可以发现一点------nginx的正则表达式需要PCRE支持。

正则表达式的路径是个指针数组，指针类型就是ngx\_http\_core\_loc\_conf\_t，所以数据结构决定算法，正则表达式路径的添加非常简单，就是在表中插入一项，这里不做介绍。

而其他路径，保存在ngx\_http\_location\_tree\_node\_t指针指向的搜索树static\_locations，则是变态复杂，可以看得各位大汗淋漓。

为了说明这棵树的构建，我们先了解其他路径包含哪些：

1. 普通前端匹配的路径，例如location / {}
2. 抢占式前缀匹配的路径，例如location \^~ / {}
3. 精确匹配的路径，例如location = / {}
4. 命名路径，比如location \@a {}
5. 无名路径，比如if {}或者limit\_except {}生成的路径

我们再来看ngx\_http\_core\_loc\_conf\_t中如何体现这些路径：

普通前端匹配的路径 无

抢占式前缀匹配的路径 noregex = 1

精确匹配的路径 exact\_match = 1

命名路径 named = 1

无名路径 noname = 1

正则路径 regex != NULL

有了这些基础知识，可以看代码了。首先是ngx\_http\_init\_locations函数

ngx\_queue\_sort(locations, ngx\_http\_cmp\_locations);  
  
for (q = ngx\_queue\_head(locations);  
 q != ngx\_queue\_sentinel(locations);  
 q = ngx\_queue\_next(q))  
{  
 clcf = lq->exact ? lq->exact : lq->inclusive;  
  
 if (ngx\_http\_init\_locations(cf, NULL, clcf) != NGX\_OK) {  
 return NGX\_ERROR;  
 }  
  
 if (clcf->regex) {  
 r++;  
  
 if (regex == NULL) {  
 regex = q;  
 }  
  
 continue;  
 }  
  
 if (clcf->named) {  
 n++;  
  
 if (named == NULL) {  
 named = q;  
 }  
  
 continue;  
 }  
  
 if (clcf->noname) {  
 break;  
 }  
}  
  
if (q != ngx\_queue\_sentinel(locations)) {  
 ngx\_queue\_split(locations, q, &tail);  
}  
  
if (named) {  
 ...  
 cscf->named\_locations = clcfp;  
 ...  
}  
  
if (regex) {  
 ...  
 pclcf->regex\_locations = clcfp;  
 ...  
}

大家可以看到，这个函数正是根据不同的路径类型将locations分成多段，并以不同的指针引用。首先注意开始的排序，根据ngx\_http\_cmp\_locations比较各个location，排序以后的顺序依次是

1. 精确匹配的路径和两类前缀匹配的路径(字母序，如果某个精确匹配的路径的名字和前缀匹配的路径相同，精确匹配的路径排在前面)
2. 正则路径(出现序)
3. 命名路径(字母序)
4. 无名路径(出现序)

这样nginx可以简单的截断列表得到不同类型的路径，nginx也正是这样处理的。

另外还要注意一点，就是ngx\_http\_init\_locations的迭代调用，这里的clcf引用了两个我们没有介绍过的字段exact和inclusive。这两个字段最初是在ngx\_http\_add\_location函数（添加location配置时必然调用）中设置的：

if (clcf->exact\_match

#if (NGX\_PCRE)

: || clcf->regex

#endif

: || clcf->named || clcf->noname)

{

: lq->exact = clcf; lq->inclusive = NULL;

} else {

: lq->exact = NULL; lq->inclusive = clcf;

}

当然这部分的具体逻辑我们在介绍location解析是再具体说明。

接着我们看ngx\_http\_init\_static\_location\_trees函数。通过刚才的ngx\_http\_init\_locations函数，留在locations数组里面的还有哪些类型的路径呢？

还有普通前端匹配的路径、抢占式前缀匹配的路径和精确匹配的路径这三类。

if (ngx\_http\_join\_exact\_locations(cf, locations) != NGX\_OK) {  
 return NGX\_ERROR;  
}  
  
ngx\_http\_create\_locations\_list(locations, ngx\_queue\_head(locations));  
  
pclcf->static\_locations = ngx\_http\_create\_locations\_tree(cf, locations, 0);  
if (pclcf->static\_locations == NULL) {  
 return NGX\_ERROR;  
}

请注意除开这段核心代码，这个函数也有一个自迭代过程。

ngx\_http\_join\_exact\_locations函数是将名字相同的精确匹配的路径和两类前缀匹配的路径合并，合并方法

lq->inclusive = lx->inclusive;  
  
ngx\_queue\_remove(x);

