nginx 源码分析

nginx 源码分析（1）- 缘起

nginx 是一个开源的高性能 web 服务器系统，事件驱动的请求处理方式和极其苛刻的资源使用方式，使得 nginx 成为名副其实的高性能服务器。 nginx 的源码质量也相当高，作者“家酿”了许多代码，自造了不少轮子，诸如内存池、缓冲区、字符串、链表、红黑树等经典数据结构，事件驱动模型，http 解析，各种子处理模块，甚至是自动编译脚本都是作者根据自己的理解写出来的，也正因为这样，才使得 nginx 比其他的 web 服务器更加高效。

nginx 的代码相当精巧和紧凑，虽然全部代码仅有 10 万行，但功能毫不逊色于几十万行的 apache。不过各个部分之间耦合的比较厉害，很难把其中某个部分的实现拆出来使用。对于这样一个中大型的复杂系统源码进行分析，是有一定的难度的，刚开始也很难找到下手的入口，所以做这样的事情就必须首先明确目标和计划。

最初决定做这件事情是为了给自己一些挑战，让生活更有意思。但看了几天之后，觉得这件事情不该这么简单看待，这里面有太多吸引人的东西了，值得有计划的系统学习和分析。首先这个系统中几乎涵盖了实现高性能服务器的各种必杀技，epoll、kqueue、master－workers、pool、buffer… …，也涵盖了很多 web 服务开发方面的技术，ssi、ssl、proxy、gzip、regex、load balancing、reconfiguration、hot code swapping… …，还有一些常用的精巧的数据结构实现，所有的东西很主流；其次是一流的代码组织结构和干净简洁的代码风格，尤其是整个系统的命名恰到好处，可读性相当高，很 kiss，这种风格值得学习和模仿；第三是通过阅读源码可以感受到作者严谨的作风和卓越的能力，可以给自己增加动力，树立榜样的力量。

另一方面，要达到这些目标难度很高，必须要制定详细的计划和采取一定有效的方法。

对于这么大的一个系统，想一口气知晓全部的细节是不可能的，并且 nginx 各个部分的实现之间关系紧密， 不可能做到窥一斑而知全身，合适的做

法似乎应该是从 main 开始，先了解 nginx 的启动过程的顺序，然后进行问题分解，再逐个重点分析每一个重要的部分。

对每个理解的关键部分进行详细的记录和整理也是很重要的，这也是这个源码分析日志系列所要完成的任务。

为了更深刻的理解代码实现的关键，修改代码和写一些测试用例是不可避免的，这就需要搭建一个方便调试的环境，这也比较容易，因为使用的 linux 系统本身就是一个天然的开发调试环境。

个人的能力是有限的，幸运的是互联网上还有一帮同好也在孜孜不倦的做着同样的事情，与他们的交流会帮助少走一些弯路，也会互相促进，更深入和准确的理解源码的本实。

开始一次愉快的旅行，go！

nginx 源码分析（**2**）**-** 概览

源码分析是一个逐步取精的过程，最开始是一个大概了解的过程，各种认识不会太深刻，但是把这些真实的感受也记录下来，觉得挺有意思的，可能有些认识是片面或者是不正确的，但可以通过后面更深入细致的分析过程，不断的纠正错误和深化理解。源码分析是一个过程，经验是逐步累积起来的，希望文字可以把这种累积的感觉也准确记录下来。

现在就看看对 nginx 源码的第一印象吧。

源码包解压之后，根目录下有几个子目录和几个文件，最重要的子目录是 auto 和 src，最重要的文件是 configure 脚本，不同于绝大多数的开源代码，nginx 的 configure 脚本是作者手工编写的，没有使用 autoconf 之类的工具去自动生成，configure 脚本会引用 auto 目录下面的脚本文件来干活。根据不同的用途，auto 目录下面的脚本各司其职，有检查编译器版本的，有检查操作系统版本的，有检查标准库版本的，有检查模块依赖情况的，有关于安装的，有关于初始化的，有关于多线程检查的等等。configure 作为一个总驱动，调用这些脚本去生成版本信息头文件、默认被包含的模块的声明代码和 Makefile 文件，版本信息头文件 (ngx_auto_config.h,ngx_auto_headers.h)和默认被包含的模块的声明代码(ngx_modules.c)被放置在新创建的 objs 目录下。要注意的是，这几个生成的文件和 src 下面的源代码一样重要，对于理解源码是不可忽略的重要部分。

src 是源码存放的目录，configure 创建的 objs/src 目录是用来存放生成的.o 文件的，注意区分一下。

src 按照功能特性划分为几个部分，对应着是几个不同的子目录。

src/core 存放着主干部分、基础数据结构和基础设施的源码，main 函数在 src/core/nginx.c 中，这是分析源码的一个很好的起点。

src/event 存放着事件驱动模型和相关模块的源码。

src/http 存放着 http server 和相关模块的源码。

src/mail 存放着邮件代理和相关模块的源码。

src/misc 存放着 C + + 兼容性测试和 google perftools 模块的源码。

src/os 存放着依赖于操作系统实现的源码，nginx 启动过程中最重要的 master 和 workers 创建代码就在这个目录下，多少让人觉得有点意外。

nginx 的实现中有非常多的结构体，一般命名为 ngx_XXX_t，这些结构体分散在许多头文件中，而在 src/core/ngx_core.h 中把几乎所有的头文件都集合起来，所有的实现文件都会包含这个 ngx_core.h 头文件，说 nginx 的各部分源码耦合厉害就是这个原因，但实际上 nginx 各个部分之间逻辑上是划分的很清晰的，整体上是一种松散的结构。nginx 实现了一些精巧的基础数据结构，例如 ngx_string_t,ngx_list_t,ngx_array_t,ngx_pool_t,ngx_buf_t,ngx_queue_t,ngx_rbtree_t,ngx_radix_tree_t 等等，还有一些重要的基础设施，比如 log,configure file,time 等等，这些数据结构和基础设施频繁的被使用在许多地方，这会让人感觉 nginx 逻辑上的联系比较紧密，但熟悉了这些基础数据结构的实现代码就会感觉到这些数据结构都是清晰分明的，并没有真正的耦合在一起，只是有些多而已，不过 nginx 中“家酿”的代码也正是它的一个很明显的亮点。

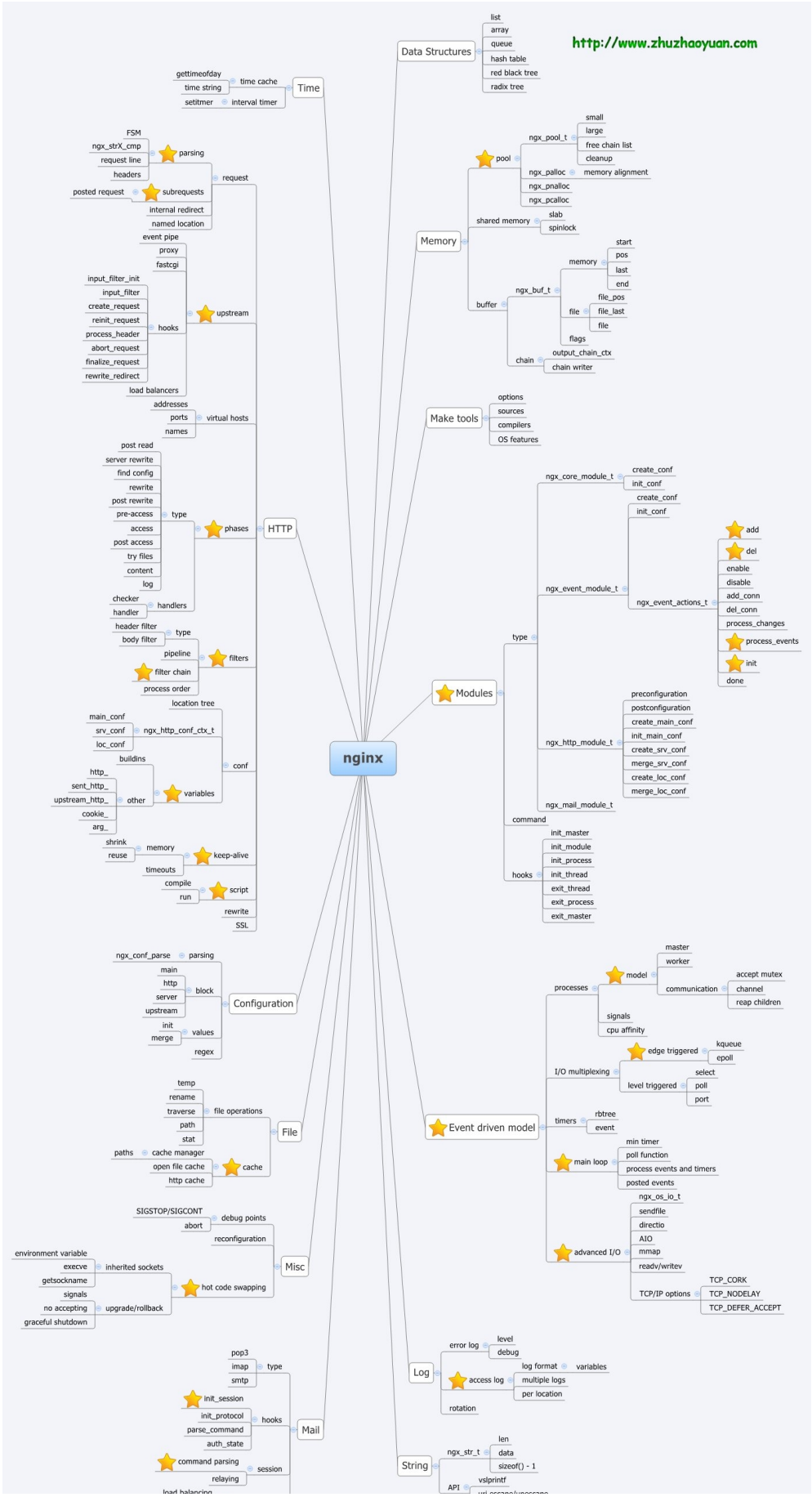
nginx 是高度模块化的，可以根据自己的需要定制模块，也可以自己根据一定的标准开发需要的模块，已经定制的模块会在 objs/ngx_modules.c 中声明，这个文件是由 configure 生成的。

nginx 启动过程中，很重要的一步就是加载和初始化模块，这是在 ngx_init_cycle 中完成的，ngx_init_cycle 会调用模块的 hook 接口（init_module）对模块初始化，ngx_init_cycle 还会调用 ngx_open_listening_sockets 初始化 socket，如果是多进程方式启动，就会调用 ngx_master_process_cycle 完成最后的启动动作，ngx_master_process_cycle 调用 ngx_start_worker_processes 生成多个工作子进程，ngx_start_worker_processes 调用 ngx_worker_process_cycle 创建工作内容，如果进程有多个子线程，这里也会初始化线程和创建线程工作内容，初始化完成之后，

ngx_worker_process_cycle 会进入处理循环，调用 ngx_process_events_and_timers，该函数调用 ngx_process_events 监听事件，并把事件投递到事件队列 ngx_posted_events 中，最终会在 ngx_event_thread_process_posted 中处理事件。

