

Nginx 开发从入门到精通

Nginx 开发从入门到精通

缘起

nginx 由于出色的性能, 在世界范围内受到了越来越多人的关注, 在淘宝内部它更是被广泛的使用, 众多的开发以及运维同学都迫切的想要了解 nginx 模块的开发以及它的内部原理, 但是国内却没有一本关于这方面的书, 源于此我们决定自己来写一本。本书的作者为淘宝核心系统服务器平台组的成员, 本书写作的思路是从模块开发逐渐过渡到 nginx 原理剖析。书籍的内容会定期在这里更新, 欢迎大家提出宝贵意见, 不管是本书的内容问题, 还是字词错误, 都欢迎大家提交 [issue](#)(章节标题的左侧有评注按钮), 我们会及时的跟进。

更新历史

日期	描述
2012/03/01	创建目录大纲
2012/03/28	增加了样章
2012/05/25	更新样章
2012/06/08	增加第 5 章
2012/06/11	增加第 4 章
2012/06/26	增加第 6 章(event module)
2012/06/27	更新第 5 章部分内容
2012/07/04	更新第 6 章 event module 部分内容
2012/07/12	增加第 12 章（请求头读取，subrequest 解析）
2012/08/14	增加第 2 章(nginx 基础架构及基础概念)
2012/08/14	增加第 2 章(ngx_str_t 数据结构介绍)
2012/08/17	增加第 7 章(模块开发高级篇之变量)
2012/08/25	增加第 11 章(nginx 的启动阶段部分内容)
2012/09/26	增加第 2 章(ngx_array_t,ngx_hash_t 及 ngx_pool_t 介绍)
2012/10/08	增加第 11 章(配置解析综述)
2012/10/12	增加第 2 章(ngx_hash_wildcard_t,ngx_hash_combined_t 及 ngx_hash_keys_arrays_t 介绍)
2012/10/21	增加第 2 章(ngx_chain_t,ngx_list_t 及 ngx_buf_t 介绍)
2012/11/09	增加第 12 章(请求体的读取和丢弃解析)
2012/11/24	更新第 2 章(ngx_buf_t 的部分字段以及其他一些书写错误和表达)
2012/12/18	更新第 11 章(解析 http 块)
2012/12/10	增加第 3 章的内容
2012/12/28	补充和完善了第 3 章的内容
2013/01/25	增加了第 2 章(nginx 的配置系统)
2013/02/18	增加了第 2 章(nginx 的模块化体系结构, nginx 的请求处理)
2013/03/05	增加了第 12 章部分内容(多阶段请求处理)
2013/03/08	完成第 11 章第 1 节(配置解析综述、ngx_http_block)
2013/04/16	完成第 9 章第 1 节(源码目录结构、configure 原理)
2013/09/30	完成第 12 章部分内容(多阶段执行链各个阶段解析)
2013/10/11	完成第 12 章部分内容(filter 解析)
2013/10/11	完成第 12 章部分内容(ssl 解析)

版权申明

本书的著作权归作者淘宝核心系统服务器平台组成员所有。你可以：

- 下载、保存以及打印本书
- 网络链接、转载本书的部分或者全部内容，但是必须在明显处提供读者访问本书发布网站的链接
- 在你的程序中任意使用本书所附的程序代码，但是由本书的程序所引起的任何问题，作者不承担任何责任

你不可以：

- 以任何形式出售本书的电子版或者打印版
- 擅自印刷、出版本书
- 以纸媒出版为目的，改写、改编以及摘抄本书的内容

目录

Nginx 开发从入门到精通.....	1
缘起.....	1
版权申明.....	3
目录.....	4
团队成员.....	9
上篇：nginx 模块开发篇.....	10
初探 nginx 架构(100%).....	10
nginx 基础概念(100%).....	15
connection.....	15
request.....	17
基本数据结构(99%).....	25
ngx_str_t(100%).....	25
ngx_pool_t(100%).....	30
ngx_array_t(100%).....	34
ngx_hash_t(100%).....	35
ngx_hash_wildcard_t(100%).....	37
ngx_hash_combined_t(100%).....	39
ngx_hash_keys_arrays_t(100%).....	40
ngx_chain_t(100%).....	42
ngx_buf_t(99%).....	43
ngx_list_t(100%).....	46
ngx_queue_t(100%).....	47
nginx 的配置系统(100%).....	50
指令概述.....	50
指令参数.....	50
指令上下文.....	51
nginx 的模块化体系结构.....	54
模块概述.....	54
模块的分类.....	54
nginx 的请求处理.....	55
请求的处理流程.....	55
handler 模块(100%).....	58
handler 模块简介.....	58
模块的基本结构.....	59
模块配置结构.....	59
模块配置指令.....	59
模块上下文结构.....	64
模块的定义.....	66
handler 模块的基本结构.....	67
handler 模块的挂载.....	68
按处理阶段挂载.....	68
按需挂载.....	69
handler 的编写步骤.....	70
示例: hello handler 模块.....	71
handler 模块的编译和使用.....	77
config 文件的编写.....	77

编译	77
使用	77
更多 handler 模块示例分析	78
http access module	78
http static module	79
http log module	85
过滤模块 (90%)	86
过滤模块简介 (90%)	86
执行时间和内容 (90%)	87
执行顺序 (90%)	87
模块编译 (90%)	88
过滤模块的分析 (90%)	89
相关结构体 (90%)	89
响应头过滤函数 (90%)	91
响应体过滤函数 (90%)	92
upstream 模块	95
upstream 模块 (100%)	95
upstream 模块接口	95
memcached 模块分析	96
本节回顾	100
负载均衡模块 (100%)	100
配置	100
指令	101
钩子	102
初始化配置	104
初始化请求	104
peer.get 和 peer.free 回调函数	105
本节回顾	106
其他模块 (40%)	106
core 模块 (40%)	107
Nginx 的启动模块 (40%)	107
event 模块 (40%)	107
event 的类型和功能 (40%)	107
accept 锁 (40%)	108
定时器 (40%)	110
mail 模块	110
mail 模块的实现	110
mail 模块的功能	110
模块开发高级篇(30%)	110
变量(80%)	110
综述	111
创建变量	112
使用变量	113
upstream	115
使用 subrequest 访问 upstream	115
超越 upstream	115
使用 lua 模块	115
下篇: nginx 原理解析篇	116
高性能服务器设计	116

c10k 问题	116
高性能服务器编写的关键原则	116
事件驱动的核心引擎.....	116
定时器管理	116
nginx 架构详解(50%)	116
nginx 的源码目录结构(100%)	116
nginx 的 configure 原理(100%).....	117
auto 脚本	117
模块编译顺序.....	132
nginx 的事件机制	134
event 框架及非阻塞模型	134
定时器实现	134
信号处理	134
惊群问题	134
nginx 的进程机制	134
master 进程.....	134
worker 进程	134
进程间通讯	134
nginx 基础设施.....	134
内存池	134
简介:.....	134
结构:.....	135
实现:.....	136
共享内存	139
slab 算法	139
buffer 管理.....	139
buffer 重用机制	139
buffer 防拷贝机制	139
chain 管理	139
chain 重用机制	139
aio 原理.....	139
锁实现	139
基本数据结构.....	139
时间缓存	139
文件缓存	139
log 机制.....	140
nginx 的启动阶段 (30%)	140
概述 (100%)	140
共有流程 (100%).....	140
配置解析接口 (100%).....	141
配置解析	143
通用过程 (100%).....	143
http 的解析	147
server 的管理.....	158
location 的管理.....	159
模块初始化	159
热代码部署	159
reload 过程解析.....	159
upgrade 过程解析	159

nginx 的请求处理阶段 (90%).....	159
接收请求流程 (99%).....	159
http 请求格式简介 (99%).....	159
请求头读取 (99%).....	160
请求体读取(100%).....	180
多阶段处理请求.....	188
多阶段执行链.....	189
POST_READ 阶段.....	193
SERVER_REWRITE 阶段.....	195
FIND_CONFIG 阶段.....	196
REWRITE 阶段.....	196
POST_REWRITE 阶段.....	197
PREACCESS 阶段.....	197
ACCESS 阶段.....	197
POST_ACCESS 阶段.....	197
TRY_FILES 阶段.....	198
CONTENT 阶段.....	198
LOG 阶段.....	200
Nginx filter.....	201
header filter 分析.....	204
body filter 分析.....	205
subrequest 原理解析 (99%).....	228
https 请求处理解析.....	238
nginx 支持 ssl 简介.....	238
ssl 连接建立(ssl 握手).....	239
nginx 的 upstream 原理解析.....	246
无缓冲的处理流程.....	246
带缓冲的处理流程.....	246
subrequest 访问 upstream 的处理流程.....	247
负载均衡算法.....	247
round-robin 算法.....	247
基于 ip 的 hash 算法.....	247
upstream 缓存机制.....	247
常用 upstream 模块分析.....	247
proxy 模块.....	247
fastcgi 模块.....	247
nginx 的 script 处理.....	247
正则表达式.....	247
变量.....	247
复杂变量.....	247
if 的处理.....	247
附录 A 编码风格 (100%).....	247
Nginx 代码风格图示 (100%).....	247
附录 B 常用 API.....	253
B.1 字符串操作.....	253
B.2 读取请求体.....	253
B.3 操作请求头.....	253
B.4 取参数.....	253
B.5 URI 处理.....	253

B.6 buffer 操作	253
B.7 变量操作	253
B.8 日志输出	253
B.9 配置解析接口	253
B.10 文件操作	253
B.11 锁操作	253
附录 C 模块编译，调试与测试	254
C.1 编译与安装	254
环境要求	254
下载	254
Nginx 在 Windows 环境下安装	255
Nginx 在 Linux 环境下安装	255
测试	259
Nginx 在 Windows 环境下查看 nginx 进程	260
Nginx 在 Linux 环境下查看 nginx 进程	260
C.2 调试日志	260
一，开启调试日志：	260
二，日志级别分析：	261
三，日志格式设置：	264
四，调试日志的几个注意点：	264
C.3 使用 GDB 调试	265
C.4 功能测试	265
C.5 性能/压力测试	265
C.6 常见缺陷分析	265

团队成员

叔度 (<http://blog.zhuzhaoyuan.com>)

雕梁 (<http://www.pagefault.info>)

文景 (<http://yaoweibin.cn>)

李子 (<http://blog.lifeibo.com>)

卫越 (<http://blog.sina.com.cn/u/1929617884>)

袁茁 (<http://yzprofile.me>)

小熊 (<http://dinic.iteye.com>)

吉兆 (<http://jizhao.blog.chinaunix.net>)

静龙 (http://blog.csdn.net/fengmo_q)

竹叔 (<http://weibo.com/u/2199139545>)

公远 (<http://100continue.iteye.com/>)

布可 (<http://weibo.com/sifeierss>)

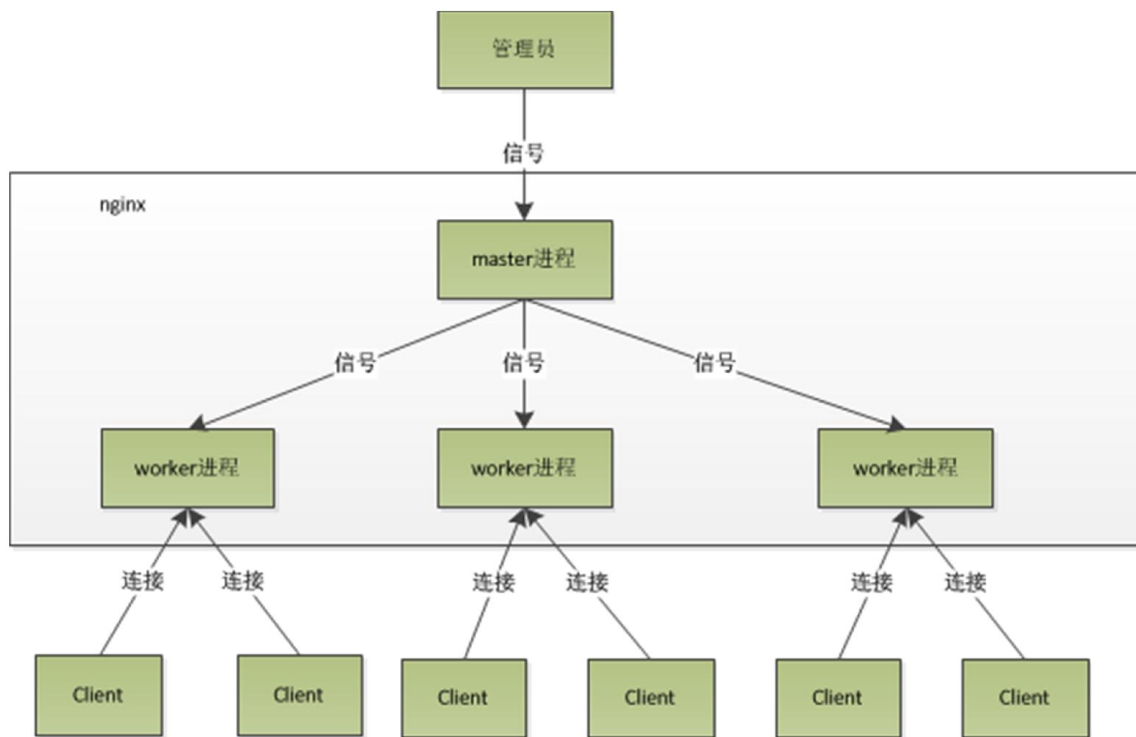
上篇：nginx 模块开发篇

初探 nginx 架构(100%)