简言之，就是将前缀匹配的路径放入精确匹配的路径的inclusive指针中，然后从列表删除前缀匹配的路径。

ngx\_http\_create\_locations\_list函数将和某个路径名拥有相同名称前缀的路径添加到此路径节点的list指针域下，并将这些路径从locations中摘除。其核心代码是

ngx\_queue\_split(&lq->list, x, &tail);  
ngx\_queue\_add(locations, &tail);  
  
ngx\_http\_create\_locations\_list(&lq->list, ngx\_queue\_head(&lq->list));  
  
ngx\_http\_create\_locations\_list(locations, x);

ngx\_http\_create\_locations\_tree函数则将刚才划分的各个list继续细分，形成一个二分搜索树，每个中间节点代表一个location，每个location有如下字段：

1. exact：两类前缀匹配路径的inclusive指针域指向这两类路径的配置上下文；
2. inclusive：精确匹配路径的exact指针域指向这些路径的配置上下文；
3. auto\_redirect：为各种upstream模块，比如proxy、fastcgi等等开启自动URI填充的功能；
4. len：路径前缀的长度。任何相同前缀的路径的len等于该路径名长度减去公共前缀的长度。比如路径/a和/ab，前者的len为2，后者的len也为1；
5. name：路径前缀，任何相同前缀的路径的name是其已于公共前缀的部分。仍举路径/a和/ab为例，前者的name为/a，后者的name为b；
6. left：左子树，当然是长度短或者字母序小的不同前缀的路径；
7. right：右子树，当然是长度长或者字母序大的不同前缀的路径。

通过上面三个步骤，nginx就将locations列表中各种类型的路径分类处理并由不同的指针引用。对于前缀路径和精确匹配的路径，形成一棵独特的二分前缀树。

##### 变量表

变量表的处理相对简单，即对照变量名表，为变量表中的每一个元素设置对应的get\_handler和data字段。在前面的章节大家已经知道，变量表variables用以处理索引变量，而变量名表variables\_keys用于处理可按变量名查找的变量。对于通过ngx\_http\_get\_variable\_index函数创建的索引变量，在变量表variables中的get\_handler初始为空，如果没有认为设置的话，将会在这里进行初始化。

特殊变量的get\_handler初始化也在这里进行：

变量前缀 get\_handler 标志

http ngx\_http\_variable\_unknown\_header\_in

sent\_http ngx\_http\_variable\_unknown\_header\_out

upstream\_http ngx\_http\_upstream\_header\_variable NGX\_HTTP\_VAR\_NOCACHEABLE

cookie ngx\_http\_variable\_cookie

arg ngx\_http\_variable\_argument NGX\_HTTP\_VAR\_NOCACHEABLE

##### 阶段处理回调表

按照下表顺序将各个模块设置的phase handler依次加入cmcf->phase\_engine.handlers列表，各个phase的phase handler的checker不同。checker主要用于限定某个phase的框架逻辑，包括处理返回值。

+-------------------------------+---------------------------------+-------------------+ +处理阶段PHASE 可自定义handler | +-------------------------------+---------------------------------+-------------------+ +NGX\_HTTP\_POST\_READ\_PHASE 是 | +-------------------------------+---------------------------------+-------------------+ +NGX\_HTTP\_SERVER\_REWRITE\_PHASE 是 | +-------------------------------+---------------------------------+-------------------+ +NGX\_HTTP\_FIND\_CONFIG\_PHASE 否 | +-------------------------------+---------------------------------+-------------------+ +NGX\_HTTP\_REWRITE\_PHASE 是 | +-------------------------------+---------------------------------+-------------------+ +NGX\_HTTP\_POST\_REWRITE\_PHASE 否 | +-------------------------------+---------------------------------+-------------------+ +NGX\_HTTP\_PREACCESS\_PHASE 是 | +-------------------------------+---------------------------------+-------------------+ +NGX\_HTTP\_ACCESS\_PHASE 是 | +-------------------------------+---------------------------------+-------------------+ +NGX\_HTTP\_POST\_ACCESS\_PHASE 否 | +-------------------------------+---------------------------------+-------------------+ +NGX\_HTTP\_TRY\_FILES\_PHASE 否 | +-------------------------------+---------------------------------+-------------------+ +NGX\_HTTP\_CONTENT\_PHASE 是 | +-------------------------------+---------------------------------+-------------------+