事件机制是 nginx 中很关键的一个部分，linux 下使用了 epool，freebsd 下使用了 kqueue 管理事件。

最后附上 Joshua 友情提供的源码大图一张，感谢：）


nginx
http://www.zhuzhaoyuan.com
Data Structures
list
array
queue
hash table
red black tree
radix tree
Time
gettimeofday
time string
setitmer
time cache
interval timer
HTTP
request
FSM
ngx_strX_cmp
request line
headers
parsing
posted request
subrequests
internal redirect
named location
upstream
event pipe
proxy
fastcgi
input_filter_init
input_filter
create_request
reinit_request
process_header
abort_request
finalize_request
rewrite_redirect
hooks
load balancers
virtual hosts
addresses
ports
names
phases
post read
server rewrite
find config
rewrite
post rewrite
pre-access
access
post access
try files
content
log
type
handlers
checker
handler
filters
header filter
body filter
type
pipeline
filter chain
process order
conf
location tree
ngx_http_conf_ctx_t
main_conf
srv_conf
loc_conf
variables
buildins
other
http_
sent_http_
upstream_http_
cookie_
arg_
keep-alive
shrink
reuse
memory
timeouts
script
compile
run
rewrite
SSL
Configuration
ngx_conf_parse
parsing
block
main
http
server
upstream
values
init
merge
regex
File
file operations
temp
rename
traverse
path
stat
cache
paths
cache manager
open file cache
http cache
Misc
debug points
SIGSTOP/SIGCONT
abort
reconfiguration
hot code swapping
inherited sockets
environment variable
execve
getsockname
signals
upgrade/rollback
no accepting
graceful shutdown
Mail
type
pop3
imap
smtp
hooks
init_session
init_protocol
parse_command
auth_state
session
command parsing
relaying
load balancing
Memory
pool
ngx_pool_t
small
large
free chain list
cleanup
ngx_palloc
memory alignment
ngx_pnalloc
ngx_pcalloc
shared memory
slab
spinlock
buffer
ngx_buf_t
memory
start
pos
last
end
file
file_pos
file_last
file
flags
chain
output_chain_ctx
chain writer
Make tools
options
sources
compilers
OS features
Modules
type
ngx_core_module_t
create_conf
init_conf
ngx_event_module_t
create_conf
init_conf
ngx_event_actions_t
add
del
enable
disable
add_conn
del_conn
process_changes
process_events
init
done
ngx_http_module_t
preconfiguration
postconfiguration
create_main_conf
init_main_conf
create_srv_conf
merge_srv_conf
create_loc_conf
merge_loc_conf
ngx_mail_module_t
command
hooks
init_master
init_module
init_process
init_thread
exit_thread
exit_process
exit_master
Event driven model
processes
model
master
worker
communication
accept mutex
channel
reap children
signals
cpu affinity
I/O multiplexing
edge triggered
kqueue
epoll
level triggered
select
poll
port
timers
rbtree
event
main loop
min timer
poll function
process events and timers
posted events
advanced I/O
ngx_os_io_t
sendfile
directio
AIO
mmap
readv/writev
TCP/IP options
TCP_CORK
TCP_NODELAY
TCP_DEFER_ACCEPT
Log
error log
level
debug
access log
log format
variables
multiple logs
per location
rotation
String
ngx_str_t
len
data
sizeof() - 1
API
vslprintf

nginx 源码分析（**3**）**-** 自动脚本

nginx 的自动脚本指的是 configure 脚本程序和 auto 子目录下面的脚本程序。自动脚本完成两件事情，其一是检查环境，其二是生成文件。生成的文件有两类，一类是编译代码需要的 Makefile 文件，一类是根据环境检查结果生成的 c 代码。生成的 Makefile 很干净，也很容易阅读。生成的 c 代码有三个文件，ngx_auto_config.h 是根据环境检查的结果声明的一些宏定义，这个头文件被 include 进 ngx_core.h 中，所以会被所有的源码引用到，这确保了源码是可移植的；ngx_auto_headers.h 中也是一些宏定义，不过是关于系统头文件存在性的声明；ngx_modules.c 是默认被包含进系统中的模块的声明，如果想去掉一些模块，只要修改这个文件即可。

configure 是自动脚本的总驱动，它通过组合 auto 目录下不同功能的脚本程序完成环境检查和生成文件的任务。环境检查主要是三个部分：编译器版本及支持特性、操作系统版本及支持特性、第三方库支持，检查的脚本程序分别存放在 auto/cc、auto/os、auto/lib 三个子目录中。检查的方法很有趣，通过自动编译用于检查某个特性的代码片段，根据编译器的输出情况判定是否支持该种特性。根据检查的情况，如果环境足以支持运行一个简单版本的 nginx，就会生成 Makefile 和 c 代码，这些文件会存放在新创建的 objs 目录下。当然，也可能会失败，假如系统不支持 pcre 和 ssh，如果没有屏蔽掉相关的模块，自动脚本就会失败。

auto 目录下的脚本职能划分非常清晰，有检查环境的，有检查模块的，有提供帮助信息的(./configure –help)，有处理脚本参数的，也有一些脚本纯粹是为了模块化自动脚本而设计出来的，比如 feature 脚本是用于检查单一特性的，其他的环境检查脚本都会调用这个脚本去检查某个特性。还有一些脚本是用来输出信息到生成文件的，比如 have、nohave、make、install 等。

之所以要在源码分析中专门谈及自动脚本，是因为 nginx 的自动脚本不是用 autoconf 之类的工具生成的，而是作者手工编写的，并且包含一定的设计成分，对于需要编写自动脚本的人来说，有很高的参考价值。这里也仅仅是粗略

的介绍一下，需要详细了解最好是读一下这些脚本，这些脚本并没有使用多少生僻的语法，可读性是不错的。

btw，后面开始进入真正的源码分析阶段，nginx 的源码中有非常多的结构体，这些结构体之间引用也很频繁，很难用文字表述清楚之间的关系，觉得用图表是最好的方式，因此需要掌握一种高效灵活的作图方法，我选择的是 graphviz，这是 at&t 贡献的跨平台的图形生成工具，通过写一种称为“the dot language”的脚本语言，然后用 dot 命令就可以直接生成指定格式的图，很方便。

nginx 源码分析（**4**）**-**方法（**1**）

看了几天的源码，进度很慢，过于关注代码实现的细节了，反而很难看清整体结构。于是问诸 google 寻找方法。大体上分析源代码都要经历三遍过程，第一遍是浏览，通过阅读源码的文档和注释，阅读接口，先弄清楚每个模块是干什么的而不关心它是怎么做的，画出架构草图；第二遍是精读，根据架构草图把系统分为小部分，每个部分从源码实现自底向上的阅读，更深入细致的理解每个模块的实现方式以及与模块外部的接口方式等，弄明白模块是怎么做的，为什么这样做，有没有更好的方式，自己会如何实现等等问题；第三遍是总结回顾，完善架构图，把架构图中那些模糊的或者空着的模块重新补充完善，把一些可复用的实现放入自己的代码库中。

现在是浏览阶段，并不适合过早涉及代码的实现细节，要借助 nginx 的文档理解其整体架构和模块划分。经过几年的发展，nginx 现在的文档已经是很丰富了，nginx 的英文 wiki 上包含了各个模块的详细文档，faq 也涵盖了很丰富的论题，利用这些文档足以建立 nginx 的架构草图。所以浏览阶段主要的工作就是阅读文档和画架构草图了。

对于源码分析，工具是相当关键的。这几天阅读源码的过程，熟悉了三个杀手级的工具：scrapbook 离线文件管理小程序、graphviz 图形生成工具、leo－editor 文学编程理念的编辑器。

scrapbook 是 firefox 下一款轻量高效的离线文件管理扩展程序，利用 scrapbook 把 nginx 的 wiki 站点镜像到本地只需要几分钟而已，管理也相当简单，和书签类似。

graphviz 是通过编程画图的工具集合，用程序把图形的逻辑和表现表示出来，然后通过命令行执行适当的命令就可以解析生成图形。

leo- editor 与其说是一个工具平台，不如说是一套理念。和其他编辑器 ide 不同的是，leo 关注的是文章内容的内在逻辑和段落层次，文章的表现形式和格式是次要的。用 leo 的过程，其实就是在编程，虽然刚开始有些不适应，但习惯之后确实很爽，杀手级的体验感，很听话。

btw，充实的 8 天长假结束了，总结一下，这个假期没有出去玩，做了三件事情，第一件事情是基本完成了房子的装修，墙面、地板、家具，都是亲力亲为的成果，虽然没有节省多少开支，却增添了动手的乐趣；第二件事情是溜冰，体会了平衡的力量；第三件事情是练习画画，锻炼了观察与概括的能力。

nginx 源码分析（ **5** ）**-**方法（ **2** ）

利用 nginx wiki 和互联网收集了不少 nginx 相关的文档资料，但是仔细阅读之后发觉对理解 nginx 架构有直接帮助的资料不多，一些有帮助的资料也要结合阅读部分源码细节才能搞清楚所述其是，可能 nginx 在非俄国之外的环境下流行不久，应用还很简单，相关的英文和中文文档也就不够丰富的原因吧。

不过还是有一些金子的。

如果要了解 nginx 的概况和使用方法，wiki 足以满足需要，wiki 上有各个模块的概要和详细指令说明，也有丰富的配置文件示例，不过对于了解 nginx 系统架构和开发没有相关的文档资料。

nginx 的开发主要是指撰写自定义模块代码。这需要了解 nginx 的模块化设计思想，模块化也是 nginx 的一个重要思想，如果要整体上了解 nginx，从模块化入手是一个不错的起点。emiller 的 nginx 模块开发指引是目前最好的相关资料了（ http://emiller.info/nginx-modules-guide.html ），这份文档作为 nginx 的入门文档也是合适的，不过其中有些内容很晦涩，很难理解，要结合阅读源码，反复比对才能真正理解其内涵。

如果要从整体上了解 nginx 架构和源码结构，Joshua zhu 的广州技术沙龙讲座的 pdf 和那张大图是不错的材料，这份 pdf 可以在 wiki 的资源页面中找到地址链接。Joshua 也给了我一些建议和指引，使我少走了不少弯路，很快进入状态，感谢。

相信最好的文档就是源码本身了。随着阅读源码的量越来越大，也越来越深入，使我认识到最宝贵的文档就在源码本身，之前提到过，nginx 的代码质量很高，命名比较讲究，虽然很少注释，但是很有条理的结构体命名和变量命名使得阅读源码就像是阅读文档。不过要想顺利的保持这种感觉也不是一件简单的事情，觉得要做好如下几点：

1 ）熟悉 C 语言，尤其是对函数指针和宏定义要有足够深入的理解，nginx 是模块化的，它的模块化不同于 apache，它不是动态加载模块，而是把需要的模块都编译到系统中，这些模块可以充分利用系统核心提供的诸多高效的组

件，把数据拷贝降到最低的水平，所以这些模块的实现利用了大量的函数指针实现回掉操作，几乎是无函数指针不 nginx 的地步。

2) 重点关注 nginx 的命名，包括函数命名，结构体命名和变量命名，这些命名把 nginx 看似耦合紧密的实现代码清晰的分开为不同层次不同部分的组件和模块，这等效于注释。尤其要关注变量的命名，后面关于模块的分析中会再次重申这一点。