众所周知，nginx 性能高，而 nginx 的高性能与其架构是分不开的。那么 nginx 究竟是怎么样的呢？这一节我们先来初识一下 nginx 框架吧。

nginx 在启动后，在 unix 系统中会以 daemon 的方式在后台运行，后台进程包含一个 master 进程和多个 worker 进程。我们也可以手动地关掉后台模式，让 nginx 在前台运行，并且通过配置让 nginx 取消 master 进程，从而可以使 nginx 以单进程方式运行。很显然，生产环境下我们肯定不会这么做，所以关闭后台模式，一般是用来调试用的，在后面的章节里面，我们会详细地讲解如何调试 nginx。所以，我们可以看到，nginx 是以多进程的方式来工作的，当然 nginx 也是支持多线程的方式的，只是我们主流的方式还是多进程的方式，也是 nginx 的默认方式。nginx 采用多进程的方式有诸多好处，所以我就主要讲解 nginx 的多进程模式吧。

刚才讲到，nginx 在启动后，会有一个 master 进程和多个 worker 进程。master 进程主要用来管理 worker 进程，包含：接收来自外界的信号，向各 worker 进程发送信号，监控 worker 进程的运行状态，当 worker 进程退出后(异常情况下)，会自动重新启动新的 worker 进程。而基本的网络事件，则是放在 worker 进程中来处理了。多个 worker 进程之间是对等的，他们同等竞争来自客户端的请求，各进程互相之间是独立的。一个请求，只可能在一个 worker 进程中处理，一个 worker 进程，不可能处理其它进程的请求。worker 进程的个数是可以设置的，一般我们会设置与机器 cpu 核数一致，这里面的原因与 nginx 的进程模型以及事件处理模型是分不开的。nginx 的进程模型，可以由下图来表示：



在 nginx 启动后，如果我们要操作 nginx，要怎么做呢？从上文中我们可以看到，master 来管理 worker 进程，所以我们只需要与 master 进程通信就行了。master 进程会接收来自外界发来的信号，再根据信号做不同的事情。所以我们要控制 nginx，只需要通过 kill 向 master 进程发送信号就行了。比如 kill -HUP pid，则是告诉 nginx，从容地重启 nginx，我们一般用这个信号来重启 nginx，或重新加载配置，因为是从容地重启，因此服务是不中断的。master 进程在接收到 HUP 信号后是怎么做的呢？首先 master 进程在接到信号后，会先重新加载配置文件，然后再启动新的 worker 进程，并向所有老的 worker 进程发送信号，告诉他们可以光荣退休了。新的 worker 在启动后，就开始接收新的请求，而老的 worker 在收到来自 master 的信号后，就不再接收新的请求，并且在当前进程中的所有未处理完的请求处理完成后，再退出。当然，直接给 master 进程发送信号，这是比较老的操作方式，nginx 在 0.8 版本之后，引入了一系列命令行参数，来方便我们管理。比如，./nginx -s reload，就是来重启 nginx，./nginx -s stop，就是来停止 nginx 的运行。如何做到的呢？我们还是拿 reload 来说，我们看到，执行命令时，我们是启动一个新的 nginx 进程，而新的 nginx 进程在解析到 reload 参数后，就知道我们的目的是控制 nginx 来重新加载配置文件了，它会向 master 进程发送信号，然后接下来的动作，就和我们直接向 master 进程发送信号一样了。

现在，我们知道了当我们在操作 `nginx` 的时候，`nginx` 内部做了些什么事情，那么，`worker` 进程又是如何处理请求的呢？我们前面有提到，`worker` 进程之间是平等的，每个进程，处理请求的机会也是一样的。当我们提供 80 端口的 `http` 服务时，一个连接请求过来，每个进程都有可能处理这个连接，怎么做到的呢？首先，每个 `worker` 进程都是从 `master` 进程 `fork` 过来，在 `master` 进程里面，先建立好需要 `listen` 的 `socket` (`listenfd`) 之后，然后再 `fork` 出多个 `worker` 进程。所有 `worker` 进程的 `listenfd` 会在新连接到来时变得可读，为保证只有一个进程处理该连接，所有 `worker` 进程在注册 `listenfd` 读事件前抢 `accept_mutex`，抢到互斥锁的那个进程注册 `listenfd` 读事件，在读事件里调用 `accept` 接受该连接。当一个 `worker` 进程在 `accept` 这个连接之后，就开始读取请求，解析请求，处理请求，产生数据后，再返回给客户端，最后才断开连接，这样一个完整的请求就是这样的了。我们可以看到，一个请求，完全由 `worker` 进程来处理，而且只在一个 `worker` 进程中处理。

那么，`nginx` 采用这种进程模型有什么好处呢？当然，好处肯定会很多了。首先，对于每个 `worker` 进程来说，独立的进程，不需要加锁，所以省掉了锁带来的开销，同时在编程以及问题查找时，也会方便很多。其次，采用独立的进程，可以让互相之间不会互相影响，一个进程退出后，其它进程还在工作，服务不会中断，`master` 进程则很快启动新的 `worker` 进程。当然，`worker` 进程的异常退出，肯定是程序有 `bug` 了，异常退出，会导致当前 `worker` 上的所有请求失败，不过不会影响到所有请求，所以降低了风险。当然，好处还有很多，大家可以慢慢体会。

上面讲了很多关于 `nginx` 的进程模型，接下来，我们来看看 `nginx` 是如何处理事件的。

有人可能要问了，`nginx` 采用多 `worker` 的方式来处理请求，每个 `worker` 里面只有一个主线程，那能够处理的并发数很有限啊，多少个 `worker` 就能处理多少个并发，何来高并发呢？非也，这就是 `nginx` 的高明之处，`nginx` 采用了异步非阻塞的方式来处理请求，也就是说，`nginx` 是可以同时处理成千上万个请求的。想想 `apache` 的常用工作方式（`apache` 也有异步非阻塞版本，但因其与自带某些模块冲突，所以不常用），每个请求会独占一个工作线程，当并发数上到几千时，就同时有几千的线程在处理请求了。这对操作系统来说，是个不小的挑战，线程带来的内存占用非常大，线程的上下文切换带来的 `cpu` 开销很大，自然性能就上不去了，而这些开销完全是没有意义的。

为什么 `nginx` 可以采用异步非阻塞的方式来处理呢，或者异步非阻塞到底是怎么回事呢？我们先回到原点，看看一个请求的完整过程。首先，请求过来，要建立连接，然后再接收数据，接收数据后，再发送数据。具体到系统底层，就是读写事件，而当读写事件没有准备好时，必然不可操作，如果不用

非阻塞的方式来调用，那就得阻塞调用了，事件没有准备好，那就只能等了，等事件准备好了，你再继续吧。阻塞调用会进入内核等待，cpu 就会让出去给别人用了，对单线程的 worker 来说，显然不合适，当网络事件越多时，大家都在等待呢，cpu 空闲下来没人用，cpu 利用率自然上不去了，更别谈高并发了。好吧，你说加进程数，这跟 apache 的线程模型有什么区别，注意，别增加无谓的上下文切换。

所以，在 nginx 里面，最忌讳阻塞的系统调用了。不要阻塞，那就非阻塞喽。非阻塞就是，事件没有准备好，马上返回 EAGAIN，告诉你，事件还没准备好呢，你慌什么，过会再来吧。好吧，你过一会，再来检查一下事件，直到事件准备好了为止，在这期间，你就可以先去做其它事情，然后再来看看事件好了没。虽然不阻塞了，但你得不时地过来检查一下事件的状态，你可以做更多的事情了，但带来的开销也是不小的。所以，才有了异步非阻塞的事件处理机制，具体到系统调用就是像

select/poll/epoll/kqueue 这样的系统调用。它们提供了一种机制，让你可以同时监控多个事件，调用他们是阻塞的，但可以设置超时时间，在超时时间之内，如果有事件准备好了，就返回。这种机制正好解决了我们上面的两个问题，拿 epoll 为例(在后面的例子中，我们多以 epoll 为例子，以代表这一类函数)，当事件没准备好时，放到 epoll 里面，事件准备好了，我们就去读写，当读写返回 EAGAIN 时，

我们将它再次加入到 epoll 里面。这样，只要有事件准备好了，我们就去处理它，只有当所有事件都没准备好时，才在 epoll 里面等着。这样，我们就可以并发处理大量的并发了，当然，这里的并发请求，是指未处理完的请求，线程只有一个，所以同时能处理的请求当然只有一个了，只是在请求间进行不断地切换而已，切换也是因为异步事件未准备好，而主动让出的。这里的切换是没有任何代价，你可以理解为循环处理多个准备好的事件，事实上就是这样的。与多线程相比，这种事件处理方式是有很大优势的，不需要创建线程，每个请求占用的内存也很少，没有上下文切换，事件处理非常的轻量级。并发数再多也不会导致无谓的资源浪费（上下文切换）。更多的并发数，只是会占用更多的内存而已。我之前有对连接数进行过测试，在 24G 内存的机器上，处理的并发请求数达到过 200 万。现在的网络服务器基本都采用这种方式，这也是 nginx 性能高效的主要原因。

我们之前说过，推荐设置 worker 的个数为 cpu 的核数，在这里就很容易理解了，更多的 worker 数，只会导致进程来竞争 cpu 资源了，从而带来不必要的上下文切换。而且，nginx 为了更好的利用多核特性，提供了 cpu 亲缘性的绑定选项，我们可以将某一个进程绑定在某一个核上，这样就不会因为进程的切换带来 cache 的失效。像这种小的优化在 nginx 中非常常见，同时也说明了 nginx 作者的苦心孤诣。比如，nginx 在做 4 个字节的字符串比较时，会将 4 个字符转换成一个 int 型，再作比较，以减少 cpu 的指令数等等。

现在，知道了 nginx 为什么会选择这样的进程模型与事件模型了。对于一个基本的 web 服务器来说，事件通常有三种类型，网络事件、信号、定时器。从上面的讲解中知道，网络事件通过异步非阻塞可以很好的解决掉。如何处理信号与定时器？

首先，信号的处理。对 nginx 来说，有一些特定的信号，代表着特定的意义。信号会中断掉程序当前的运行，在改变状态后，继续执行。如果是系统调用，则可能会导致系统调用的失败，需要重入。关于信号的处理，大家可以学习一些专业书籍，这里不多说。对于 nginx 来说，如果 nginx 正在等待事件（epoll_wait 时），如果程序收到信号，在信号处理函数处理完后，epoll_wait 会返回错误，然后程序可再次进入 epoll_wait 调用。

另外，再来看看定时器。由于 epoll_wait 等函数在调用的时候是可以设置一个超时时间的，所以 nginx 借助这个超时时间来实现定时器。nginx 里面的定时器事件是放在一颗维护定时器的红黑树里面，每次在进入 epoll_wait 前，先从该红黑树里面拿到所有定时器事件的最小时间，在计算出 epoll_wait 的超时时间后进入 epoll_wait。所以，当没有事件产生，也没有中断信号时，epoll_wait 会超时，也就是说，定时器事件到了。这时，nginx 会检查所有的超时事件，将他们的状态设置为超时，然后再去处理网络事件。由此可以看出，当我们写 nginx 代码时，在处理网络事件的回调函数时，通常做的第一个事情就是判断超时，然后再去处理网络事件。