注意相同PHASE的phase handler是按模块顺序的反序加入回调表的。另外在NGX\_HTTP\_POST\_REWRITE\_PHASE中，ph->next指向NGX\_HTTP\_FIND\_CONFIG\_PHASE第一个phase handler，以实现rewrite last逻辑。

##### server对照表

大家如果读过nginx的"Server names"这篇官方文档，会了解nginx对于server name的处理分为4中情况：精确匹配、前缀通配符匹配、后缀通配符匹配和正则匹配。那么，下面是又一个公理，

公理11-4：nginx对于不同类型的server name分别处理。

所以，所谓server对照表，其实是四张表，分别对应四种类型的server。数据结构决定算法，四张表决定了nginx必须建立这四张表的行为。鉴于前三种类型和正则匹配可以分成两大类，nginx使用两套策略生成server对照表------对正则匹配的虚拟主机名，nginx为其建立一个数组，按照主机名在配置文件的出现顺序依次写入数组；而对于其他虚拟主机名，nginx根据它们的类型为它们分别存放在三张hash表中。三张hash表的结构完全相同，但对于前缀通配或者后缀通配这两种类型的主机名，nginx对通配符进行的预处理不同。其中".taobao.com"这种特殊的前缀通配与普通的前缀通配处理又有不同。我们现在来介绍这些不同。

处理前缀通配是将字符串按节翻转，然后去掉通配符。举个例子，"\*.example.com"会被转换成"com.example.\0"，而特殊的前缀通配".example.com"会被转换成"com.example\0"。

处理后缀通配更简单，直接去掉通配符。也举个例子，"www.example.\*"会被转换成"www.example\0"。

##### 端口监听表

对于所有写在server配置中的listen指令，nginx开始会建立一张server和端口的对照索引表。虽然这不是本节的要点，但要说明索引表到监听表的转换过程，还是需要描述其结构。如图11-3所示，这张索引表是二级索引，第一级索引以listen指定的端口为键，第二级索引以listen指定的地址为键，索引的对象就是server上下文数据结构。而端口监视表是两张表，其结构如图11-4所示。 索引表和监听表在结构上非常类似，但是却有一个非常明显的不同。索引表中第一张表的各表项的端口是唯一的，而监听表的第一张表中的不同表项的端口却可能是相同的。之所以出现这样的差别，是因为nginx会为监听表第一张表中的每一项分别建立监听套接字，而在索引表中，如果配置显式定义了需要监听不同IP地址的相同端口，它在索引表中会放在同一个端口的二级索引中，而在监听表中必须存放为两个端口相同的不同监听表项。

说明了两张表的结构，现在可以介绍转换过程：

第一步，在ngx\_http\_optimize\_servers()函数中，对索引表一级索引中的所有port下辖的二级索引分别进行排序。排序的规则是

1. 含wildcard属性的二级索引最终会尽可能排到尾部。这些二级索引类似于

listen \*:80;  
listen 80;

1. 含bind属性的二级索引最终会尽可能排到首部。这些二级索引是由那些设置了\"bind\"、\"backlog\"、\"rcvbuf\"、\"sndbuf\"、\"accept\_filter\"、\"deferred\"、\"ipv6only\"和\"so\_keepalive\"参数的listen指令生成的。
2. 其他二级索引，其相对顺序不变，排在含bind属性的二级索引之后，而在含wildcard属性的二级索引之前。

第二步，将索引表转换为监听表，这是在ngx\_http\_init\_listening()函数中实现的。其步骤是

1. 得到是否有二级索引含有wildcard属性，只需要看看排序后的二级索引的最后一项就可以了。
2. 顺次将所有含有bind属性的二级索引以一对一的方式生成监听表的表项（第一级和第二级都只有一项）。
3. 如果第一步检测到不含wildcard属性，则顺次将后续所有二级索引以一对一的方式生成监听表的表项。
4. 如果第一步检测到含wildcard属性，则以含wildcard属性的二级索引创建监听表的一级表项，并将二级索引中从第一不含bind属性的表项开始的所有表项一同转换成为刚刚创建的监听表一级表项的下级表项。

#### 善后工作

善后工作基本的就是一件事，还原解析上下文。"http"指令是这个进行的

\*cf = pcf;

### server的管理

前面介绍的http处理逻辑在处理"server {}"时仍然适用。server相对较为特殊的是两个指令，一个是\"server\_name\"，一个是\"listen\"。

就在上一节，我们已经介绍了\"server\_name\"

### location的管理

## 模块初始化

## 热代码部署

### reload过程解析

### upgrade过程解析