3) 写一个自定义的模块，利用 nginx 强大的内部组件，这是深入理解 nginx 的一个有效手段。

接下来的分析过程，着眼于两个重点，一个就是上面提到的模块化思想的剖析，力争结合自身理解把这个部分分析透彻；另一个重点是 nginx 的事件处理流程，这是高性能的核心，是 nginx 的 core。

nginx 源码分析（**6**）**-**模块化（**1**）

源码的 src/core 目录下实现了不少精巧的数据结构，最重要的有：内存池 ngx_pool_t、缓冲区 ngx_buf_t、缓冲区链 ngx_chain_t、字符串 ngx_str_t、数组 ngx_array_t、链表 ngx_list_t、队列 ngx_queue_t、基于 hash 的关联数组 ngx_hash_t、红黑树 ngx_rbtree_t、radix 树 ngx_radix_tree_t 等，这些数据结构频繁出现在源码中，而且这些结构之间也是彼此配合，联系紧密，很难孤立某一个出来。每个数据结构都提供了一批相关的操作接口，一般设置数据的接口（add）只是从内存池中分配空间并返回指向结构体的指针，然后利用这个指针再去设置成员，这使得此类接口的实现保持干净简洁。这些数据结构的实现分析是有一定难度的，不过我觉得不了解实现也并不影响理解整体的架构，只要知道有这么些重要的数据结构和基本的功用就足以满足需要了，我会在搞清楚整体架构之后再详细分析一下这些精致的玩意。

nginx 的模块化架构和实现方式是源码分析系列中最关键的一个部分。模块化是 nginx 的骨架，事件处理是 nginx 的心脏，而每个具体模块构成了 nginx 的血肉，摸清骨架，才能游刃有余。

先高屋建瓴的快速浏览一下 nginx 的模块化架构，后面再详细分析每一个重要数据结构和实现方式。

nginx 的模块在源码中对应着是 ngx_module_t 结构的变量，有一个全局的 ngx_module_t 指针数组，这个指针数组包含了当前编译版本支持的所有模块，这个指针数组的定义是在自动脚本生成的 objs/ngx_modules.c 文件中，下面是在我的机器上编译的 nginx 版本（0.8.9）的模块声明：

```
extern ngx_module_t ngx_core_module;
extern ngx_module_t ngx_errlog_module;
extern ngx_module_t ngx_conf_module;
extern ngx_module_t ngx_events_module;
extern ngx_module_t ngx_event_core_module;
extern ngx_module_t ngx_epoll_module;
```

```
extern ngx_module_t ngx_http_module;
extern ngx_module_t ngx_http_core_module;
extern ngx_module_t ngx_http_log_module;
extern ngx_module_t ngx_http_upstream_module;
extern ngx_module_t ngx_http_static_module;
extern ngx_module_t ngx_http_autoindex_module;
extern ngx_module_t ngx_http_index_module;
extern ngx_module_t ngx_http_auth_basic_module;
extern ngx_module_t ngx_http_access_module;
extern ngx_module_t ngx_http_limit_zone_module;
extern ngx_module_t ngx_http_limit_req_module;
extern ngx_module_t ngx_http_geo_module;
extern ngx_module_t ngx_http_map_module;
extern ngx_module_t ngx_http_referer_module;
extern ngx_module_t ngx_http_rewrite_module;
extern ngx_module_t ngx_http_proxy_module;
extern ngx_module_t ngx_http_fastcgi_module;
extern ngx_module_t ngx_http_memcached_module;
extern ngx_module_t ngx_http_empty_gif_module;
extern ngx_module_t ngx_http_browser_module;
extern ngx_module_t ngx_http_upstream_ip_hash_module;
extern ngx_module_t ngx_http_write_filter_module;
extern ngx_module_t ngx_http_header_filter_module;
extern ngx_module_t ngx_http_chunked_filter_module;
extern ngx_module_t ngx_http_range_header_filter_module;
extern ngx_module_t ngx_http_gzip_filter_module;
extern ngx_module_t ngx_http_postpone_filter_module;
extern ngx_module_t ngx_http_charset_filter_module;
extern ngx_module_t ngx_http_ssi_filter_module;
```

```
extern ngx_module_t ngx_http_userid_filter_module;
extern ngx_module_t ngx_http_headers_filter_module;
extern ngx_module_t ngx_http_copy_filter_module;
extern ngx_module_t ngx_http_range_body_filter_module;
extern ngx_module_t ngx_http_not_modified_filter_module;
ngx_module_t *ngx_modules[] = {
&ngx_core_module,
&ngx_errlog_module,
&ngx_conf_module,
&ngx_events_module,
&ngx_event_core_module,
&ngx_epoll_module,
&ngx_http_module,
&ngx_http_core_module,
&ngx_http_log_module,
&ngx_http_upstream_module,
&ngx_http_static_module,
&ngx_http_autoindex_module,
&ngx_http_index_module,
&ngx_http_auth_basic_module,
&ngx_http_access_module,
&ngx_http_limit_zone_module,
&ngx_http_limit_req_module,
&ngx_http_geo_module,
&ngx_http_map_module,
&ngx_http_referer_module,
&ngx_http_rewrite_module,
&ngx_http_proxy_module,
&ngx_http_fastcgi_module,
```

```
&ngx_http_memcached_module,
&ngx_http_empty_gif_module,
&ngx_http_browser_module,
&ngx_http_upstream_ip_hash_module,
&ngx_http_write_filter_module,
&ngx_http_header_filter_module,
&ngx_http_chunked_filter_module,
&ngx_http_range_header_filter_module,
&ngx_http_gzip_filter_module,
&ngx_http_postpone_filter_module,
&ngx_http_charset_filter_module,
&ngx_http_ssi_filter_module,
&ngx_http_userid_filter_module,
&ngx_http_headers_filter_module,
&ngx_http_copy_filter_module,
&ngx_http_range_body_filter_module,
&ngx_http_not_modified_filter_module,
NULL
};
```

这里只有每个模块变量的声明（注意 extern 声明符），而每个模块变量的定义包含在各自模块实现的文件中，比如 ngx_core_module 定义在 src/core/nginx.c 中:

```
ngx_module_t ngx_core_module = {
NGX_MODULE_V1,
&ngx_core_module_ctx, /* module context */
ngx_core_commands, /* module directives */
NGX_CORE_MODULE, /* module type */
NULL, /* init master */
NULL, /* init module */
```

```
NULL, /* init process */
NULL, /* init thread */
NULL, /* exit thread */
NULL, /* exit process */
NULL, /* exit master */
NGX_MODULE_V1_PADDING
};
```

提问：这么多的模块，在 nginx 中究竟是如何使用的呢，模块的功能是如何体现的呢？

这就要先说一下 nginx 启动的过程了，nginx 是一个 master 主进程 + 多个 worker 子进程的工作模式 (详细会在分析事件处理的时候解释)，nginx 主进程启动的过程中会按照初始化 master、初始化模块、初始化工作进程、（初始化线程、退出线程）、退出工作进程、退出 master 顺序进行，而在这些子过程内部和子过程之间，又会有读取配置、创建配置、初始化配置、合并配置、http 解析、http 过滤、http 输出、http 代理等过程，在这些过程开始前后、过程中、结束前后等时机，nginx 调用合适的模块接口完成特定的任务。

所谓的合适模块接口，是各个模块通过一些方式注册到系统内的回调函数，这些回调函数都要符合一定的接口规范，比如上面那个 ngx_core_module 变量的定义初始化中，那些赋值为 NULL 的都是函数指针，如果要注册一个回调函数，就可以按照函数指针指定的接口编写函数，然后通过赋值给这些函数指针来注册回调函数，不同的接口被 nginx 调用的时机是不同的，比如 init_master 是在初始化 master 的时候被调用。

到这里，应该对 nginx 的模块化架构有了一些感性认识了，接下来就进入具体实现方式的分析，会稍微复杂一些，最好能够结合源码加深一下理解。

nginx 源码分析（**7**）**-**模块化（**2**）

分析 nginx 的模块化架构的实现方式，就要从 ngx_module_t 结构体入手。

ngx_module_t 的声明在 src/core/ngx_conf_file.h 中：

```
#define NGX_MODULE_V1 0, 0, 0, 0, 0, 0, 1
#define NGX_MODULE_V1_PADDING 0, 0, 0, 0, 0, 0, 0, 0
struct ngx_module_s {
ngx_uint_t ctx_index;
ngx_uint_t index;
ngx_uint_t spare0;
ngx_uint_t spare1;
ngx_uint_t spare2;
ngx_uint_t spare3;
ngx_uint_t version;
void *ctx;
ngx_command_t *commands;
ngx_uint_t type;
ngx_int_t (*init_master)(ngx_log_t *log);
ngx_int_t (*init_module)(ngx_cycle_t *cycle);
ngx_int_t (*init_process)(ngx_cycle_t *cycle);
ngx_int_t (*init_thread)(ngx_cycle_t *cycle);
void (*exit_thread)(ngx_cycle_t *cycle);
void (*exit_process)(ngx_cycle_t *cycle);
void (*exit_master)(ngx_cycle_t *cycle);
uintptr_t spare_hook0;
uintptr_t spare_hook1;
uintptr_t spare_hook2;
uintptr_t spare_hook3;
uintptr_t spare_hook4;
```

uintptr_t spare_hook5;

uintptr_t spare_hook6;

uintptr_t spare_hook7;

};

index 是一个模块计数器，按照每个模块在 ngx_modules[]数组中的声明顺序（见 objs/ngx_modules.c），从 0 开始依次给每个模块进行编号：

ngx_max_module = 0;

for (i = 0; ngx_modules[i]; i++) {

ngx_modules[i]->index = ngx_max_module++;

}（见 src/core/nginx.c）

ctx_index 是分类的模块计数器，nginx 的模块可以分为四种：core、event、http 和 mail，每一种的模块又会各自计数一下，这个 ctx_index 就是每个模块在其所属类组的计数值:

ngx_event_max_module = 0;

for (i = 0; ngx_modules[i]; i++) {

if (ngx_modules[i]->type != NGX_EVENT_MODULE) {

continue;

}

ngx_modules[i]->ctx_index = ngx_event_max_module++;

}（见 src/event/ngx_event.c）

ngx_http_max_module = 0;

for (m = 0; ngx_modules[m]; m++) {

if (ngx_modules[m]->type != NGX_HTTP_MODULE) {

continue;

}

ngx_modules[m]->ctx_index = ngx_http_max_module++;