我们可以用一段伪代码来总结一下 nginx 的事件处理模型：

```
while (true) {
    for t in run_tasks:
        t.handler();
    update_time(&now);
    timeout = ETERNITY;
    for t in wait_tasks: /* sorted already */
        if (t.time <= now) {
            t.timeout_handler();
        } else {
            timeout = t.time - now;
            break;
        }
    nevents = poll_function(events, timeout);
    for i in nevents:
        task t;
        if (events[i].type == READ) {
            t.handler = read_handler;
        } else { /* events[i].type == WRITE */
```

```
        t.handler = write_handler;
    }
    run_tasks_add(t);
}
```

好，本节我们讲了进程模型，事件模型，包括网络事件，信号，定时器事件。

nginx 基础概念(100%)

connection

在 nginx 中 connection 就是对 tcp 连接的封装，其中包括连接的 socket，读事件，写事件。利用 nginx 封装的 connection，我们可以很方便的使用 nginx 来处理与连接相关的事情，比如，建立连接，发送与接受数据等。而 nginx 中的 http 请求的处理就是建立在 connection 之上的，所以 nginx 不仅可以作为一个 web 服务器，也可以作为邮件服务器。当然，利用 nginx 提供的 connection，我们可以与任何后端服务打交道。

结合一个 tcp 连接的生命周期，我们看看 nginx 是如何处理一个连接的。首先，nginx 在启动时，会解析配置文件，得到需要监听的端口与 ip 地址，然后在 nginx 的 master 进程里面，先初始化好这个监控的 socket(创建 socket，设置 addrreuse 等选项，绑定到指定的 ip 地址端口，再 listen)，然后再 fork 出多个子进程出来，然后子进程会竞争 accept 新的连接。此时，客户端就可以向 nginx 发起连接了。当客户端与服务端通过三次握手建立好一个连接后，nginx 的某一个子进程会 accept 成功，得到这个建立好的连接的 socket，然后创建 nginx 对连接的封装，即 ngx_connection_t 结构体。接着，设置读写事件处理函数并添加读写事件来与客户端进行数据的交换。最后，nginx 或客户端来主动关掉连接，到此，一个连接就寿终正寝了。

当然，nginx 也是可以作为客户端来请求其它 server 的数据的(如 upstream 模块)，此时，与其它 server 创建的连接，也封装在 ngx_connection_t 中。作为客户端，nginx 先获取一个 ngx_connection_t 结构体，然后创建 socket，并设置 socket 的属性（比如非阻塞）。然后再通过添加读写事件，调用 connect/read/write 来调用连接，最后关掉连接，并释放 ngx_connection_t。

在 nginx 中，每个进程会有一个连接数的最大上限，这个上限与系统对 fd 的限制不一样。在操作系统中，通过 `ulimit -n`，我们可以得到一个进程所能够打开的 fd 的最大数，即 `nofile`，因为每个 socket 连接会占用掉一个 fd，所以这也会限制我们进程的最大连接数，当然也会直接影响到我们程序所能支持的最大并发数，当 fd 用完后，再创建 socket 时，就会失败。nginx 通过设置 `worker_connections` 来设置每个进程支持的最大连接数。如果该值大于 `nofile`，那么实际的最大连接数是 `nofile`，nginx 会有警告。nginx 在实现时，是通过一个连接池来管理的，每个 worker 进程都有一个独立的连接池，连接池的大小是 `worker_connections`。这里的连接池里面保存的其实不是真实的连接，它只是一个 `worker_connections` 大小的一个 `ngx_connection_t` 结构的数组。并且，nginx 会通过一个链表 `free_connections` 来保存所有的空闲 `ngx_connection_t`，每次获取一个连接时，就从空闲连接链表中获取一个，用完后，再放回空闲连接链表里面。

在这里，很多人会误解 `worker_connections` 这个参数的意思，认为这个值就是 nginx 所能建立连接的最大值。其实不然，这个值是表示每个 worker 进程所能建立连接的最大值，所以，一个 nginx 能建立的最大连接数，应该是 `worker_connections * worker_processes`。当然，这里说的是最大连接数，对于 HTTP 请求本地资源来说，能够支持的最大并发数量是 `worker_connections * worker_processes`，而如果是 HTTP 作为反向代理来说，最大并发数量应该是 `worker_connections * worker_processes/2`。因为作为反向代理服务器，每个并发会建立与客户端的连接和与后端服务的连接，会占用两个连接。那么，我们前面有说过一个客户端连接过来后，多个空闲的进程，会竞争这个连接，很容易看到，这种竞争会导致不公平，如果某个进程得到 `accept` 的机会比较多，它的空闲连接很快就用完了，如果不提前做一些控制，当 `accept` 到一个新的 tcp 连接后，因为无法得到空闲连接，而且无法将此连接转交给其它进程，最终会导致此 tcp 连接得不到处理，就中止掉了。很显然，这是不公平的，有的进程有空余连接，却没有处理机会，有的进程因为没有空余连接，却人为地丢弃连接。那么，如何解决这个问题呢？首先，nginx 的处理得先打开 `accept_mutex` 选项，此时，只有获得了 `accept_mutex` 的进程才会去添加 `accept` 事件，也就是说，nginx 会控制进程是否添加 `accept` 事件。nginx 使用一个叫 `ngx_accept_disabled` 的变量来控制是否去竞争 `accept_mutex` 锁。在第一段代码中，计算 `ngx_accept_disabled` 的值，这个值是 nginx 单进程的所有连接总数的八分之一，减去剩下的空闲连接数量，得到的这个 `ngx_accept_disabled` 有一个规律，当剩余连接数小于总连接数的八分之一时，其值

才大于 0，而且剩余的连接数越小，这个值越大。再看第二段代码，当 ngx_accept_disabled 大于 0 时，不会去尝试获取 accept_mutex 锁，并且将 ngx_accept_disabled 减 1，于是，每次执行到此处时，都会去减 1，直到小于 0。不去获取 accept_mutex 锁，就是等于让出获取连接的机会，很显然可以看出，当空余连接越少时，ngx_accept_disable 越大，于是让出的机会就越多，这样其它进程获取锁的机会也就越大。不去 accept，自己的连接就控制下来了，其它进程的连接池就会得到利用，这样，nginx 就控制了多进程间连接的平衡了。

```
ngx_accept_disabled = ngx_cycle->connection_n / 8
    - ngx_cycle->free_connection_n;

if (ngx_accept_disabled > 0) {
    ngx_accept_disabled--;
} else {
    if (ngx_trylock_accept_mutex(cycle) == NGX_ERROR) {
        return;
    }

    if (ngx_accept_mutex_held) {
        flags |= NGX_POST_EVENTS;
    } else {
        if (timer == NGX_TIMER_INFINITE
            || timer > ngx_accept_mutex_delay)
        {
            timer = ngx_accept_mutex_delay;
        }
    }
}
```

好了，连接就先介绍到这，本章的目的是介绍基本概念，知道在 nginx 中连接是个什么东西就行了，而且连接是属于比较高级的用法，在后面的模块开发高级篇会有专门的章节来讲解连接与事件的实现及使用。

request

这节我们讲 request，在 nginx 中我们指的是 http 请求，具体到 nginx 中的数据结构是 ngx_http_request_t。

ngx_http_request_t 是对一个 http 请求的封装。我们知道，一个 http 请求，包含请求行、请求头、请求体、响应行、响应头、响应体。

http 请求是典型的请求-响应类型的网络协议，而 http 是文件协议，所以我们在分析请求行与请求头，以及输出响应行与响应头，往往是一行一行的进行处理。如果我们自己来写一个 http 服务器，通常在一个连接建立好后，客户端会发送请求过来。然后我们读取一行数据，分析出请求行中包含的 method、uri、http_version 信息。然后再一行一行处理请求头，并根据请求 method 与请求头的信息来决定是否有请求体以及请求体的长度，然后再去读取请求体。得到请求后，我们处理请求产生需要输出的数据，然后再生成响应行，响应头以及响应体。在将响应发送给客户端之后，一个完整的请求就处理完了。当然这是最简单的 webserver 的处理方式，其实 nginx 也是这样做的，只是有一些小小的区别，比如，当请求头读取完成后，就开始进行请求的处理了。nginx 通过 ngx_http_request_t 来保存解析请求与输出响应相关的数据。

那接下来，简要讲讲 nginx 是如何处理一个完整的请求的。对于 nginx 来说，一个请求是从 ngx_http_init_request 开始的，在这个函数中，会设置读事件为 ngx_http_process_request_line，也就是说，接下来的网络事件，会由 ngx_http_process_request_line 来执行。从 ngx_http_process_request_line 的函数名，我们可以看到，这就是来处理请求行的，正好与之前讲的，处理请求的第一件事就是处理请求行是一致的。通过 ngx_http_read_request_header 来读取请求数据。然后调用 ngx_http_parse_request_line 函数来解析请求行。nginx 为提高效率，采用状态机来解析请求行，而且在进行 method 的比较时，没有直接使用字符串比较，而是将四个字符转换成一个整型，然后一次比较以减少 cpu 的指令数，这个前面有说过。很多人可能很清楚一个请求行包含请求的方法，uri，版本，却不知道其实在请求行中，也是可以包含有 host 的。比如一个请求 GET <http://www.taobao.com/uri> HTTP/1.0 这样一个请求行也是合法的，而且 host 是 www.taobao.com，这个时候，nginx 会忽略请求头中的 host 域，而以请求行中的这个为准来查找虚拟主机。另外，对于 http0.9 版来说，是不支持请求头的，所以这里也是要特别的处理。所以，在后面解析请求头时，协议版本都是 1.0 或 1.1。整个请求行解析到的参数，会保存到 ngx_http_request_t 结构当中。

在解析完请求行后，nginx 会设置读事件的 handler 为 ngx_http_process_request_headers，然后后续的请求就在 ngx_http_process_request_headers 中进行读取与解析。ngx_http_process_request_headers 函数用来读取请求头，跟请求行一样，还是调用 ngx_http_read_request_header 来读取请求头，调用

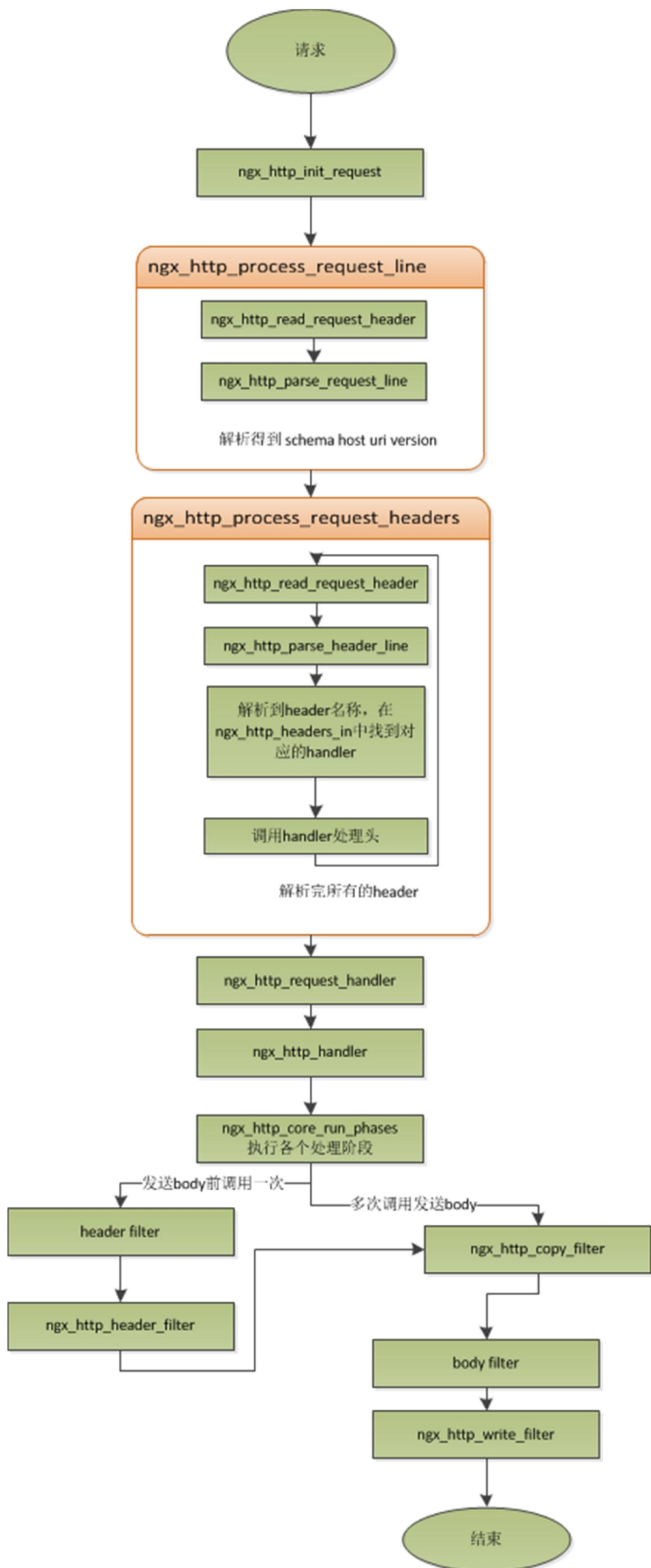
`ngx_http_parse_header_line` 来解析一行请求头，解析到的请求头会保存到 `ngx_http_request_t` 的域 `headers_in` 中，`headers_in` 是一个链表结构，保存所有的请求头。而 HTTP 中有些请求是需要特别处理的，这些请求头与请求处理函数存放在一个映射表里面，即 `ngx_http_headers_in`，在初始化时，会生成一个 hash 表，当每解析到一个请求头后，就会先在这个 hash 表中查找，如果有找到，则调用相应的处理函数来处理这个请求头。比如:Host 头的处理函数是 `ngx_http_process_host`。

当 nginx 解析到两个回车换行符时，就表示请求头的结束，此时就会调用 `ngx_http_process_request` 来处理请求了。`ngx_http_process_request` 会设置当前的连接的读写事件处理函数为 `ngx_http_request_handler`，然后再调用 `ngx_http_handler` 来真正开始处理一个完整的 http 请求。这里可能比较奇怪，读写事件处理函数都是 `ngx_http_request_handler`，其实在这个函数中，会根据当前事件是读事件还是写事件，分别调用 `ngx_http_request_t` 中的 `read_event_handler` 或者是 `write_event_handler`。由于此时，我们的请求头已经读取完成了，之前有说过，nginx 的做法是先不读取请求 body，所以这里面我们设置 `read_event_handler` 为 `ngx_http_block_reading`，即不读取数据了。刚才说到，真正开始处理数据，是在 `ngx_http_handler` 这个函数里面，这个函数会设置 `write_event_handler` 为 `ngx_http_core_run_phases`，并执行 `ngx_http_core_run_phases` 函数。`ngx_http_core_run_phases` 这个函数将执行多阶段请求处理，nginx 将一个 http 请求的处理分为多个阶段，那么这个函数就是执行这些阶段来产生数据。因为 `ngx_http_core_run_phases` 最后会产生数据，所以我们就很容易理解，为什么设置写事件的处理函数为 `ngx_http_core_run_phases` 了。在这里，我简要说明了一下函数的调用逻辑，我们需要明白最终是调用 `ngx_http_core_run_phases` 来处理请求，产生的响应头会放在 `ngx_http_request_t` 的 `headers_out` 中，这一部分内容，我会放在请求处理流程里面去讲。nginx 的各种阶段会对请求进行处理，最后会调用 `filter` 来过滤数据，对数据进行加工，如 `truncked` 传输、`gzip` 压缩等。这里的 `filter` 包括 `header filter` 与 `body filter`，即对响应头或响应体进行处理。`filter` 是一个链表结构，分别有 `header filter` 与 `body filter`，先执行 `header filter` 中的所有 `filter`，然后再执行 `body filter` 中的所有 `filter`。在 `header filter` 中的最后一个 `filter`，即 `ngx_http_header_filter`，这个 `filter` 将会遍历所有的响应头，最后需要输出的响应头在一个连续的内存，然后调用

ngx_http_write_filter 进行输出。ngx_http_write_filter 是 body filter 中的最后一个，所以 nginx 首先的 body 信息，在经过一系列的 body filter 之后，最后也会调用 ngx_http_write_filter 来进行输出(有图来说明)。

这里要注意的是，nginx 会将整个请求头都放在一个 buffer 里面，这个 buffer 的大小通过配置项 client_header_buffer_size 来设置，如果用户的请求头太大，这个 buffer 装不下，那 nginx 就会重新分配一个新的更大的 buffer 来装请求头，这个大 buffer 可以通过 large_client_header_buffers 来设置，这个 large_buffer 这一组 buffer，比如配置 4 8k，就是表示有四个 8k 大小的 buffer 可以用。注意，为了保存请求行或请求头的完整性，一个完整的请求行或请求头，需要放在一个连续的内存里面，所以，一个完整的请求行或请求头，只会保存在一个 buffer 里面。这样，如果请求行大于一个 buffer 的大小，就会返回 414 错误，如果一个请求头大小大于一个 buffer 大小，就会返回 400 错误。在了解了这些参数的值，以及 nginx 实际的做法之后，在应用场景，我们就需要根据实际的需求来调整这些参数，来优化我们的程序了。

处理流程图：



以上这些，就是 nginx 中一个 http 请求的生命周期了。我们再看看与请求相关的一些概念吧。

keepalive

当然，在 nginx 中，对于 http1.0 与 http1.1 也是支持长连接的。什么是长连接呢？我们知道，http 请求是基于 TCP 协议之上的，那么，当客户端在发起请求前，需要先与服务端建立 TCP 连接，而每一次的 TCP 连接是需要三次握手来确定的，如果客户端与服务端之间网络差一点，这三次交互消费的时间会比较多，而且三次交互也会带来网络流量。当然，当连接断开后，也会有四次的交互，当然对用户体验来说就不重要了。而 http 请求是请求应答式的，如果我们能知道每个请求头与响应体的长度，那么我们是可以在一个连接上面执行多个请求的，这就是所谓的长连接，但前提条件是我们先得确定请求头与响应体的长度。对于请求来说，如果当前请求需要有 body，如 POST 请求，那么 nginx 就需要客户端在请求头中指定 content-length 来表明 body 的大小，否则返回 400 错误。也就是说，请求体的长度是确定的，那么响应体的长度呢？先来看看 http 协议中关于响应 body 长度的确定：

1. 对于 http1.0 协议来说，如果响应头中有 content-length 头，则以 content-length 的长度就可以知道 body 的长度了，客户端在接收 body 时，就可以依照这个长度来接收数据，接收完后，就表示这个请求完成了。而如果没有 content-length 头，则客户端会一直接收数据，直到服务端主动断开连接，才表示 body 接收完了。
2. 而对于 http1.1 协议来说，如果响应头中的 Transfer-encoding 为 chunked 传输，则表示 body 是流式输出，body 会被分成多个块，每块的开始会标识出当前块的长度，此时，body 不需要通过长度来指定。如果是非 chunked 传输，而且有 content-length，则按照 content-length 来接收数据。否则，如果是非 chunked，并且没有 content-length，则客户端接收数据，直到服务端主动断开连接。

从上面，我们可以看到，除了 http1.0 不带 content-length 以及 http1.1 非 chunked 不带 content-length 外，body 的长度是可知的。此时，当服务端在输出完 body 之后，会可以考虑使用长连接。能否使用长连接，也是有条件限制的。如果客户端的请求头中的 connection 为 close，则表示客户端需要关掉长连接，如果为 keep-alive，则客户端需要打开长连接，如果客户端的请求中没有 connection 这个头，

那么根据协议，如果是 http1.0，则默认为 close，如果是 http1.1，则默认为 keep-alive。如果结果为 keepalive，那么，nginx 在输出完响应体后，会设置当前连接的 keepalive 属性，然后等待客户端下一次请求。当然，nginx 不可能一直等待下去，如果客户端一直不发数据过来，岂不是一直占用这个连接？所以当 nginx 设置了 keepalive 等待下一次的请求时，同时也会设置一个最大等待时间，这个时间是通过选项 keepalive_timeout 来配置的，如果配置为 0，则表示关掉 keepalive，此时，http 版本无论是 1.1 还是 1.0，客户端的 connection 不管是 close 还是 keepalive，都会强制为 close。

如果服务端最后的决定是 keepalive 打开，那么在响应的 http 头里面，也会包含有 connection 头域，其值是“Keep-Alive”，否则就是“Close”。如果 connection 值为 close，那么在 nginx 响应完数据后，会主动关掉连接。所以，对于请求量比较大的 nginx 来说，关掉 keepalive 最后会产生比较多的 time-wait 状态的 socket。一般来说，当客户端的一次访问，需要多次访问同一个 server 时，打开 keepalive 的优势非常大，比如图片服务器，通常一个网页会包含很多个图片。打开 keepalive 也会大量减少 time-wait 的数量。

pipe

在 http1.1 中，引入了一种新的特性，即 pipeline。那么什么是 pipeline 呢？pipeline 其实就是流水线作业，它可以看作为 keepalive 的一种升华，因为 pipeline 也是基于长连接的，目的就是利用一个连接做多次请求。如果客户端要提交多个请求，对于 keepalive 来说，那么第二个请求，必须要等到第一个请求的响应接收完全后，才能发起，这和 TCP 的停止等待协议是一样的，得到两个响应的的时间至少为 $2*RTT$ 。而对 pipeline 来说，客户端不必等到第一个请求处理完后，就可以马上发起第二个请求。得到两个响应的的时间可能能够达到 $1*RTT$ 。nginx 是直接支持 pipeline 的，但是，nginx 对 pipeline 中的多个请求的处理却不是并行的，依然是一个请求接一个请求的处理，只是在处理第一个请求的时候，客户端就可以发起第二个请求。这样，nginx 利用 pipeline 减少了处理完一个请求后，等待第二个请求的请求头数据的时间。其实 nginx 的做法很简单，前面说到，nginx 在读取数据时，会将读取的数据放到一个 buffer 里面，所以，如果 nginx 在处理完前一个请求后，如果发现 buffer 里面还有数据，就认为剩下的数据是下一个请求的开始，然后就接下来处理下一个请求，否则就设置 keepalive。

lingering_close

lingering_close，字面意思就是延迟关闭，也就是说，当 nginx 要关闭连接时，并非立即关闭连接，而是先关闭 tcp 连接的写，再等待一段时间后再关掉连接的读。为什么要这样呢？我们先来看看这样一个场景。nginx 在接收客户端的请求时，可能由于客户端或服务端出错了，要立即响应错误信息给客户端，而 nginx 在响应错误信息后，大部分情况下是需要关闭当前连接。nginx 执行完 write() 系统调用把错误信息发送给客户端，write() 系统调用返回成功并不表示数据已经发送到客户端，有可能还在 tcp 连接的 write buffer 里。接着如果直接执行 close() 系统调用关闭 tcp 连接，内核会首先检查 tcp 的 read buffer 里有没有客户端发送过来的数据留在内核态没有被用户态进程读取，如果有则发送给客户端 RST 报文来关闭 tcp 连接丢弃 write buffer 里的数据，如果没有则等待 write buffer 里的数据发送完毕，然后再经过正常的 4 次分手报文断开连接。所以，当在某些场景下出现 tcp write buffer 里的数据在 write() 系统调用之后到 close() 系统调用执行之前没有发送完毕，且 tcp read buffer 里面还有数据没有读，close() 系统调用会导致客户端收到 RST 报文且不会拿到服务端发送过来的错误信息数据。那客户端肯定会想，这服务器好霸道，动不动就 reset 我的连接，连个错误信息都没有。

在上面这个场景中，我们可以看到，关键点是服务端给客户端发送了 RST 包，导致自己发送的数据在客户端忽略掉了。所以，解决问题的重点是，让服务端别发 RST 包。再想想，我们发送 RST 是因为我们关掉了连接，关掉连接是因为我们不想再处理此连接了，也不会有任何数据产生了。对于全双工的 TCP 连接来说，我们只需要关掉写就行了，读可以继续进行，我们只需要丢掉读到的任何数据就行了，这样的话，当我们关掉连接后，客户端再发过来的数据，就不会再收到 RST 了。当然最终我们还是需要关掉这个读端的，所以我们会设置一个超时时间，在这个时间过后，就关掉读，客户端再发送数据来就不管了，作为服务端我会认为，都这么长时间了，发给你的错误信息也应该读到了，再慢就不关我事了，要怪就怪你 RP 不好了。当然，正常的客户端，在读取到数据后，会关掉连接，此时服务端就会在超时时间内关掉读端。这些正是 lingering_close 所做的事情。协议栈提供 SO_LINGER 这个选项，它的一种配置情况就是来处理 lingering_close 的情况的，不过 nginx 是自己实现的 lingering_close。lingering_close 存在的意义就是来读取剩下的客户端发来的数据，所以 nginx 会有一个读超时时间，通过 lingering_timeout 选项来设置，如果在 lingering_timeout 时间内还没有收到数据，则直接关掉连接。

nginx 还支持设置一个总的读取时间，通过 `lingering_time` 来设置，这个时间也就是 nginx 在关闭写之后，保留 socket 的时间，客户端需要在这个时间内发送完所有的数据，否则 nginx 在这个时间过后，会直接关掉连接。当然，nginx 是支持配置是否打开 `lingering_close` 选项的，通过 `lingering_close` 选项来配置。那么，我们在实际应用中，是否应该打开 `lingering_close` 呢？这个就没有固定的推荐值了，如 Maxim Dounin 所说，`lingering_close` 的主要作用是保持更好的客户端兼容性，但是却需要消耗更多的额外资源（比如连接会一直占着）。