}（见 src/http/ngx_http.c）

ngx_mail_max_module = 0;

for (m = 0; ngx_modules[m]; m++) {

```
if (ngx_modules[m]->type != NGX_MAIL_MODULE) {
continue;
}
    ngx_modules[m]->ctx_index = ngx_mail_max_module++;
}
```
（见 src/mail/ngx_mail.c）

ctx 是模块的上下文，不同种类的模块有不同的上下文，四类模块就有四种模块上下文，实现为四个不同的结构体，所以 ctx 是 void *。这是一个很重要的字段，后面会详细剖析。

commands 是模块的指令集，nginx 的每个模块都可以实现一些自定义的指令，这些指令写在配置文件的适当配置项中，每一个指令在源码中对应着一个 ngx_command_t 结构的变量，nginx 会从配置文件中把模块的指令读取出来放到模块的 commands 指令数组中，这些指令一般是把配置项的参数值赋给一些程序中的变量或者是在不同的变量之间合并或转换数据（例如 include 指令），指令可以带参数也可以不带参数，你可以把这些指令想象为 unix 的命令行或者是一种模板语言的指令。在 nginx 的 wiki 上有每个系统内置模块的指令说明。指令集会在下一篇中详细剖析。

type 就是模块的种类，前面已经说过，nginx 模块分为 core、event、http 和 mail 四类，type 用宏定义标识四个分类。

init_master、init_module、init_process、init_thread、exit_thread、exit_process、exit_master 是函数指针，指向模块实现的自定义回调函数，这些回调函数分别在初始化 master、初始化模块、初始化工作进程、初始化线程、退出线程、退出工作进程和退出 master 的时候被调用，如果模块需要在这些时机做处理，就可以实现对应的函数，并把它赋值给对应的函数指针来注册一个回调函数接口。

其余的参数没研究过，貌似从来没有用过，在定义模块的时候，最前面的 7 个字段和最后面的 8 个字段的初始化是使用宏 NGX_MODULE_V1 和 NGX_MODULE_V1_PADDING 完成的，例如：

```
ngx_module_t ngx_core_module = {
NGX_MODULE_V1,
&ngx_core_module_ctx, /* module context */
ngx_core_commands, /* module directives */
NGX_CORE_MODULE, /* module type */
NULL, /* init master */
NULL, /* init module */
NULL, /* init process */
NULL, /* init thread */
NULL, /* exit thread */
NULL, /* exit process */
NULL, /* exit master */
NGX_MODULE_V1_PADDING
};
```
（见 src/core/nginx.c）

接下来剖析一下 ngx_module_t 的 ctx 成员，这个成员的意义是每个模块的上下文，所谓的上下文，也就是这个模块究竟可以做什么，从前面的分析可以知道 nginx 把所有模块分为四类（core/event/http/mail），对应的，nginx 也认为模块的上下文是四种，分别用四个结构体表示：ngx_core_module_t、ngx_event_module_t、ngx_http_module_t、ngx_mail_module_t。也就是说，如果一个模块属于 core 分类，那么其上下文就是 ngx_core_module_t 结构的变量，其他类推。这四个结构体类似于 ngx_module_t，也是一些函数指针的集合，每个模块根据自己所属的分类，自定义一些操作函数，通过把这些操作函数赋值为对应分类结构体中的函数指针，这就注册了一个回调函数接口，从而就可以实现更细致的功能了，例如可以为 event 模块添加事件处理函数，可以为 http 模块添加过滤函数等。

这四个结构体和 commands 会在下一篇中详细剖析，现在把注意力转移一下，来品味一下 nginx 的命名，找出其规律，这对阅读源码的帮助是非常大的。

首先是模块，模块是 ngx_module_t 结构的变量，其命名格式为：ngx__module。

然后是模块上下文，模块上下文根据不同的模块分类分别是 ngx_core_module_t、ngx_event_module_t、 ngx_http_module_t 和 ngx_mail_module_t 结构的变量，其命名格式为：ngx__module_ctx。

接着是模块命令集，模块命令集是 ngx_command_t 的指针数组，其命名格式为：ngx__commands。

nginx 源码分析（**8**）**-**模块化（**3**）

接下来剖析模块的指令。模块的指令在源码中是 ngx_command_t 结构的变量，ngx_command_t 的声明在 src/core/ngx_conf_file.h 中:

struct ngx_command_s {
ngx_str_t name;
ngx_uint_t type;
char *(*set)(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
ngx_uint_t conf;
ngx_uint_t offset;
void *post;
};

name 是指令名称的字符串，不包含空格。

type 是标识符集，标识指令在配置文件中的合法位置和指令的参数个数。这是一个至少有 32bit 的无符号整形，前 16bit 用于标识位置，后 16bit 用于标识参数。

先看看参数的标识，宏定义在 src/core/ngx_conf_file.h 中:

#define NGX_CONF_NOARGS 0×00000001（没有参数）
#define NGX_CONF_TAKE1 0×00000002（有 1 个参数）
#define NGX_CONF_TAKE2 0×00000004（有 2 个参数）
#define NGX_CONF_TAKE3 0×00000008（有 3 个参数）
#define NGX_CONF_TAKE4 0×00000010（有 4 个参数）
#define NGX_CONF_TAKE5 0×00000020（有 5 个参数）
#define NGX_CONF_TAKE6 0×00000040（有 6 个参数）
#define NGX_CONF_TAKE7 0×00000080（有 7 个参数）

#define NGX_CONF_MAX_ARGS 8

#define NGX_CONF_TAKE12 (NGX_CONF_TAKE1|
NGX_CONF_TAKE2)（有 1 个或者有 2 个参数）
#define NGX_CONF_TAKE13 (NGX_CONF_TAKE1|NGX_CONF_TAKE3)

#define NGX_CONF_TAKE23 (NGX_CONF_TAKE2|
NGX_CONF_TAKE3)

#define NGX_CONF_TAKE123 (NGX_CONF_TAKE1|
NGX_CONF_TAKE2|NGX_CONF_TAKE3)

#define NGX_CONF_TAKE1234 (NGX_CONF_TAKE1|
NGX_CONF_TAKE2|NGX_CONF_TAKE3 \
|NGX_CONF_TAKE4)

#define NGX_CONF_ARGS_NUMBER 0×000000ff
#define NGX_CONF_BLOCK 0×00000100
#define NGX_CONF_FLAG 0×00000200（有一个布尔型参数）
#define NGX_CONF_ANY 0×00000400
#define NGX_CONF_1MORE 0×00000800（至多有 1 个参数）
#define NGX_CONF_2MORE 0×00001000（至多有 2 个参数）
#define NGX_CONF_MULTI 0×00002000

再看看位置的标识，位置的标识宏定义在几个文件中:

src/core/ngx_conf_file.h:

#define NGX_DIRECT_CONF 0×00010000

#define NGX_MAIN_CONF 0×01000000
#define NGX_ANY_CONF 0×0F000000

src/event/ngx_event.h:

#define NGX_EVENT_CONF 0×02000000

src/http/ngx_http_config.h:

#define NGX_HTTP_MAIN_CONF 0×02000000
#define NGX_HTTP_SRV_CONF 0×04000000
#define NGX_HTTP_LOC_CONF 0×08000000
#define NGX_HTTP_UPS_CONF 0×10000000
#define NGX_HTTP_SIF_CONF 0×20000000
#define NGX_HTTP_LIF_CONF 0×40000000
#define NGX_HTTP_LMT_CONF 0×80000000

src/mail/ngx_mail.h:

#define NGX_MAIL_MAIN_CONF 0×02000000

#define NGX_MAIL_SRV_CONF 0×04000000

要理解上面所谓的合法位置的真正含义，就要了解一下 nginx 的配置文件了，这里就不累述了，不影响下面的分析，我会在很后面的时候分析一下 nginx 的配置文件，因为那是一个 big topic。

set 是一个函数指针，这个函数主要是从配置文件中把该指令的参数（存放在 ngx_conf_t 中）转换为合适的数据类型并将转换后的值保存到模块的配置结构体中（void *conf），这个配置结构体又是用 void *指向的，应该可以料到这说明每个模块的配置结构体是不同的，这些结构体命名格式为:ngx__conf_t，至于要把转换后的值放到配置结构体的什么位置，就要依靠 offset 了，offset 是调用了 offsetof 函数计算出的结构体中某个成员的偏移位置。

并不是所有的模块都要定义一个配置结构体，因为 set 也可能是一个简单的操作函数，它可能只是从配置中（ngx_conf_t）读取一些数据进行简单的操作，比如 errlog 模块的“error_log”指令就是调用 ngx_error_log 写一条日志，并不需要存储什么配置数据。

conf 和 offset，offset 前面已经提到，它是配置结构体中成员的偏移 。conf 也是一个偏移值，不过它是配置文件结构体的（ngx_conf_t）成员 ctx 的成员的偏移，一般是用来把 ctx 中指定偏移位置的成员赋值给 void *conf。

post 指向模块读配置的时候需要的一些零碎变量。

从上面的分析可以看出，每个模块会映射到配置文件中的某个位置，全局位置的配置会被下一级的配置继承，比如 http_main 会被 http_svr 继承，http_svr 会被 http_loc 继承，这些继承在源码中是调用模块上下文的合并配置的接口完成的。

ngx_command_t 的 set 成员也可以作为一个回调函数，通过把自定义的操作函数赋值给 set 来注册一些操作。

到目前为止，已经了解不少回调函数了，这些回调函数用来注册模块的自定义操作，我有时称它为接口，有时称它为回调函数，有点混乱，接下来的分析文章中，进行一下统一，全部称为钩子（hook）。把现在已经分析过的钩子罗列一下：

ngx_module_t -> init_master

ngx_module_t -> init_module

ngx_module_t -> init_process

ngx_module_t -> init_thread

ngx_module_t -> exit_thread

ngx_module_t -> exit_process

ngx_module_t -> exit_master

ngx_command_t -> set

下一篇剖析模块上下文的时候，会有更多的钩子，这就是为什么要对 c 语言的指针深入理解的原因了，nginx 中到处都是钩子，假如要自己写一个模块，可以通过这些钩子把自己的模块挂到 nginx 的处理流中，参与到 nginx 运行的每个特定阶段，当然，也不是随意的嵌入，要精确定义模块何时如何产生作用才是有意义的，这不是一件轻松的事情。

nginx源码分析（**9**）**-**模块化（**4**）

模块的上下文是四个结构体定义的**:**

ngx_core_module_t、ngx_event_module_t、ngx_http_module_t、ngx_mail_module_t，分别对应于四类模块。

typedef struct {

ngx_str_t name;

void *(*create_conf)(ngx_cycle_t *cycle);

char *(*init_conf)(ngx_cycle_t *cycle, void *conf);

} ngx_core_module_t; (见 src/core/ngx_conf_file.h)

typedef struct {

ngx_int_t (*add)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);

ngx_int_t (*del)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);

ngx_int_t (*enable)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);

ngx_int_t (*disable)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);

ngx_int_t (*add_conn)(ngx_connection_t *c);

ngx_int_t (*del_conn)(ngx_connection_t *c, ngx_uint_t flags);

ngx_int_t (*process_changes)(ngx_cycle_t *cycle, ngx_uint_t nowait);

ngx_int_t (*process_events)(ngx_cycle_t *cycle, ngx_msec_t timer, ngx_uint_t flags);

ngx_int_t (*init)(ngx_cycle_t *cycle, ngx_msec_t timer);

void (*done)(ngx_cycle_t *cycle);

} ngx_event_actions_t;

typedef struct {

ngx_str_t *name;

void *(*create_conf)(ngx_cycle_t *cycle);

char *(*init_conf)(ngx_cycle_t *cycle, void *conf);

```
    ngx_event_actions_t actions;
} ngx_event_module_t; ( 见 src/event/ngx_event.h )
    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; ( 见 src/http/ngx_http_config.h )
    typedef struct {
ngx_mail_protocol_t *protocol;
    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);
} ngx_mail_module_t; ( 见 src/mail/ngx_mail.h )
```

这四个结构体都提供了钩子供模块注册与上下文有关的操作，核心模块提供了 create_conf 和 init_conf 两个钩子；event 模块也提供了 create_conf 和 init_conf 两个钩子，除此之外还提供了一集操作事件的钩子；http 模块提供了几个在读取配置文件前后和操作 mail/srv/loc 配置项时候执行的钩子；mail 模块也提供一些类似于 http 模块的钩子，不过比 http 模块简单一些。

只要设置了钩子，这些钩子就会在特定的时机被正确调用，只不过模块的种类不同，其钩子调用的时机也就大不同，这是由于 nginx 的四种模块之间的差别很大：核心类模块一般是全局性的模块，它们会在系统的许多部分被使用，比如 errlog 模块负责写错误日志，它被使用在许多地方；event 类模块是事件

驱动相关的模块，nginx 在不同操作系统上都会有结合该操作系统特有接口的事件处理模块，比如在 linux 中可以使用 epoll 模块，在 freebsd 中可以使用 kqueue 模块，在 solaris 中可以使用 devpoll 事件等；http 类模块是用于处理 http 输入，产生输出，过滤输出，负载均衡等的模块，这是 nginx 作为 web 服务器的核心部分，Emiller 的论文就是关于 http 模块的开发指引；mail 类模块是实现邮件代理的模块，实现了 smtp/pop3/imap 等协议的邮件代理。

到这里，已经把 nginx 拆开为一个个的模块，后面的分析就可以对每个模块进行详细的剖析，也就可以进入阅读源码的精读阶段了。在这之前，再尝试把这些独立的模块串一下，从 main 开始，看看 nginx 是何时在何地调度这些模块干活的。

```
    main
{
…
    // 所有模块点一下数
ngx_max_module = 0;
for (i = 0; ngx_modules[i]; i++) {
ngx_modules[i]->index = ngx_max_module++;
}
    // 主要的初始化过程，全局的数据存放到 ngx_cycle_t 结构的变量中
cycle = ngx_init_cycle(&init_cycle);

    …
    // 启动进程干活
if (ngx_process == NGX_PROCESS_SINGLE) {
ngx_single_process_cycle(cycle);
    } else {
ngx_master_process_cycle(cycle);
}
    return 0;
}
```

```
    ngx_init_cycle
{
…
    // 调度 core 模块的钩子 create_conf，并且把创建的配置结构体变量存放到 cyclep->conf_ctx 中
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;
}
}
    …
ngx_conf_parse
…
    // 调度 core 模块的钩子 init_conf，设置刚才创建的配置结构体变量（用从配置文件中读取的数据赋值）
for (i = 0; ngx_modules[i]; i++) {
if (ngx_modules[i]->type != NGX_CORE_MODULE) {
continue;
}
    module = ngx_modules[i]->ctx;
```

```
    if (module->init_conf) {
if (module->init_conf(cycle, cycle->conf_ctx[ngx_modules[i]-
>index])
== NGX_CONF_ERROR)
{
ngx_destroy_cycle_pools(&conf);
return NULL;
}
}
}
    ...
    // 调度所有模块的 init_module 钩子，初始化模块
for (i = 0; ngx_modules[i]; i++) {
if (ngx_modules[i]->init_module) {
if (ngx_modules[i]->init_module(cycle) != NGX_OK) {
/* fatal */
exit(1);
}
}
}
    ...
}
    ngx_conf_parse
{
...
ngx_conf_handler
...
}
```

```
    ngx_conf_handler
{
...
    // 处理模块的指令集
for (i = 0; ngx_modules[i]; i++) {
    /* look up the directive in the appropriate modules */
    if (ngx_modules[i]->type != NGX_CONF_MODULE
&& ngx_modules[i]->type != cf->module_type)
{
continue;
}
    cmd = ngx_modules[i]->commands;
if (cmd == NULL) {
continue;
}
    for ( /* void */ ; cmd->name.len; cmd++) {
    if (name->len != cmd->name.len) {
continue;
}
    if (ngx_strcmp(name->data, cmd->name.data) != 0) {
continue;
}
    /* is the directive's location right ? */
    if (!(cmd->type & cf->cmd_type)) {
if (cmd->type & NGX_CONF_MULTI) {
multi = 1;
continue;
}
```

```
        goto not_allowed;
}
        if (!(cmd->type & NGX_CONF_BLOCK) && last != NGX_OK) {
ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
“directive \”%s\” is not terminated by \”;\”",
name->data);
return NGX_ERROR;
}
        if ((cmd->type & NGX_CONF_BLOCK) && last !=
NGX_CONF_BLOCK_START) {
ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
“directive \”%s\” has no opening \”{\”",
name->data);
return NGX_ERROR;
}
        /* is the directive’s argument count right ? */
        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;
}
}
        /* set up the directive’s configuration context */
        conf = NULL;
        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];
}
}
        // 调度指令的钩子 set
rv = cmd->set(cf, cmd, conf);
        if (rv == NGX_CONF_OK) {
return NGX_OK;
}
```

```
    if (rv == NGX_CONF_ERROR) {
return NGX_ERROR;
}
    ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
“\”%s\” directive %s”, name->data, rv);
    return NGX_ERROR;
}
}
    …
}
    ngx_master_process_cycle
{
…
ngx_start_worker_processes
…
ngx_master_process_exit
…
}
    ngx_start_worker_processes
{
…
    // for i = 0 to n
ngx_worker_process_cycle
    …
}
    ngx_worker_process_cycle
{
…
ngx_worker_process_init
```

```
…
ngx_worker_process_exit
…
}
    ngx_worker_process_init
{
…
    // 调度所有模块的钩子 init_process
for (i = 0; ngx_modules[i]; i++) {
if (ngx_modules[i]->init_process) {
if (ngx_modules[i]->init_process(cycle) == NGX_ERROR) {
/* fatal */
exit(2);
}
}
}
    …
}
    ngx_worker_process_exit
{
…
    // 调度所有模块的钩子 exit_process
for (i = 0; ngx_modules[i]; i++) {
if (ngx_modules[i]->exit_process) {
ngx_modules[i]->exit_process(cycle);
}
}
    …
}
```

```
    ngx_master_process_exit
{
…
    // 调度所有模块的钩子 exit_master
for (i = 0; ngx_modules[i]; i++) {
if (ngx_modules[i]->exit_master) {
ngx_modules[i]->exit_master(cycle);
}
}
    …
}
```

在这个调度流中，我们并没有看到 event 类模块、http 类模块和 mail 类模块钩子的调度，那是由于这三类模块注册到 ngx_module_t 的钩子和 ngx_command_t 的钩子上的操作中调度了这些独特的模块上下文的钩子。

nginx 源码分析（**10**）**-**启动过程分析

ginx 有两个重要头文件：ngx_config.h 和 ngx_core.h。

src/core/ngx_config.h 文件中包含的是和操作系统相关的宏定义和头文件，其中又会包含 objs/ngx_auto_headers.h 和 src/os/unix/ngx__config.h，前面提到过，这个头文件是自动脚本检验操作系统后生成的，这个头文件中包含了一些宏定义，这些宏定义说明了存在哪些与特定操作系统有关的头文件，并依此判断出操作系统的种类（linux 、freebsd、solaris、darwin 等）,根据判断出的操作系统种类，ngx_config.h 包含具体的与操作系统有关的头文件 src/os/unix/ngx__config.h，这个头文件中包含几乎所有需要的系统头文件，并包含了 objs/ngx_auto_config.h，这是自动脚本生成的另一个头文件，其中包含一些宏定义，这些宏定义说明当前操作系统中存在的特性，根据这些特性又判断出可以支持的 nginx 内嵌模块情况，比如 rewrite 模块需要 pcre 的支持等。

src/core/ngx_core.h 文件中包含的是 nginx 中几乎所有实现代码的头文件，包含这个头文件，就可以访问 nginx 中的所有数据结构和函数接口。

几乎所有的模块都包含了 ngx_config.h 和 ngx_core.h。所以，在 nginx 的任何实现代码中，可以直接使用很多操作系统的接口和 nginx 实现的接口。

纵观整个 nginx 的代码，可以大致分为三块，一块是一些重要的数据结构及其操作接口，代码主要在 src/core 和 src/os/unix 目录下；第二块是模块的实现，四个种类几十个模块的实现分散在 src/core、src/event、src/http、src/mail 目录下；第三块是启动过程的代码，上一篇也大致列了一下启动调用的函数序列，其代码分布在 src/core 和 src/os/unix 目录下。

0.8.9 版本 nginx 的核心类模块有 7 个，event 类模块有 10 个，http 类模块有 47 个，mail 类模块有 7 个。另外还有一个模块是没有上下文的，是 conf 模块，所以准确的说 nginx 的模块有五种：

core/ngx_conf_file.h:#define NGX_CORE_MODULE 0×45524F43 /* “CORE” */
core/ngx_conf_file.h:#define NGX_CONF_MODULE 0×464E4F43 /* “CONF” */
event/ngx_event.h:#define NGX_EVENT_MODULE 0×544E5645 /* “EVNT” */
http/ngx_http_config.h:#define NGX_HTTP_MODULE 0×50545448 /* “HTTP” */
mail/ngx_mail.h:#define NGX_MAIL_MODULE 0×4C49414D /* “MAIL” */

上一篇简单的走了一遍 nginx 的启动流程，列了一下函数调用序列，流程中最重要的是三个函数:main(src/core /nginx.c)->ngx_init_cycle(src/core /ngx_cycle.c)->ngx_master_process_cycle(src/os/unix /ngx_process_cycle.c)。

下面会更深入细致的分析一下启动流程，重点围绕上述三个函数进行。

nginx 的启动过程是从设置一个变量开始的：ngx_cycle_t *ngx_cycle(src/core/ngx_cycle.c)，ngx_cycle_t 是一个重要的数据结构，它是一个很重要的容器，保存了一些全局性质的数据，在整个系统内都会被使用到。

struct ngx_cycle_s {
void ****conf_ctx;
ngx_pool_t *pool;
ngx_log_t *log;
ngx_log_t new_log;
ngx_connection_t **files;
ngx_connection_t *free_connections;
ngx_uint_t free_connection_n;
ngx_array_t listening;
ngx_array_t pathes;

ngx_list_t open_files;

ngx_list_t shared_memory;

ngx_uint_t connection_n;

ngx_uint_t files_n;

ngx_connection_t *connections;

ngx_event_t *read_events;

ngx_event_t *write_events;

ngx_cycle_t *old_cycle;

ngx_str_t conf_file;

ngx_str_t conf_param;

ngx_str_t conf_prefix;

ngx_str_t prefix;

ngx_str_t lock_file;

ngx_str_t hostname;

};（见 src/core/ngx_cycle.h）

下面从 main 开始详细分析启动流程中处理过程，在遇到设置 ngx_cycle_t 字段的时候再详细解释这个字段。

main 函数的处理过程可以分为以下步骤：

1、从控制台获取参数并处理:ngx_get_options(argc, argv)；

2、简单初始化，初始化一些数据结构和模块:ngx_debug_init(),ngx_time_init(),ngx_regex_init(),ngx_log_init(),ngx_ssl_init()；