这节，我们介绍了 nginx 中，连接与请求的基本概念，下节，我们讲基本的数据结构。

基本数据结构(99%)

nginx 的作者为追求极致的高效，自己实现了很多颇具特色的 nginx 风格的数据结构以及公共函数。比如，nginx 提供了带长度的字符串，根据编译器选项优化过的字符串拷贝函数 `ngx_copy` 等。所以，在我们写 nginx 模块时，应该尽量调用 nginx 提供的 api，尽管有些 api 只是对 glibc 的宏定义。本节，我们介绍 `string`、`list`、`buffer`、`chain` 等一系列最基本的数据结构及相关 api 的使用技巧以及注意事项。

ngx_str_t(100%)

在 nginx 源码目录的 `src/core` 下面的 `ngx_string.h` 里面，包含了字符串的封装以及字符串相关操作的 api。nginx 提供了一个带长度的字符串结构 `ngx_str_t`，它的原型如下：

```
typedef struct {
    size_t    len;
    u_char    *data;
} ngx_str_t;
```

在结构体当中，`data` 指向字符串数据的第一个字符，字符串的结束用长度来表示，而不是由 `'\0'` 来表示结束。所以，在写 nginx 代码时，处理字符串的方法跟我们平时使用有很大的不一样，但要时刻记住，字符串不以 `'\0'` 结束，尽量使用 nginx 提供的字符串操作的 api 来操作字符串。那么，nginx 这样做有什么好处呢？首先，通过长度来表示字符串长度，减少计算字符串长度的次数。其次，nginx 可以重复引用一段字符串内存，`data` 可以指向任意内存，长度表示结束，而不用去 copy 一份自己的字符串(因

为如果要以'\0'结束，而不能更改原字符串，所以势必要 copy 一段字符串)。我们在 ngx_http_request_t 结构体的成员中，可以找到很多字符串引用一段内存的例子，比如 request_line、uri、args 等等，这些字符串的 data 部分，都是指向在接收数据时创建 buffer 所指向的内存中，uri，args 就没有必要 copy 一份出来。这样的话，减少了很多不必要的内存分配与拷贝。正是基于此特性，在 nginx 中，必须谨慎的去修改一个字符串。在修改字符串时需要认真的去考虑：是否可以修改该字符串；字符串修改后，是否会对其它的引用造成影响。在后面介绍 ngx_unescape_uri 函数的时候，就会看到这一点。但是，使用 nginx 的字符串会产生一些问题，glibc 提供的很多系统 api 函数大多是通过'\0'来表示字符串的结束，所以我们在调用系统 api 时，就不能直接传入 str->data 了。此时，通常的做法是创建一段 str->len + 1 大小的内存，然后 copy 字符串，最后一个字节置为'\0'。比较 hack 的做法是，将字符串最后一个字符的后一个字符 backup 一个，然后设置为'\0'，在做完调用后，再由 backup 改回来，但前提条件是，你得确定这个字符是可以修改的，而且是有内存分配，不会越界，但一般不建议这么做。接下来，看看 nginx 提供的操作字符串相关的 api。

```
#define ngx_string(str)      { sizeof(str) - 1, (u_char *) str }
```

ngx_string(str)是一个宏，它通过一个以'\0'结尾的普通字符串 str 构造一个 nginx 的字符串，鉴于其中采用 sizeof 操作符计算字符串长度，因此参数必须是一个常量字符串。

```
#define ngx_null_string      { 0, NULL }
```

定义变量时，使用 ngx_null_string 初始化字符串为空字符串，字符串的长度为 0，data 为 NULL。

```
#define ngx_str_set(str, text) \
    (str)->len = sizeof(text) - 1; (str)->data = (u_char *) text
```

ngx_str_set 用于设置字符串 str 为 text，由于使用 sizeof 计算长度，故 text 必须为常量字符串。

```
#define ngx_str_null(str)    (str)->len = 0; (str)->data = NULL
```

ngx_str_null 用于设置字符串 str 为空串，长度为 0，data 为 NULL。

上面这四个函数，使用时一定要小心，ngx_string 与 ngx_null_string 是“{, }”格式的，故只能用于赋值时初始化，如：

```
ngx_str_t str = ngx_string("hello world");
ngx_str_t str1 = ngx_null_string;
```

如果向下面这样使用，就会有问题，这里涉及到 c 语言中对结构体变量赋值操作的语法规则，在此不做介绍。

```
ngx_str_t str, str1;
str = ngx_string("hello world");    // 编译出错
str1 = ngx_null_string;             // 编译出错
```

这种情况，可以调用 ngx_str_set 与 ngx_str_null 这两个函数来做：

```
ngx_str_t str, str1;
ngx_str_set(&str, "hello world");
ngx_str_null(&str1);
```

按照 C99 标准，您也可以这么做：

```
ngx_str_t str, str1;
str = (ngx_str_t) ngx_string("hello world");
str1 = (ngx_str_t) ngx_null_string;
```

另外要注意的是，ngx_string 与 ngx_str_set 在调用时，传进去的字符串一定是常量字符串，否则会得到意想不到的错误(因为 ngx_str_set 内部使用了 sizeof()，如果传入的是 u_char*，那么计算的是这个指针的长度，而不是字符串的长度)。如：

```
ngx_str_t str;
u_char *a = "hello world";
ngx_str_set(&str, a);    // 问题产生
```

此外，值得注意的是，由于 ngx_str_set 与 ngx_str_null 实际上是两行语句，故在 if/for/while 等语句中单独使用需要用花括号括起来，例如：

```
ngx_str_t str;
if (cond)
    ngx_str_set(&str, "true");    // 问题产生
else
```

```
ngx_str_set(&str, "false");    // 问题产生
```

```
void ngx_strlow(u_char *dst, u_char *src, size_t n);
```

将 src 的前 n 个字符转换成小写存放在 dst 字符串当中，调用者需要保证 dst 指向的空间大于等于 n，且指向的空间必须可写。操作不会对原字符串产生变动。如要更改原字符串，可以：

```
ngx_strlow(str->data, str->data, str->len);  
ngx_strncmp(s1, s2, n)
```

区分大小写的字符串比较，只比较前 n 个字符。

```
ngx_strcmp(s1, s2)
```

区分大小写的不带长度的字符串比较。

```
ngx_int_t ngx_strcasecmp(u_char *s1, u_char *s2);
```

不区分大小写的不带长度的字符串比较。

```
ngx_int_t ngx_strncasecmp(u_char *s1, u_char *s2, size_t n);
```

不区分大小写的带长度的字符串比较，只比较前 n 个字符。

```
u_char * ngx_cdecl ngx_sprintf(u_char *buf, const char *fmt, ...);  
u_char * ngx_cdecl ngx_snprintf(u_char *buf, size_t max, const char *fmt, ...);  
u_char * ngx_cdecl ngx_slprintf(u_char *buf, u_char *last, const char *fmt, ...);
```

上面这三个函数用于字符串格式化，ngx_snprintf 的第二个参数 max 指明 buf 的空间大小，ngx_slprintf 则通过 last 来指明 buf 空间的大小。推荐使用第二个或第三个函数来格式化字符串，ngx_sprintf 函数还是比较危险的，容易产生缓冲区溢出漏洞。在这一系列函数中，nginx 在兼容 glibc 中格式化字符串的形式之外，还添加了一些方便格式化 nginx 类型的一些转义字符，比如 %V 用于格式化 ngx_str_t 结构。在 nginx 源文件的 ngx_string.c 中有说明：

```
/*  
 * supported formats:  
 *   %[0][width][x][X]O      off_t  
 *   %[0][width]T            time_t  
 *   %[0][width][u][x|X]z    ssize_t/size_t  
 *   %[0][width][u][x|X]d    int/u_int  
 *   %[0][width][u][x|X]l    long
```

```

*   %[0][width|m][u][x|X]i    ngx_int_t/ngx_uint_t
*   %[0][width][u][x|X]D      int32_t/uint32_t
*   %[0][width][u][x|X]L      int64_t/uint64_t
*   %[0][width|m][u][x|X]A    ngx_atomic_int_t/ngx_atomic_uint_t
*   %[0][width][.width]f      double, max valid number fits to %18.15f
*   %P                          ngx_pid_t
*   %M                          ngx_msec_t
*   %r                          rlim_t
*   %p                          void *
*   %V                          ngx_str_t *
*   %v                          ngx_variable_value_t *
*   %s                          null-terminated string
*   %*s                          length and string
*   %Z                          '\0'
*   %N                          '\n'
*   %c                          char
*   %%                          %
*
*   reserved:
*   %t                          ptrdiff_t
*   %S                          null-terminated wchar string
*   %C                          wchar
*/

```

这里特别要提醒的是，我们最常用于格式化 `ngx_str_t` 结构，其对应的转义符是 `%V`，传给函数的一定要是指针类型，否则程序就会 `coredump` 掉。这也是我们最容易犯的错。比如：

```

ngx_str_t str = ngx_string("hello world");
char buffer[1024];
ngx_snprintf(buffer, 1024, "%V", &str);    // 注意，str 取地址

void ngx_encode_base64(ngx_str_t *dst, ngx_str_t *src);
ngx_int_t ngx_decode_base64(ngx_str_t *dst, ngx_str_t *src);

```

这两个函数用于对 `str` 进行 `base64` 编码与解码，调用前，需要保证 `dst` 中有足够的空间来存放结果，如果不知道具体大小，可先调用 `ngx_base64_encoded_length` 与 `ngx_base64_decoded_length` 来预估最大占用空间。

```

uintptr_t ngx_escape_uri(u_char *dst, u_char *src, size_t size,
    ngx_uint_t type);

```

对 `src` 进行编码，根据 `type` 来按不同的方式进行编码，如果 `dst` 为 `NULL`，则返回需要转义的字符的数量，由此可得到需要的空间大小。`type` 的类型可以是：

```
#define NGX_ESCAPE_URI          0
#define NGX_ESCAPE_ARGS        1
#define NGX_ESCAPE_HTML        2
#define NGX_ESCAPE_REFRESH      3
#define NGX_ESCAPE_MEMCACHED    4
#define NGX_ESCAPE_MAIL_AUTH    5

void ngx_unescape_uri(u_char **dst, u_char **src, size_t size, ngx_uint_t type);
```

对 src 进行反编码，type 可以是 0、NGX_UNESCAPE_URI、NGX_UNESCAPE_REDIRECT 这三个值。如果是 0，则表示 src 中的所有字符都要进行转码。如果是 NGX_UNESCAPE_URI 与 NGX_UNESCAPE_REDIRECT，则遇到‘?’后就结束了，后面的字符就不管了。而 NGX_UNESCAPE_URI 与 NGX_UNESCAPE_REDIRECT 之间的区别是 NGX_UNESCAPE_URI 对于遇到的需要转码的字符，都会转码，而 NGX_UNESCAPE_REDIRECT 则只会对非可见字符进行转码。

```
uintptr_t ngx_escape_html(u_char *dst, u_char *src, size_t size);
```

对 html 标签进行编码。

当然，我这里只介绍了一些常用的 api 的使用，大家可以先熟悉一下，在实际使用过程中，遇到不明白的，最快最直接的方法就是去看源码，看 api 的实现或看 nginx 自身调用 api 的地方是怎么做的，代码就是最好的文档。

ngx_pool_t(100%)

ngx_pool_t 是一个非常重要的数据结构，在很多重要的场合都有使用，很多重要的数据结构也都在使用它。那么它究竟是一个什么东西呢？简单的说，它提供了一种机制，帮助管理一系列的资源（如内存，文件等），使得对这些资源的使用和释放统一进行，免除了使用过程中考虑到对各种各样资源的什么时候释放，是否遗漏了释放的担心。

例如对于内存的管理，如果我们需要使用内存，那么总是从一个 ngx_pool_t 的对象中获取内存，在最终的某个时刻，我们销毁这个 ngx_pool_t 对象，所有这些内存都被释放了。这样我们就不必要对对这些内存进行 malloc 和 free 的操作，不用担心是否某块被 malloc 出来的内存没有被释放。因为当 ngx_pool_t 对象被销毁的时候，所有从这个对象中分配出来的内存都会被统一释放掉。