3、初始化局部的 ngx_cycle_t init_cycle 结构体变量:

ngx_memzero(&init_cycle, sizeof(ngx_cycle_t));

init_cycle.log = log;

ngx_cycle = &init_cycle;

init_cycle.pool = ngx_create_pool(1024, log);

if (init_cycle.pool == NULL) {

return 1;
}

4、保存参数，设置几个全局变量：ngx_argc,ngx_os_argv,ngx_argv,ngx_os_environ；

5、调用 ngx_process_options，设置 init_cycle 的一些字段，这些字段是从控制台的命令中取得的：conf_prefix（config prefix path ）、prefix（prefix path:-p prefix）、conf_file（配置文件路径:-c filenname）、conf_param(-g directives)，另外还把 init_cycle.log.log_level 设置为 NGX_LOG_INFO；

6、调用 ngx_os_init，这个调用会设置一些全局变量，这些全局变量和操作系统相关，比如:ngx_pagesize,ngx_cacheline_size,ngx_ncpu,ngx_cpuinfo(),ngx_max_sockets 等；

7、调用初始化函数 ngx_crc32_table_init()；

8、调用 ngx_set_inherited_sockets(&init_cycle)，初始化 init_cycle.listening，这是一个 ngx_listening_t 的结构数组，其 socket_fd 是从环境变量 NGINX 中读取的；

9、对系统所有模块点一下数，然后进入 ngx_init_cycle 作主要的模块相关的初始化，init_cycle 作为旧的全局设置传进去，这个函数会创建一下新的 ngx_cycle_t 变量，并返回其指针：

```
ngx_max_module = 0;
for (i = 0; ngx_modules[i]; i++) {
ngx_modules[i]->index = ngx_max_module++;
}
```

cycle = ngx_init_cycle(&init_cycle);

10、与信号量相关的一些操作代码；

11、多进程的情况下，调用 ngx_master_process_cycle(cycle)，单进程情况下调用 ngx_single_process_cycle 完成最后的启动工作。

ngx_init_cycle 函数的处理过程分为以下步骤：

1、调用 ngx_timezone_update()、ngx_timeofday()和 ngx_time_update(0, 0)做时间校准，nginx 的时间以及计数器相关的内容以后作专题介绍吧；

2、创建一个新的 ngx_cycle_t 变量 cycle，并且初始化其大部分的成员字段，有一些是从传入的 old_cycle 直接拷贝过来的，这些字段包括：log,conf_prefix,prefix,conf_file,conf_param；还有一些字段会判断一下 old_cycle 中是否存在，如果存在，则取得这些字段的占用空间，在 cycle 中申请等大的空间，并初始化（不拷贝），否则就申请默认大小的空间，这些字段有:pathes,open_files,share_memory,listening；还有一些字段是重新创建或者第一次赋值的：
pool,new_log.log_level(=NGX_LOG_ERR),old_cycle(=old_cycle),hostname(gethostname)；
最重要的一个字段是 conf_ctx,它被初始化为 ngx_max_module 个 void *指针，这预示着 conf_ctx 是所有模块的配置结构的指针数组；

3、调用所有核心类模块的钩子 create_conf，并把返回的配置结构指针放到 conf_ctx 数组中，偏移位置为 ngx_module_t.index；

4、从命令行和配置文件中把所有配置更新到 cycle 的 conf_ctx 中，首先调用 ngx_conf_param 把命令行中的指令（-g directives）转换为配置结构并把指针加入到 cycle.conf_ctx 中，接着调用 ngx_conf_parse(..,filename)把配置文件中的指令转换为配置结构并把指针加入到 cycle.conf_ctx 中。

ngx_conf_param 最后也会调用 ngx_conf_parse(..,NULL)，所以配置的更新主要是在 ngx_conf_parse 中进行的，这个函数中有一个 for 循环，每次循环调用 ngx_conf_read_token 取得一个配置指令（以；结尾），然后调用 ngx_conf_handler 处理这条指令，ngx_conf_handler 每次都会遍历所有模块的指令集，查找这条配置指令并分析其合法性，如果指令正确，则会创建配置结构并把指针加入到 cycle.conf_ctx 中，配置结构的赋值是调用该指令的钩子 set 完成的。

遍历指令集的过程首先是遍历所有的核心类模块，若是 event 类的指令，则会遍历到 ngx_events_module，这个模块是属于核心类的，其钩子 set 又会嵌套调用 ngx_conf_parse 去遍历所有的 event 类模块，同样的，若是 http 类指令，则会遍历到 ngx_http_module，该模块的钩子 set 进一步遍历所有的 http 类模块，mail 类指令会遍历到 ngx_mail_module，该模块的钩子进一步遍历到所有的 mail 类模块。要特别注意的是：这三个遍历过程中会在适当的时机调用 event 类模块、http 类模块和 mail 类模块的创建配置和初始化配置的钩子。从这里可以看出，event、http、mail 三类模块的钩子是配置中的指令驱动的；

5、获得核心模块 ngx_core_dodule 的配置结构，然后调用 ngx_create_pidfile 创建 pid 文件。获取配置结构的代码：ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module)，这里的 ngx_get_conf 是一个宏定义：#define ngx_get_conf(conf_ctx, module) conf_ctx[module.index];

6、调用 ngx_test_lockfile(filename,log)，ngx_create_pathes(cycle,user)，接着打开 errorlog 文件并赋值给 cycle->new_log.file：cycle->new_log.file = ngx_conf_open_file(cycle, &error_log);

7、打开新文件，在第 2 步的时候提到 cycle->open_files 这个链表是空的，只是给它预先分配了空间，并没有数据，这里之所以可能会有文件被打开，估计是前面读配置文件的时候，调用各个钩子的过程中，填充了这个链表，把 ngx_open_file_t 结构变量填充进来（结构体中包含要打开文件的路径信息），这是我猜测的，之后再验证：）接着修改一下 cycle 的成员：cycle->log = &cycle->new_log;pool->log = &cycle->new_log;

8、创建共享内存，和 open_files 类似，在第 2 步的时候 cycle－>share_memory 也初始化为一个空的链表，也是预分配了空间，如果此时链表中已经被填充了 ngx_shm_zone_t 结构变量（其中包含需要共享内存的尺寸和标识等信息），那么这里就会分配共享内存，并且调用合适的初始化钩子

初始化分配的共享内存，每块共享内存都会有 name 标识，这里也会做一些排重，已经分配的就不会再去分配，从对 open_files 和 share_memory 的处理过程可以看出，nginx 在资源管理上是集中分配的，请求资源的时候分配说明性的结构变量，然后在恰当的时机才去真正分配资源；

9、处理 listening sockets，cycle->listening 是 ngx_listening_t 结构的数组，把 cycle->listening 于 old_cycle->listening 进行比较，设置 cycle->listening 的一些状态信息，接着调用 ngx_open_listening_sockets(cycle)启动 cycle->listening 中的所有监听 socket，循环调用 socket,bind,listen 完成服务端监听 socket 的启动。接着调用 ngx_configure_listening_sockets(cycle)配置监听 socket，会根据 ngx_listening_t 中的状态信息设置 socket 的读写缓存和 TCP_DEFER_ACCEPT；

10、调用所有模块的钩子 init_module；

11、关闭或者删除一些残留在 old_cycle 中的资源，首先释放不用的共享内存，接着关闭不使用的监听 socket，再关闭不使用的打开文件，最后把 old_cycle 放入 ngx_old_cycles 中，这是一个 ngx_cycle_t *的数组，最后设定一个定时器，定期回调 ngx_cleaner_event 清理 ngx_old_cycles，这里设置了 30000ms 清理一次。

ngx_master_process_cycle 的分析留待下一篇，这个函数会启动工作进程干活，并且会处理信号量，处理的过程中会杀死或者创建新的进程。

nginx 源码分析（**11**）**-**进程启动分析（**1**）

nginx 的进程启动过程是在 ngx_master_process_cycle（src/os/unix/ngx_process_cycle.c）中完成的(单进程是通过 ngx_single_process_cycle 完成，这里只分析多进程的情况)，在 ngx_master_process_cycle 中，会根据配置文件的 worker_processes 值创建多个子进程，即一个 master 进程和多个 worker 进程。worker 进程之间、master 与 worker 之间、master 与外部之间保持通信，worker 之间以及 master 与 worker 之间是通过 socketpair 进行通信的，socketpair 是一对全双工的无名 socket，可以当作管道使用，和管道不同的是，每条 socket 既可以读也可以写，而管道只能用于写或者用于读；master 与外部之间是通过信号通信的。

master 进程主要进行一些全局性的初始化工作和管理 worker 的工作；事件处理是在 worker 中进行的。

进程启动的过程中，有一些重要的全局数据会被设置，最重要的是进程表 ngx_processes，master 每创建一个 worker 都会把一个设置好的 ngx_process_t 结构变量放入 ngx_processes 中，进程表长度为 1024，刚创建的进程存放在 ngx_process_slot 位置，ngx_last_process 是进程表中最后一个存量进程的下一个位置，ngx_process_t 是进程在 nginx 中的抽象：

```
typedef void (*ngx_spawn_proc_pt) (ngx_cycle_t *cycle, void *data);
typedef struct {
ngx_pid_t pid;
int status;
ngx_socket_t channel[2];
ngx_spawn_proc_pt proc;
void *data;
char *name;
```

unsigned respawn:1;
unsigned just_spawn:1;
unsigned detached:1;
unsigned exiting:1;
unsigned exited:1;
} ngx_process_t;(src/os/unix/ngx_process.h)

pid 是进程的 id；

status 是进程的退出状态；

channel[2]是 socketpair 创建的一对 socket 句柄；

proc 是进程的执行函数，data 为 proc 的参数；

最后的几个位域是进程的状态,respawn:重新创建的、just_spawn:第一次创建的、detached:分离的、exiting:正在退出、exited:已经退出。

进程间通信是利用 socketpair 创建的一对 socket 进行的，通信中传输的是 ngx_channel_t 结构变量：

typedef struct {
ngx_uint_t command;
ngx_pid_t pid;
ngx_int_t slot;
ngx_fd_t fd;
} ngx_channel_t;(src/os/unix/ngx_channel.h)

command 是要发送的命令，有 5 种：

#define NGX_CMD_OPEN_CHANNEL 1
#define NGX_CMD_CLOSE_CHANNEL 2
#define NGX_CMD_QUIT 3
#define NGX_CMD_TERMINATE 4
#define NGX_CMD_REOPEN 5

pid 是发送方进程的进程 id；

slot 是发送方进程在进程表中偏移位置；

fd 是发送给对方的句柄。

进程的启动过程是比较重要的一个环节，为了把这个过程分析透彻，下面会多采用注释代码的方式分析。

首先分析 ngx_master_process_cycle 函数，可以分解为以下各步骤：

1、master 设置一些需要处理的信号，这些信号包括 SIGCHLD,SIGALRM,SIGIO,SIGINT,NGX_RECONFIGURE_SIGNAL(SIGHUP),NGX_REOPEN_SIGNAL(SIGUSR1), NGX_NOACCEPT_SIGNAL(SIGWINCH),NGX_TERMINATE_SIGNAL(SIGTERM),NGX_SHUTDOWN_SIGNAL(SIGQUIT), NGX_CHANGEBIN_SIGNAL(SIGUSR2)；

2、调用 ngx_setproctilte 设置进程标题，title = “master process” + ngx_argv[0] + … + ngx_argv[ngx_argc-1]；

3、调用 ngx_start_worker_processes(cycle, ccf->worker_processes, NGX_PROCESS_RESPAWN)启动 worker 进程；

4、调用 ngx_start_cache_manager_processes(cycle, 0)启动文件 cache 管理进程，有些模块需要文件 cache，比如 fastcgi 模块，这些模块会把文件 cache 路径添加到 cycle->paths 中，文件 cache 管理进程会定期调用这些模块的文件 cache 处理钩子处理一下文件 cache；

5、master 循环处理信号量。

```
ngx_new_binary = 0;
delay = 0;
live = 1;
    for ( ;; ) {
// delay 用来设置等待 worker 退出的时间，master 接收了退出信号后首先发送退出信号给 worker，
// 而 worker 退出需要一些时间
```

```
if (delay) {
delay *= 2;
    ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
“temination cycle: %d”, delay);
    itv.it_interval.tv_sec = 0;
itv.it_interval.tv_usec = 0;
itv.it_value.tv_sec = delay / 1000;
itv.it_value.tv_usec = (delay % 1000 ) * 1000;
    // 设置定时器
if (setitimer(ITIMER_REAL, &itv, NULL) == -1) {
ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
“setitimer() failed”);
}
}
    ngx_log_debug0(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
“sigsuspend”);
    // 挂起信号量，等待定时器
sigsuspend(&set);
    ngx_time_update(0, 0);
    ngx_log_debug0(NGX_LOG_DEBUG_EVENT, cycle->log, 0, “wake
up”);
    // 收到了 SIGCHLD 信号，有 worker 退出（ngx_reap==1）
if (ngx_reap) {
ngx_reap = 0;
ngx_log_debug0(NGX_LOG_DEBUG_EVENT, cycle->log, 0, “reap
children”);
    // 处理所有 worker，如果有 worker 异常退出则重启这个 worker，如果
所有 worker 都退出
// 返回 0 赋值给 live
```

```
live = ngx_reap_children(cycle);
}
    // 如果 worker 都已经退出，
// 并且收到了 NGX_CMD_TERMINATE 命令或者 SIGTERM 信号或者 SIGINT 信号(ngx_terminate=1)
// 或者 NGX_CMD_QUIT 命令或者 SIGQUIT 信号(ngx_quit=1)，则 master 退出
if (!live && (ngx_terminate || ngx_quit)) {
ngx_master_process_exit(cycle);
}
    // 收到了 NGX_CMD_TERMINATE 命令或者 SIGTERM 信号或者 SIGINT 信号，
// 通知所有 worker 退出，并且等待 worker 退出
if (ngx_terminate) {
// 设置延时
if (delay == 0) {
delay = 50;
}
    if (delay > 1000) {
// 延时已到，给所有 worker 发送 SIGKILL 信号，强制杀死 worker
ngx_signal_worker_processes(cycle, SIGKILL);
} else {
// 给所有 worker 发送 SIGTERM 信号，通知 worker 退出
ngx_signal_worker_processes(cycle,
ngx_signal_value(NGX_TERMINATE_SIGNAL));
}
    continue;
}
```

```
    // 收到了 NGX_CMD_QUIT 命令或者 SIGQUIT 信号
if (ngx_quit) {
// 给所有 worker 发送 SIGQUIT 信号
ngx_signal_worker_processes(cycle,
ngx_signal_value(NGX_SHUTDOWN_SIGNAL));
    // 关闭所有监听的 socket
ls = cycle->listening.elts;
for (n = 0; n < cycle->listening.nelts; n++) {
if (ngx_close_socket(ls[n].fd) == -1) {
ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_socket_errno,
ngx_close_socket_n " %V failed",
&ls[n].addr_text);
}
}
cycle->listening.nelts = 0;
    continue;
}
    // 收到了 SIGHUP 信号
if (ngx_reconfigure) {
ngx_reconfigure = 0;
    // 代码已经被替换，重启 worker，不需要重新初始化配置
if (ngx_new_binary) {
ngx_start_worker_processes(cycle, ccf->worker_processes,
NGX_PROCESS_RESPAWN);
ngx_start_cache_manager_processes(cycle, 0);
ngx_noaccepting = 0;
    continue;
}
    ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "reconfiguring");
```

```
    // 重新初始化配置
cycle = ngx_init_cycle(cycle);
if (cycle == NULL) {
cycle = (ngx_cycle_t *) ngx_cycle;
continue;
}
    // 重启 worker
ngx_cycle = cycle;
ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx,
ngx_core_module);
ngx_start_worker_processes(cycle, ccf->worker_processes,
NGX_PROCESS_JUST_RESPAWN);
ngx_start_cache_manager_processes(cycle, 1);
live = 1;
ngx_signal_worker_processes(cycle,
ngx_signal_value(NGX_SHUTDOWN_SIGNAL));
}
    // 当 ngx_noaccepting=1 的时候会把 ngx_restart 设为 1，重启
worker
if (ngx_restart) {
ngx_restart = 0;
ngx_start_worker_processes(cycle, ccf->worker_processes,
NGX_PROCESS_RESPAWN);
ngx_start_cache_manager_processes(cycle, 0);
live = 1;
}
    // 收到 SIGUSR1 信号，重新打开 log 文件
if (ngx_reopen) {
ngx_reopen = 0;
```

```
ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, “reopening logs”);
ngx_reopen_files(cycle, ccf->user);
ngx_signal_worker_processes(cycle,
ngx_signal_value(NGX_REOPEN_SIGNAL));
}
    // 收到 SIGUSR2 信号，热代码替换
if (ngx_change_binary) {
ngx_change_binary = 0;
ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, “changing binary”);
// 调用 execve 执行新的代码
ngx_new_binary = ngx_exec_new_binary(cycle, ngx_argv);
}
    // 收到 SIGWINCH 信号，不再接收请求，worker 退出，master 不退出
if (ngx_noaccept) {
ngx_noaccept = 0;
ngx_noaccepting = 1;
ngx_signal_worker_processes(cycle,
ngx_signal_value(NGX_SHUTDOWN_SIGNAL));
}
}
```

真正创建 worker 子进程的函数是 ngx_start_worker_processes，这个函数本身很简单：

```
    static void
ngx_start_worker_processes(ngx_cycle_t *cycle, ngx_int_t n,
ngx_int_t type)
{
ngx_int_t i;
ngx_channel_t ch;
```

```
    ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, “start worker
processes”);
    // 传递给其他 worker 子进程的命令：打开通信管道
ch.command = NGX_CMD_OPEN_CHANNEL;
    // 创建 n 个 worker 子进程
for (i = 0; i < n; i++) {

    cpu_affinity = ngx_get_cpu_affinity(i);
    // ngx_spawn_process 创建 worker 子进程并初始化相关资源和属性，
// 然后执行子进程的执行函数 ngx_worker_process_cycle
ngx_spawn_process(cycle, ngx_worker_process_cycle, NULL,
"worker process", type);
    // master 父进程向所有已经创建的 worker 子进程（不包括本子进程）广
播消息，
// 告知当前 worker 子进程的进程 id、在进程表中的位置和管道句柄；这些
worker 子进程收到消息后，
// 会更新这些消息到自己进程空间的进程表，这样就可以实现任意两个 worker
子进程之间的通信了
// 好像 worker 子进程没有办法向 master 进程发送消息?
ch.pid = ngx_processes[ngx_process_slot].pid;
ch.slot = ngx_process_slot;
ch.fd = ngx_processes[ngx_process_slot].channel[0];
    ngx_pass_open_channel(cycle, &ch);
}
}
```

把 ngx_pass_open_channel 展开如下：

```
    static void
ngx_pass_open_channel(ngx_cycle_t *cycle, ngx_channel_t *ch)
{
ngx_int_t i;
```

```
    for (i = 0; i < ngx_last_process; i++) {
    // 跳过自己和异常的 worker
if (i == ngx_process_slot
|| ngx_processes[i].pid == -1
|| ngx_processes[i].channel[0] == -1)
{
continue;
}
    ngx_log_debug6(NGX_LOG_DEBUG_CORE, cycle->log, 0,
“pass channel s:%d pid:%P fd:%d to s:%i pid:%P fd:%d”,
ch->slot, ch->pid, ch->fd,
i, ngx_processes[i].pid,
ngx_processes[i].channel[0]);
    /* TODO: NGX_AGAIN */
    // 发送消息给其他的 worker
ngx_write_channel(ngx_processes[i].channel[0],
ch, sizeof(ngx_channel_t), cycle->log);
}
}
```