再比如我们要使用一系列的文件，但是我们打开以后，最终需要都关闭，那么我们就把这些文件统一登记到一个 ngx_pool_t 对象中，当这个 ngx_pool_t 对象被销毁的时候，所有这些文件都将会被关闭。

从上面举的两个例子中我们可以看出，使用 ngx_pool_t 这个数据结构的时候，所有的资源的释放都在这个对象被销毁的时刻，统一进行了释放，那么就会带来一个问题，就是这些资源的生存周期（或者说被占用的时间）是跟 ngx_pool_t 的生存周期基本一致（ngx_pool_t 也提供了少量操作可以提前释放资源）。从最高效的角度来说，这并不是最好的。比如，我们需要依次使用 A，B，C 三个资源，且使用完 B 的时候，A 就不会再被使用了，使用 C 的时候 A 和 B 都不会被使用到。如果不使用 ngx_pool_t 来管理这三个资源，那我们可能从系统里面申请 A，使用 A，然后在释放 A。接着申请 B，使用 B，再释放 B。最后申请 C，使用 C，然后释放 C。但是当我们使用一个 ngx_pool_t 对象来管理这三个资源的时候，A，B 和 C 的释放是在最后一起发生的，也就是在使用完 C 以后。诚然，这在客观上增加了程序在一段时间的资源使用量。但是这也减轻了程序员分别管理三个资源的生命周期的工作。这也就是有所得，必有所失的道理。实际上是一个取舍的问题，要看在具体的情况下，你更在乎的是哪个。

可以看一下在 nginx 里面一个典型的使用 ngx_pool_t 的场景，对于 nginx 处理的每个 http request, nginx 会生成一个 ngx_pool_t 对象与这个 http request 关联，所有处理过程中需要申请的资源都从这个 ngx_pool_t 对象中获取，当这个 http request 处理完成以后，所有在处理过程中申请的资源，都将随着这个关联的 ngx_pool_t 对象的销毁而释放。

ngx_pool_t 相关结构及操作被定义在文件 src/core/nginx_palloc.h|c 中。

```
typedef struct ngx_pool_s      ngx_pool_t;

struct ngx_pool_s {
    ngx_pool_data_t      d;
    size_t                max;
    ngx_pool_t            *current;
    ngx_chain_t            *chain;
    ngx_pool_large_t       *large;
    ngx_pool_cleanup_t     *cleanup;
    ngx_log_t              *log;
};
```

从 ngx_pool_t 的一般使用者的角度来说，可不用关注 ngx_pool_t 结构中各字段作用。所以这里也不会进行详细的解释，当然在说明某些操作函数的使用的时候，如有必要，会进行说明。

下面我们来分别解释下 ngx_pool_t 的相关操作。

```
ngx_pool_t *ngx_create_pool(size_t size, ngx_log_t *log);
```

创建一个初始节点大小为 size 的 pool，log 为后续在该 pool 上进行操作时输出日志的对象。需要说明的是 size 的选择，size 的大小必须小于等于 NGX_MAX_ALLOC_FROM_POOL，且必须大于 sizeof(ngx_pool_t)。

选择大于 NGX_MAX_ALLOC_FROM_POOL 的值会造成浪费，因为大于该限制的空间不会被用到（只是说在第一个由 ngx_pool_t 对象管理的内存块上的内存，后续的分配如果第一个内存块上的空闲部分已用完，会再分配的）。

选择小于 sizeof(ngx_pool_t) 的值会造成程序崩溃。由于初始大小的内存块中要用一部分来存储 ngx_pool_t 这个信息本身。

当一个 ngx_pool_t 对象被创建以后，该对象的 max 字段被赋值为 size-sizeof(ngx_pool_t) 和

NGX_MAX_ALLOC_FROM_POOL 这两者中比较小的。后续的从这个 pool 中分配的内存块，在第一块内存使用完成以后，如果要继续分配的话，就需要继续从操作系统申请内存。当内存的大小小于等于 max 字段的时候，则分配新的内存块，链接在 d 这个字段（实际上是 d.next 字段）管理的一条链表上。

当要分配的内存块是比 max 大的，那么从系统中申请的内存是被挂接在 large 字段管理的一条链表上。我们暂且把这个称之为大块内存链和小块内存链。

```
void *ngx_palloc(ngx_pool_t *pool, size_t size);
```

从这个 pool 中分配一块为 size 大小的内存。注意，此函数分配的内存的起始地址按照 NGX_ALIGNMENT 进行了对齐。对齐操作会提高系统处理的速度，但会造成少量内存的浪费。

```
void *ngx_pnalloc(ngx_pool_t *pool, size_t size);
```

从这个 pool 中分配一块为 size 大小的内存。但是此函数分配的内存并没有像上面的函数那样进行过对齐。

```
void *ngx_pcalloc(ngx_pool_t *pool, size_t size);
```

该函数也是分配 size 大小的内存，并且对分配的内存块进行了清零。内部实际上是转调用 ngx_palloc 实现的。

```
void *ngx_pmemalign(ngx_pool_t *pool, size_t size, size_t alignment);
```

按照指定对齐大小 `alignment` 来申请一块大小为 `size` 的内存。此处获取的内存不管大小都将被置于大内存块链中管理。

```
ngx_int_t ngx_pfree(ngx_pool_t *pool, void *p);
```

对于被置于大块内存链，也就是被 `large` 字段管理的一系列内存中的某块进行释放。该函数的实现是顺序遍历 `large` 管理的大块内存链表。所以效率比较低下。如果在这个链表中找到了这块内存，则释放，并返回 `NGX_OK`。否则返回 `NGX_DECLINED`。

由于这个操作效率比较低下，除非必要，也就是说这块内存非常大，确应及时释放，否则一般不需要调用。反正内存存在这个 `pool` 被销毁的时候，总归会都释放掉的嘛！

```
ngx_pool_cleanup_t *ngx_pool_cleanup_add(ngx_pool_t *p, size_t size);
```

`ngx_pool_t` 中的 `cleanup` 字段管理着一个特殊的链表，该链表的每一项都记录着一个特殊的需要释放的资源。对于这个链表中每个节点所包含的资源如何去释放，是自说明的。这也就提供了非常大的灵活性。意味着，`ngx_pool_t` 不仅仅可以管理内存，通过这个机制，也可以管理任何需要释放的资源，例如，关闭文件，或者删除文件等等。下面我们看一下这个链表每个节点的类型：

```
typedef struct ngx_pool_cleanup_s  ngx_pool_cleanup_t;
typedef void (*ngx_pool_cleanup_pt)(void *data);

struct ngx_pool_cleanup_s {
    ngx_pool_cleanup_pt  handler;
    void                 *data;
    ngx_pool_cleanup_t   *next;
};
```

data: 指明了该节点所对应的资源。

handler: 是一个函数指针，指向一个可以释放 `data` 所对应资源的函数。该函数只有一个参数，就是 `data`。

next: 指向该链表中下一个元素。

看到这里，`ngx_pool_cleanup_add` 这个函数的用法，我相信大家都应该有一些明白了。但是这个参数 `size` 是起什么作用的呢？这个 `size` 就是要存储这个 `data` 字段所指向的资源的大小，该函数会为 `data` 分配 `size` 大小的空间。

比如我们需要最后删除一个文件。那我们在调用这个函数的时候，把 `size` 指定为存储文件名的字符串的大小，然后调用这个函数给 `cleanup` 链表中增加一项。该函数会返回新添加的这个节点。我们然后把这个节点中的 `data` 字段拷贝为文件名。把 `handler` 字段赋值为一个删除文件的函数（当然该函数的原型要按照 `void (*ngx_pool_cleanup_pt)(void *data)`）。

```
void ngx_destroy_pool(ngx_pool_t *pool);
```

该函数就是释放 `pool` 中持有的所有内存，以及依次调用 `cleanup` 字段所管理的链表中每个元素的 `handler` 字段所指向的函数，来释放掉所有该 `pool` 管理的资源。并且把 `pool` 指向的 `ngx_pool_t` 也释放掉了，完全不可用了。

```
void ngx_reset_pool(ngx_pool_t *pool);
```

该函数释放 `pool` 中所有大块内存链表上的内存，小块内存链上的内存块都修改为可用。但是不会去处理 `cleanup` 链表上的项目。

ngx_array_t(100%)

`ngx_array_t` 是 `nginx` 内部使用的数组结构。`nginx` 的数组结构在存储上与大家认知的 C 语言内置的数组有相似性，比如实际上存储数据的区域也是一大块连续的内存。但是数组除了存储数据的内存以外还包含一些元信息来描述相关的一些信息。下面我们从数组的定义上来详细的了解一下。`ngx_array_t` 的定义位于 `src/core/ngx_array.c|h` 里面。

```
typedef struct ngx_array_s      ngx_array_t;
struct ngx_array_s {
    void      *elts;
    ngx_uint_t nelts;
    size_t     size;
    ngx_uint_t nalloc;
    ngx_pool_t *pool;
};
```

elts: 指向实际的数据存储区域。

nelts: 数组实际元素个数。

size: 数组单个元素的大小，单位是字节。

数组的容量。表示该数组在不引发扩容的前提下，可以最多存储的元素的个数。当 `nelts` 增

nalloc: 长到达 `nalloc` 时，如果再往此数组中存储元素，则会引发数组的扩容。数组的容量将会扩展到原有容量的 2 倍大小。实际上是分配新的一块内存，新的一块内存的大小是原有内存大

小的 2 倍。原有的数据会被拷贝到新的一块内存中。

pool: 该数组用来分配内存的内存池。

下面介绍 ngx_array_t 相关操作函数。

```
ngx_array_t *ngx_array_create(ngx_pool_t *p, ngx_uint_t n, size_t size);
```

创建一个新的数组对象，并返回这个对象。

p: 数组分配内存使用的内存池；

n: 数组的初始容量大小，即在不扩容的情况下最多可以容纳的元素个数。

size: 单个元素的大小，单位是字节。

```
void ngx_array_destroy(ngx_array_t *a);
```

销毁该数组对象，并释放其分配的内存回内存池。

```
void *ngx_array_push(ngx_array_t *a);
```

在数组 a 上新追加一个元素，并返回指向新元素的指针。需要把返回的指针使用类型转换，转换为具体的类型，然后再给新元素本身或者是各字段（如果数组的元素是复杂类型）赋值。

```
void *ngx_array_push_n(ngx_array_t *a, ngx_uint_t n);
```

在数组 a 上追加 n 个元素，并返回指向这些追加元素的首个元素的位置的指针。

```
static ngx_inline ngx_int_t ngx_array_init(ngx_array_t *array, ngx_pool_t *pool,
ngx_uint_t n, size_t size);
```

如果一个数组对象是被分配在堆上的，那么当调用 ngx_array_destroy 销毁以后，如果想再次使用，就可以调用此函数。

如果一个数组对象是被分配在栈上的，那么就需要调用此函数，进行初始化的工作以后，才可以使用。

注意事项: 由于使用 ngx_palloc 分配内存，数组在扩容时，旧的内存不会被释放，会造成内存的浪费。因此，最好能提前规划好数组的容量，在创建或者初始化的时候一次搞定，避免多次扩容，造成内存浪费。

ngx_hash_t(100%)

ngx_hash_t 是 nginx 自己的 hash 表的实现。定义和实现位于 src/core/nginx_hash.h 中。ngx_hash_t

的实现也与数据结构教科书上所描述的 hash 表的实现是大同小异。对于常用的解决冲突的方法有线性

探测，二次探测和开链法等。ngx_hash_t 使用的是最常用的一种，也就是开链法，这也是 STL 中的 hash 表使用的方法。

但是 ngx_hash_t 的实现又有其几个显著的特点:

1. ngx_hash_t 不像其他的 hash 表的实现，可以插入删除元素，它只能一次初始化，就构建起整个 hash 表以后，既不能再删除，也不能在插入元素了。
2. ngx_hash_t 的开链并不是真的开了一个链表，实际上是开了一段连续的存储空间，几乎可以看做是一个数组。这是因为 ngx_hash_t 在初始化的时候，会经历一次预计算的过程，提前把每个桶里面会有多少元素放进去给计算出来，这样就提前知道每个桶的大小了。那么就不需要使用链表，一段连续的存储空间就足够了。这也从一定程度上节省了内存的使用。

从上面的描述，我们可以看出来，这个值越大，越造成内存的浪费。就两步，首先是初始化，然后就可以在里面进行查找了。下面我们详细来看一下。

ngx_hash_t 的初始化。

```
ngx_int_t ngx_hash_init(ngx_hash_init_t *hinit, ngx_hash_key_t *names,
ngx_uint_t nelts);
```

首先我们来看一下初始化函数。该函数的第一个参数 hinit 是初始化的一些参数的一个集合。 names 是初始化一个 ngx_hash_t 所需要的所有 key 的一个数组。而 nelts 就是 key 的个数。下面先看一下 ngx_hash_init_t 类型，该类型提供了初始化一个 hash 表所需要的一些基本信息。

```
typedef struct {
    ngx_hash_t      *hash;
    ngx_hash_key_pt  key;

    ngx_uint_t       max_size;
    ngx_uint_t       bucket_size;

    char             *name;
    ngx_pool_t       *pool;
    ngx_pool_t       *temp_pool;
} ngx_hash_init_t;
```

hash:	该字段如果为 NULL，那么调用完初始化函数后，该字段指向新创建出来的 hash 表。如果该字段不为 NULL，那么在初始的时候，所有的数据被插入了这个字段所指的 hash 表中。
key:	指向从字符串生成 hash 值的 hash 函数。nginx 的源代码中提供了默认的实现函数 ngx_hash_key_lc。

max_size: hash 表中的桶的个数。该字段越大，元素存储时冲突的可能性越小，每个桶中存储的元素会更少，则查询起来的速度更快。当然，这个值越大，越造成内存的浪费也越大，(实际上也浪费不了多少)。

bucket_size: 每个桶的最大限制大小，单位是字节。如果在初始化一个 hash 表的时候，发现某个桶里面无法存的下所有属于该桶的元素，则 hash 表初始化失败。

name: 该 hash 表的名字。

pool: 该 hash 表分配内存使用的 pool。

temp_pool: 该 hash 表使用的临时 pool，在初始化完成以后，该 pool 可以被释放和销毁掉。

下面来看一下存储 hash 表 key 的数组的结构。

```
typedef struct {  
    ngx_str_t      key;  
    ngx_uint_t     key_hash;  
    void           *value;  
} ngx_hash_key_t;
```

key 和 value 的含义显而易见，就不用解释了。key_hash 是对 key 使用 hash 函数计算出来的值。对这两个结构分析完成以后，我想大家应该都已经明白这个函数应该是如何使用了。该函数成功初始化一个 hash 表以后，返回 NGX_OK，否则返回 NGX_ERROR。

```
void *ngx_hash_find(ngx_hash_t *hash, ngx_uint_t key, u_char *name, size_t len);
```

在 hash 里面查找 key 对应的 value。实际上这里的 key 是对真正的 key (也就是 name) 计算出的 hash 值。len 是 name 的长度。

如果查找成功，则返回指向 value 的指针，否则返回 NULL。

ngx_hash_wildcard_t(100%)

nginx 为了处理带有通配符的域名的匹配问题，实现了 ngx_hash_wildcard_t 这样的 hash 表。他可以支持两种类型的带有通配符的域名。一种是通配符在前的，例如：“*.abc.com”，也可以省略掉星号，直接写成“.abc.com”。这样的 key，可以匹配 www.abc.com，qqq.www.abc.com 之类的。另外一种是通过通配符在末尾的，例如：“mail.xxx.*”，请特别注意通配符在末尾的不像位于开始的通配符可以被省略掉。这样的通配符，可以匹配 mail.xxx.com、mail.xxx.com.cn、mail.xxx.net 之类的域名。

有一点必须说明，就是一个 `ngx_hash_wildcard_t` 类型的 hash 表只能包含通配符在前的 key 或者是通配符在后的 key。不能同时包含两种类型的通配符的 key。`ngx_hash_wildcard_t` 类型变量的构建是通过函数 `ngx_hash_wildcard_init` 完成的，而查询是通过函数 `ngx_hash_find_wc_head` 或者 `ngx_hash_find_wc_tail` 来做的。`ngx_hash_find_wc_head` 是查询包含通配符在前的 key 的 hash 表的，而 `ngx_hash_find_wc_tail` 是查询包含通配符在后的 key 的 hash 表的。

下面详细说明这几个函数的用法。

```
ngx_int_t ngx_hash_wildcard_init(ngx_hash_init_t *hinit, ngx_hash_key_t *names,
    ngx_uint_t nelts);
```

该函数用来构建一个可以包含通配符 key 的 hash 表。

- hinit:** 构造一个通配符 hash 表的一些参数的一个集合。关于该参数对应的类型的说明，请参见 `ngx_hash_t` 类型中 `ngx_hash_init` 函数的说明。
- 构造此 hash 表的所有的通配符 key 的数组。特别要注意的是这里的 key 已经都是被预处理过的。例如：“*.abc.com”或者“.abc.com”被预处理完成以后，变成了“com.abc.”。而“mail.xxx.*”则被预处理为“mail.xxx.”。为什么会被处理这样？这里不得不简单地描述一下通配符 hash 表的实现原理。当构造此类型的 hash 表的时候，实际上是构造了一个 hash 表的一个“链表”，是通过 hash 表中的 key “链接”起来的。比如：对于“*.abc.com”将会构造出 2 个 hash 表，第一个 hash 表中有一个 key 为 com 的表项，该表项的 value 包含有指向第二个 hash 表的指针，而第二个 hash 表中有一个表项 abc，该表项的 value 包含有指向 *.abc.com 对应的 value 的指针。那么查询的时候，比如查询 www.abc.com 的时候，先查 com，通过查 com 可以找到第二级的 hash 表，在第二级 hash 表中，再查找 abc，依次类推，直到在某一级的 hash 表中查到的表项对应的 value 对应一个真正的值而非一个指向下一级 hash 表的指针的时候，查询过程结束。这里有一点需要特别注意的，就是 `names` 数组中元素的 value 值低两位 bit 必须为 0（有特殊用途）。如果不满足这个条件，这个 hash 表查询不出正确结果。
- names:** 向第二个 hash 表的指针，而第二个 hash 表中有一个表项 abc，该表项的 value 包含有指向 *.abc.com 对应的 value 的指针。那么查询的时候，比如查询 www.abc.com 的时候，先查 com，通过查 com 可以找到第二级的 hash 表，在第二级 hash 表中，再查找 abc，依次类推，直到在某一级的 hash 表中查到的表项对应的 value 对应一个真正的值而非一个指向下一级 hash 表的指针的时候，查询过程结束。这里有一点需要特别注意的，就是 `names` 数组中元素的 value 值低两位 bit 必须为 0（有特殊用途）。如果不满足这个条件，这个 hash 表查询不出正确结果。
- nelts:** `names` 数组元素的个数。

该函数执行成功返回 `NGX_OK`，否则 `NGX_ERROR`。

```
void *ngx_hash_find_wc_head(ngx_hash_wildcard_t *hwc, u_char *name, size_t len);
```

该函数查询包含通配符在前的 key 的 hash 表的。

- hwc:** hash 表对象的指针。
- name:** 需要查询的域名，例如：www.abc.com。
- len:** name 的长度。

该函数返回匹配的通配符对应 value。如果没有查到，返回 `NULL`。

```
void *ngx_hash_find_wc_tail(ngx_hash_wildcard_t *hwc, u_char *name, size_t len);
```

该函数查询包含通配符在末尾的 key 的 hash 表的。 参数及返回值请参加上个函数的说明。

ngx_hash_combined_t(100%)

组合类型 hash 表，该 hash 表的定义如下：

```
typedef struct {  
    ngx_hash_t          hash;  
    ngx_hash_wildcard_t *wc_head;  
    ngx_hash_wildcard_t *wc_tail;  
} ngx_hash_combined_t;
```

从其定义显见，该类型实际上包含了三个 hash 表，一个普通 hash 表，一个包含前向通配符的 hash 表和一个包含后向通配符的 hash 表。

nginx 提供该类型的作用，在于提供一个方便的容器包含三个类型的 hash 表，当有包含通配符的和不包含通配符的一组 key 构建 hash 表以后，以一种方便的方式来查询，你不需要再考虑一个 key 到底是应该到哪个类型的 hash 表里去查了。

构造这样一组合 hash 表的时候，首先定义一个该类型的变量，再分别构造其包含的三个子 hash 表即可。

对于该类型 hash 表的查询，nginx 提供了一个方便的函数 ngx_hash_find_combined。

```
void *ngx_hash_find_combined(ngx_hash_combined_t *hash, ngx_uint_t key,  
u_char *name, size_t len);
```

该函数在此组合 hash 表中，依次查询其三个子 hash 表，看是否匹配，一旦找到，立即返回查找结果，也就是说如果有多个可能匹配，则只返回第一个匹配的结果。

hash: 此组合 hash 表对象。

key: 根据 name 计算出的 hash 值。

name: key 的具体内容。

len: name 的长度。

返回查询的结果，未查到则返回 NULL。

ngx_hash_keys_arrays_t(100%)

大家看到在构建一个 ngx_hash_wildcard_t 的时候，需要对通配符的哪些 key 进行预处理。这个处理起来比较麻烦。而当有一组 key，这些里面既有无通配符的 key，也有包含通配符的 key 的时候。我们就需要构建三个 hash 表，一个包含普通的 key 的 hash 表，一个包含前向通配符的 hash 表，一个包含后向通配符的 hash 表（或者也可以把这三个 hash 表组合成一个 ngx_hash_combined_t）。在这种情况下，为了让大家方便的构造这些 hash 表，nginx 提供给了此辅助类型。

该类型以及相关的操作函数也定义在 src/core/nginx_hash.h|c 里。我们先来看一下该类型的定义。

```
typedef struct {
    ngx_uint_t      hsize;

    ngx_pool_t      *pool;
    ngx_pool_t      *temp_pool;

    ngx_array_t      keys;
    ngx_array_t      *keys_hash;

    ngx_array_t      dns_wc_head;
    ngx_array_t      *dns_wc_head_hash;

    ngx_array_t      dns_wc_tail;
    ngx_array_t      *dns_wc_tail_hash;
} ngx_hash_keys_arrays_t;
```

- hsize:** 将要构建的 hash 表的桶的个数。对于使用这个结构中包含的信息构建的三种类型的 hash 表都会使用此参数。
- pool:** 构建这些 hash 表使用的 pool。
- temp_pool:** 在构建这个类型以及最终的三个 hash 表过程中可能用到临时 pool。该 temp_pool 可以在构建完成以后，被销毁掉。这里只是存放临时的一些内存消耗。
- keys:** 存放所有非通配符 key 的数组。
- keys_hash:** 这是个二维数组，第一个维度代表的是 bucket 的编号，那么 keys_hash[i] 中存放的是所有的 key 算出来的 hash 值对 hsize 取模以后的值为 i 的 key。假设有 3 个 key，分别是 key1, key2 和 key3 假设 hash 值算出来以后对 hsize 取模的值都是 i，那么这三个 key 的值就顺序存放在 keys_hash[i][0], keys_hash[i][1], keys_hash[i][2]。该值在调用的过程中用来保存和检测是否有冲突的 key 值，也就是是否有重复。
- dns_wc_head:** 放前向通配符 key 被处理完成以后的值。比如：“*.abc.com” 被处理完成以后，变成 “com.abc.” 被存放在此数组中。
- dns_wc_tail:** 存放后向通配符 key 被处理完成以后的值。比如：“mail.xxx.*” 被处理完成以后，变成 “mail.xxx.” 被存放在此数组中。
- dns_wc_head_hash:** 该值在调用的过程中用来保存和检测是否有冲突的前向通配符的 key 值，也就是是否

有重复。

dns_wc_tail_hash:

该值在调用的过程中用来保存和检测是否有冲突的后向通配符的 key 值，也就是是否有重复。

在定义一个这个类型的变量，并对字段 `pool` 和 `temp_pool` 赋值以后，就可以调用函数

`ngx_hash_add_key` 把所有的 key 加入到这个结构中了，该函数会自动实现普通 key，带前向通配符的 key 和带后向通配符的 key 的分类和检查，并将这些值存放到对应的字段中去，然后就可以通过检查这个结构体中的 `keys`、`dns_wc_head`、`dns_wc_tail` 三个数组是否为空，来决定是否构建普通 hash 表，前向通配符 hash 表和后向通配符 hash 表了（在构建这三个类型的 hash 表的时候，可以分别使用 `keys`、`dns_wc_head`、`dns_wc_tail` 三个数组）。

构建出这三个 hash 表以后，可以组合在一个 `ngx_hash_combined_t` 对象中，使用

`ngx_hash_find_combined` 进行查找。或者是仍然保持三个独立的变量对应这三个 hash 表，自己决定何时以及在哪个 hash 表中进行查询。

```
ngx_int_t ngx_hash_keys_array_init(ngx_hash_keys_arrays_t *ha, ngx_uint_t type);
```