第三个要剖开的函数是创建子进程的 ngx_pid_t ngx_spawn_process(ngx_cycle_t *cycle, ngx_spawn_proc_pt proc, void *data, char *name, ngx_int_t respawn)，这个函数定义在 src/os/unix/ngx_process.c 中，proc 是子进程的执行函数，data 是其参数，name 是子进程的名字。

```
    {
u_long on;
ngx_pid_t pid;
ngx_int_t s; // 将要创建的子进程在进程表中的位置
```

```
    if (respawn >= 0) {
// 替换进程 ngx_processes[respawn]，可安全重用该进程表项
s = respawn;
    } else {
// 先找到一个被回收的进程表项
for (s = 0; s < ngx_last_process; s++) {
if (ngx_processes[s].pid == -1) {
break;
}
}
    // 进程表已满
if (s == NGX_MAX_PROCESSES) {
ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
“no more than %d processes can be spawned”,
NGX_MAX_PROCESSES);
return NGX_INVALID_PID;
}
}
    // 不是分离的子进程
if (respawn != NGX_PROCESS_DETACHED) {
    /* Solaris 9 still has no AF_LOCAL */
    // 创建一对已经连接的无名 socket（管道句柄）
if (socketpair(AF_UNIX, SOCK_STREAM, 0, ngx_processes[s].channel)
== -1)
{
ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
“socketpair() failed while spawning \”%s\”", name);
return NGX_INVALID_PID;
}
```

```
    ngx_log_debug2(NGX_LOG_DEBUG_CORE, cycle->log, 0,
“channel %d:%d”,
ngx_processes[s].channel[0],
ngx_processes[s].channel[1]);
    // 设置管道句柄为非阻塞模式
if (ngx_nonblocking(ngx_processes[s].channel[0]) == -1) {
ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
ngx_nonblocking_n ” failed while spawning \”%s\”",
name);
ngx_close_channel(ngx_processes[s].channel, cycle->log);
return NGX_INVALID_PID;
}
    if (ngx_nonblocking(ngx_processes[s].channel[1]) == -1) {
ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
ngx_nonblocking_n ” failed while spawning \”%s\”",
name);
ngx_close_channel(ngx_processes[s].channel, cycle->log);
return NGX_INVALID_PID;
}
    // 打开异步模式
on = 1;
if (ioctl(ngx_processes[s].channel[0], FIOASYNC, &on) == -1) {
ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
“ioctl(FIOASYNC) failed while spawning \”%s\”", name);
ngx_close_channel(ngx_processes[s].channel, cycle->log);
return NGX_INVALID_PID;
}
    // 设置异步 io 的所有者
if (fcntl(ngx_processes[s].channel[0], F_SETOWN, ngx_pid) == -1) {
```

```
ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
“fcntl(F_SETOWN) failed while spawning \”%s\”", name);
ngx_close_channel(ngx_processes[s].channel, cycle->log);
return NGX_INVALID_PID;
}
    // 执行了 exec 后关闭管道句柄
if (fcntl(ngx_processes[s].channel[0], F_SETFD, FD_CLOEXEC) == -1)
{
ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
“fcntl(FD_CLOEXEC) failed while spawning \”%s\”",
name);
ngx_close_channel(ngx_processes[s].channel, cycle->log);
return NGX_INVALID_PID;
}
    if (fcntl(ngx_processes[s].channel[1], F_SETFD, FD_CLOEXEC)
== -1) {
ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
“fcntl(FD_CLOEXEC) failed while spawning \”%s\”",
name);
ngx_close_channel(ngx_processes[s].channel, cycle->log);
return NGX_INVALID_PID;
}
    // 设置当前子进程的管道句柄
ngx_channel = ngx_processes[s].channel[1];
    } else {
ngx_processes[s].channel[0] = -1;
ngx_processes[s].channel[1] = -1;
}
```

```
    // 设置当前子进程的进程表项索引
ngx_process_slot = s;
    // 创建子进程
pid = fork();

    switch (pid) {
    case -1:
ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
“fork() failed while spawning \”%s\”", name);
ngx_close_channel(ngx_processes[s].channel, cycle->log);
return NGX_INVALID_PID;
    case 0:
// 设置当前子进程的进程 id
ngx_pid = ngx_getpid();
// 子进程运行执行函数
proc(cycle, data);
break;
    default:
break;
}

    ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, “start %s %P”,
name, pid);
    // 设置一些进程表项字段
ngx_processes[s].pid = pid;
ngx_processes[s].exited = 0;
    // 替换进程 ngx_processes[respawn]，不用设置其他进程表项字段了
if (respawn >= 0) {
return pid;
}
```

```
    // 设置其他的进程表项字段
ngx_processes[s].proc = proc;
ngx_processes[s].data = data;
ngx_processes[s].name = name;
ngx_processes[s].exiting = 0;
    // 设置进程表项的一些状态字段
switch (respawn) {
    case NGX_PROCESS_NORESPAWN:
ngx_processes[s].respawn = 0;
ngx_processes[s].just_spawn = 0;
ngx_processes[s].detached = 0;
break;
    case NGX_PROCESS_JUST_SPAWN:
ngx_processes[s].respawn = 0;
ngx_processes[s].just_spawn = 1;
ngx_processes[s].detached = 0;
break;
    case NGX_PROCESS_RESPAWN:
ngx_processes[s].respawn = 1;
ngx_processes[s].just_spawn = 0;
ngx_processes[s].detached = 0;
break;
    case NGX_PROCESS_JUST_RESPAWN:
ngx_processes[s].respawn = 1;
ngx_processes[s].just_spawn = 1;
ngx_processes[s].detached = 0;
break;
    // 分离的子进程，不受 master 控制？
case NGX_PROCESS_DETACHED:
```

```
ngx_processes[s].respawn = 0;
ngx_processes[s].just_spawn = 0;
ngx_processes[s].detached = 1;
break;
}
    if (s == ngx_last_process) {
ngx_last_process++;
}
    return pid;
}
```

nginx 源码分析（**12**）**-**进程启动分析（**2**）

第四个剖析的是 worker 子进程的执行函数 ngx_worker_process_cycle(src/os/unix/ngx_process_cycle.c)。

```
static void
ngx_worker_process_cycle(ngx_cycle_t *cycle, void *data)
{
ngx_uint_t i;
ngx_connection_t *c;
    // 初始化
ngx_worker_process_init(cycle, 1);
    ngx_setproctitle(”worker process”);
    #if (NGX_THREADS)
// 略去关于线程的代码
#endif
    for ( ;; ) {
    // 退出状态已设置，关闭所有连接
if (ngx_exiting) {
    c = cycle->connections;
    for (i = 0; i < cycle->connection_n; i++) {
    /* THREAD: lock */
    if (c[i].fd != -1 && c[i].idle) {
c[i].close = 1;
c[i].read->handler(c[i].read);
}
}
    if (ngx_event_timer_rbtree.root ==
ngx_event_timer_rbtree.sentinel)
```

```
{
ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, “exiting”);
    ngx_worker_process_exit(cycle);
}
}
    ngx_log_debug0(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
“worker cycle”);
    // 处理事件和计时
ngx_process_events_and_timers(cycle);
    // 收到 NGX_CMD_TERMINATE 命令
if (ngx_terminate) {
ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, “exiting”);
    // 清理后进程退出，会调用所有模块的钩子 exit_process
ngx_worker_process_exit(cycle);
}
    // 收到 NGX_CMD_QUIT 命令
if (ngx_quit) {
ngx_quit = 0;
ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0,
“gracefully shutting down”);
ngx_setproctitle(”worker process is shutting down”);
    if (!ngx_exiting) {
// 关闭监听 socket，设置退出状态
ngx_close_listening_sockets(cycle);
ngx_exiting = 1;
}
}
    // 收到 NGX_CMD_REOPEN 命令，重新打开 log
if (ngx_reopen) {
```

```
ngx_reopen = 0;
ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, “reopening logs”);
ngx_reopen_files(cycle, -1);
}
}
}
```

worker 子进程的初始化函数是 ngx_worker_process_init(ngx_cycle_t *cycle, ngx_uint_t priority)，这个函数可分解为以下步骤：

1、设置 ngx_process = NGX_PROCESS_WORKER,在 master 进程中这个变量被设置为 NGX_PROCESS_MASTER；

2、全局性的设置，根据全局的配置信息设置执行环境、优先级、限制 、setgid、setuid、信号初始化等；

3、调用所有模块的钩子 init_process；

4、关闭不使用的管道句柄，关闭当前 worker 子进程（本进程）的 channel[0]句柄和继承来的其他进程的 channel[1]句柄，本进程会使用其他进程的 channel[0]句柄发送消息，使用本进程的 channel[1]句柄监听事件：

```
for (n = 0; n < ngx_last_process; n++) {
if (ngx_processes[n].pid == -1) {
continue;
}
if (n == ngx_process_slot) {
continue;
}
if (ngx_processes[n].channel[1] == -1) {
continue;
}
if (close(ngx_processes[n].channel[1]) == -1) {
ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
```

“close() channel failed”);
}
}
if (close(ngx_processes[ngx_process_slot].channel[0]) == -1) {
ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
“close() channel failed”);
}

5、在本进程的 channel[1]句柄监听事件：

if (ngx_add_channel_event(cycle, ngx_channel,
NGX_READ_EVENT,
ngx_channel_handler)
== NGX_ERROR)
{
/* fatal */
exit(2);
}

ngx_add_channel_event 把句柄 ngx_channel(本进程的 channel[1])上建立的连接的可读事件加入事件监控队列，事件处理函数为 ngx_channel_hanlder(ngx_event_t *ev)。当有可读事件的时候（master 或者其他 worker 子进程调用 ngx_write_channel 向本进程的管道句柄发送了消息），ngx_channel_handler 负责处理消息，过程如下：

static void
ngx_channel_handler(ngx_event_t *ev)
{
ngx_int_t n;
ngx_channel_t ch;
ngx_connection_t *c;
if (ev->timedout) {
ev->timedout = 0;

```
return;
}
    c = ev->data;
    ngx_log_debug0(NGX_LOG_DEBUG_CORE, ev->log, 0, “channel handler”);
    for ( ;; ) {
    // 从管道句柄中读取消息
n = ngx_read_channel(c->fd, &ch, sizeof(ngx_channel_t), ev->log);
    ngx_log_debug1(NGX_LOG_DEBUG_CORE, ev->log, 0, “channel: %i”, n);
    if (n == NGX_ERROR) {
    if (ngx_event_flags & NGX_USE_EPOLL_EVENT) {
ngx_del_conn(c, 0);
}
    ngx_close_connection(c);
return;
}
    if (ngx_event_flags & NGX_USE_EVENTPORT_EVENT) {
if (ngx_add_event(ev, NGX_READ_EVENT, 0) == NGX_ERROR) {
return;
}
}
    if (n == NGX_AGAIN) {
return;
}
    ngx_log_debug1(NGX_LOG_DEBUG_CORE, ev->log, 0, “channel command: %d”, ch.command);
```

```
    // 处理消息命令
switch (ch.command) {
    case NGX_CMD_QUIT:
ngx_quit = 1;
break;
    case NGX_CMD_TERMINATE:
ngx_terminate = 1;
break;
    case NGX_CMD_REOPEN:
ngx_reopen = 1;
break;

    case NGX_CMD_OPEN_CHANNEL:
    ngx_log_debug3(NGX_LOG_DEBUG_CORE, ev->log, 0,
“get channel s:%i pid:%P fd:%d”,
ch.slot, ch.pid, ch.fd);
    ngx_processes[ch.slot].pid = ch.pid;
ngx_processes[ch.slot].channel[0] = ch.fd;
break;

    case NGX_CMD_CLOSE_CHANNEL:
    ngx_log_debug4(NGX_LOG_DEBUG_CORE, ev->log, 0,
“close channel s:%i pid:%P our:%P fd:%d”,
ch.slot, ch.pid, ngx_processes[ch.slot].pid,
ngx_processes[ch.slot].channel[0]);
    if (close(ngx_processes[ch.slot].channel[0]) == -1) {
ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_errno,
“close() channel failed”);
}
    ngx_processes[ch.slot].channel[0] = -1;
break;
```

```
}
}
}
```

nginx 的启动过程就分析完了，上面分析的只是启动过程的主要逻辑分支，这个过程中和每个模块相关的细节留待后续按功能分析的时候再剖析。

简单总结一下。

nginx 的启动过程可以划分为两个部分，第一部分是读取配置文件并设置全局的配置结构信息以及每个模块的配置结构信息，这期间会调用模块的 create_conf 钩子和 init_conf 钩子；第二部分是创建进程和进程间通信机制，master 进程负责管理各个 worker 子进程，通过 socketpair 向子进程发送消息，各个 worker 子进程服务利用事件机制处理请求，通过 socketpair 与其他子进程通信（发送消息或者接收消息），进程启动的各个适当时机会调用模块的 init_module 钩子、init_process 钩子、exit_process 钩子和 exit_master 钩子，init_master 钩子没有被调用过。

nginx 的全局变量非常多，这些全局变量作用不一，有些是作为全局的配置信息，比如 ngx_cycle；有些是作为重要数据的全局缓存，比如 ngx_processes；有些是状态信息，比如 ngx_process，ngx_quit。很难对所有的全局变量作出清晰的分类，而且要注意的是，子进程继承了父进程的全局变量之后，子进程和父进程就会独立处理这些全局变量，有些全局量需要在父子进程之间同步就要通过通信方式了，比如 ngx_processes（进程表）的同步。