初始化这个结构，主要是对这个结构中的 `ngx_array_t` 类型的字段进行初始化，成功返回 `NGX_OK`。

ha: 该结构的对象指针。

该字段有 2 个值可选择，即 `NGX_HASH_SMALL` 和 `NGX_HASH_LARGE`。用来指明将要建立的 hash

type: 表的类型，如果是 `NGX_HASH_SMALL`，则有比较小的桶的个数和数组元素大小。`NGX_HASH_LARGE` 则相反。

```
ngx_int_t ngx_hash_add_key(ngx_hash_keys_arrays_t *ha, ngx_str_t *key,
void *value, ngx_uint_t flags);
```

一般是循环调用这个函数，把一组键值对加入到这个结构体中。返回 `NGX_OK` 是加入成功。返回

`NGX_BUSY` 意味着 key 值重复。

ha: 该结构的对象指针。

key: 参数名自解释了。

value: 参数名自解释了。

flags: 有两个标志位可以设置，`NGX_HASH_WILDCARD_KEY` 和 `NGX_HASH_READONLY_KEY`。同时要设置的使用逻辑与操作符就可以了。`NGX_HASH_READONLY_KEY` 被设置的时候，在计算 hash 值的时候，key 的值不会被转成小写字符，否则会。`NGX_HASH_WILDCARD_KEY` 被设置的时候，说明 key 里面可能含有通配符，会进行相应的处理。如果两个标志位都不设置，传 0。

有关于这个数据结构的使用，可以参考 `src/http/nginx_http.c` 中的 `ngx_http_server_names` 函数。

ngx_chain_t(100%)

nginx 的 filter 模块在处理从别的 filter 模块或者是 handler 模块传递过来的数据（实际上就是需要发送给客户端的 http response）。这个传递过来的数据是以一个链表的形式(`ngx_chain_t`)。而且数据可能被分多次传递过来。也就是多次调用 filter 的处理函数，以不同的 `ngx_chain_t`。

该结构被定义在 `src/core/nginx_buf.h`。下面我们来看一下 `ngx_chain_t` 的定义。

```
typedef struct ngx_chain_s      ngx_chain_t;

struct ngx_chain_s {
    ngx_buf_t      *buf;
    ngx_chain_t    *next;
};
```

就 2 个字段，`next` 指向这个链表的下个节点。`buf` 指向实际的数据。所以在这个链表上追加节点也是非常容易，只要把末尾元素的 `next` 指针指向新的节点，把新节点的 `next` 赋值为 `NULL` 即可。

```
ngx_chain_t *ngx_alloc_chain_link(ngx_pool_t *pool);
```

该函数创建一个 `ngx_chain_t` 的对象，并返回指向对象的指针，失败返回 `NULL`。

```
#define ngx_free_chain(pool, cl)                                \
    cl->next = pool->chain;                                       \
    pool->chain = cl
```

该宏释放一个 `ngx_chain_t` 类型的对象。如果要释放整个 `chain`，则迭代此链表，对每个节点使用此宏即可。

注意: 对 `ngx_chaint_t` 类型的释放，并不是真的释放了内存，而仅仅是把这个对象挂在了这个 `pool` 对象的一个叫做 `chain` 的字段对应的 `chain` 上，以供下次从这个 `pool` 上分配 `ngx_chain_t` 类型对象的时候，快速的从这个 `pool->chain` 上取下链首元素就返回了，当然，如果这个链是空的，才会真的在这个 `pool` 上使用 `ngx_palloc` 函数进行分配。

ngx_buf_t(99%)

这个 ngx_buf_t 就是这个 ngx_chain_t 链表的每个节点的实际数据。该结构实际上是一种抽象的数据结构，它代表某种具体的数据。这个数据可能是指向内存中的某个缓冲区，也可能指向一个文件的某一部分，也可能是一些纯元数据（元数据的作用在于指示这个链表的读取者对读取的数据进行不同的处理）。

该数据结构位于 src/core/ngx_buf.h|c 文件中。我们来看一下它的定义。

```
struct ngx_buf_s {
    u_char          *pos;
    u_char          *last;
    off_t           file_pos;
    off_t           file_last;

    u_char          *start;          /* start of buffer */
    u_char          *end;            /* end of buffer */
    ngx_buf_tag_t   tag;
    ngx_file_t      *file;
    ngx_buf_t       *shadow;

    /* the buf's content could be changed */
    unsigned         temporary:1;

    /*
     * the buf's content is in a memory cache or in a read only memory
     * and must not be changed
     */
    unsigned         memory:1;

    /* the buf's content is mmap()ed and must not be changed */
    unsigned         mmap:1;

    unsigned         recycled:1;
    unsigned         in_file:1;
    unsigned         flush:1;
    unsigned         sync:1;
    unsigned         last_buf:1;
    unsigned         last_in_chain:1;

    unsigned         last_shadow:1;
    unsigned         temp_file:1;

    /* STUB */ int     num;
};
```

pos:	当 buf 所指向的数据在内存里的时候, pos 指向的是这段数据开始的位置。
last:	当 buf 所指向的数据在内存里的时候, last 指向的是这段数据结束的位置。
file_pos:	当 buf 所指向的数据是在文件里的时候, file_pos 指向的是这段数据的开始位置在文件中的偏移量。
file_last:	当 buf 所指向的数据是在文件里的时候, file_last 指向的是这段数据的结束位置在文件中的偏移量。
start:	当 buf 所指向的数据在内存里的时候, 这一整块内存包含的内容可能被包含在多个 buf 中(比如在某段数据中间插入了其他的数据, 这一块数据就需要被拆分开)。那么这些 buf 中的 start 和 end 都指向这一块内存的开始地址和结束地址。而 pos 和 last 指向本 buf 所实际包含的数据的开始和结尾。
end:	解释参见 start。
tag:	实际上是一个 void*类型的指针, 使用者可以关联任意的对象上去, 只要对使用者有意义。
file:	当 buf 所包含的内容在文件中时, file 字段指向对应的文件对象。
shadow:	当这个 buf 完整 copy 了另外一个 buf 的所有字段的时候, 那么这两个 buf 指向的实际上是同一块内存, 或者是同一个文件的同一部分, 此时这两个 buf 的 shadow 字段都是指向对方的。那么对于这样的两个 buf, 在释放的时候, 就需要使用者特别小心, 具体是由哪里释放, 要提前考虑好, 如果造成资源的多次释放, 可能会造成程序崩溃!
temporary:	为 1 时表示该 buf 所包含的内容是在一个用户创建的内存块中, 并且可以被在 filter 处理的过程中进行变更, 而不会造成问题。
memory:	为 1 时表示该 buf 所包含的内容是在内存中, 但是这些内容却不能被进行处理的 filter 进行变更。
mmap:	为 1 时表示该 buf 所包含的内容是在内存中, 是通过 mmap 使用内存映射从文件中映射到内存中的, 这些内容却不能被进行处理的 filter 进行变更。
recycled:	可以回收的。也就是这个 buf 是可以被释放的。这个字段通常是配合 shadow 字段一起使用的, 对于使用 ngx_create_temp_buf 函数创建的 buf, 并且是另外一个 buf 的 shadow, 那么可以使用这个字段来标示这个 buf 是可以被释放的。
in_file:	为 1 时表示该 buf 所包含的内容是在文件中。
flush:	遇到有 flush 字段被设置为 1 的 buf 的 chain, 则该 chain 的数据即便不是最后结束的数据 (last_buf 被设置, 标志所有要输出的内容都完了), 也会进行输出, 不会受 postpone_output 配置的限制, 但是会受到发送速率等其他条件的限制。
sync:	
last_buf:	数据被以多个 chain 传递给了过滤器, 此字段为 1 表明这是最后一个 buf。
last_in_chain:	在当前的 chain 里面, 此 buf 是最后一个。特别要注意的是 last_in_chain 的 buf 不一定是 last_buf, 但是 last_buf 的 buf 一定是 last_in_chain 的。这是因为数据会被以多个 chain 传递给某个 filter 模块。
last_shadow:	在创建一个 buf 的 shadow 的时候, 通常将新创建的一个 buf 的 last_shadow 置为 1。
temp_file:	由于受到内存使用的限制, 有时候一些 buf 的内容需要被写到磁盘上的临时文件中, 那么这时, 就设置此标志。

对于此对象的创建, 可以直接在某个 ngx_pool_t 上分配, 然后根据需要, 给对应的字段赋值。也可以使用定义好的 2 个宏:

```
#define ngx_alloc_buf(pool) ngx_palloc(pool, sizeof(ngx_buf_t))
#define ngx_calloc_buf(pool) ngx_pcalloc(pool, sizeof(ngx_buf_t))
```

这两个宏使用类似函数，也是不说自明的。

对于创建 `temporary` 字段为 1 的 `buf`（就是其内容可以被后续的 `filter` 模块进行修改），可以直接使用函数 `ngx_create_temp_buf` 进行创建。

```
ngx_buf_t *ngx_create_temp_buf(ngx_pool_t *pool, size_t size);
```

该函数创建一个 `ngx_buf_t` 类型的对象，并返回指向这个对象的指针，创建失败返回 `NULL`。

对于创建的这个对象，它的 `start` 和 `end` 指向新分配内存开始和结束的地方。`pos` 和 `last` 都指向这块新分配内存的开始处，这样，后续的操作可以在这块新分配的内存上存入数据。

pool: 分配该 `buf` 和 `buf` 使用的内存所使用的 `pool`。

size: 该 `buf` 使用的内存的大小。

为了配合对 `ngx_buf_t` 的使用，`nginx` 定义了以下的宏方便操作。

```
#define ngx_buf_in_memory(b)      (b->temporary || b->memory || b->mmap)
```

返回这个 `buf` 里面的内容是否在内存里。

```
#define ngx_buf_in_memory_only(b) (ngx_buf_in_memory(b) && !b->in_file)
```

返回这个 `buf` 里面的内容是否仅仅在内存里，并且没有在文件里。

```
#define ngx_buf_special(b) \
    ((b->flush || b->last_buf || b->sync) \
     && !ngx_buf_in_memory(b) && !b->in_file)
```

返回该 `buf` 是否是一个特殊的 `buf`，只含有特殊的标志和没有包含真正的数据。

```
#define ngx_buf_sync_only(b) \
    (b->sync \
     && !ngx_buf_in_memory(b) && !b->in_file && !b->flush && !b->last_buf)
```

返回该 `buf` 是否是一个只包含 `sync` 标志而不包含真正数据的特殊 `buf`。

```
#define ngx_buf_size(b) \
    (ngx_buf_in_memory(b) ? (off_t) (b->last - b->pos): \
     (b->file_last - b->file_pos))
```

返回该 buf 所含数据的大小，不管这个数据是在文件里还是在内存里。

ngx_list_t(100%)

ngx_list_t 顾名思义，看起来好像是一个 list 的数据结构。这样的说法，算对也不算对。因为它符合 list 类型数据结构的一些特点，比如可以添加元素，实现自增长，不会像数组类型的数据结构，受到初始设定的数组容量的限制，并且它跟我们常见的 list 型数据结构也是一样的，内部实现使用了一个链表。

那么它跟我们常见的链表实现的 list 有什么不同呢？不同点就在于它的节点，它的节点不像我们常见的 list 的节点，只能存放一个元素，ngx_list_t 的节点实际上是一个固定大小的数组。

在初始化的时候，我们需要设定元素需要占用的空间大小，每个节点数组的容量大小。在添加元素到这个 list 里面的时候，会在最尾部的节点里的数组上添加元素，如果这个节点的数组存满了，就再增加一个新的节点到这个 list 里面去。

好了，看到这里，大家应该基本上明白这个 list 结构了吧？还不明白也没有关系，下面我们来具体看一下它的定义，这些定义和相关的操作函数定义在 src/core/ngx_list.h|c 文件中。

```
typedef struct {
    ngx_list_part_t *last;
    ngx_list_part_t part;
    size_t size;
    ngx_uint_t nalloc;
    ngx_pool_t *pool;
} ngx_list_t;
```

last: 指向该链表的最后一个节点。
part: 该链表的首个存放具体元素的节点。
size: 链表中存放的具体元素所需内存大小。
nalloc: 每个节点所含的固定大小的数组的容量。
pool: 该 list 使用的分配内存的 pool。

好，我们在看一下每个节点的定义。

```
typedef struct ngx_list_part_s ngx_list_part_t;
struct ngx_list_part_s {
```

```
void          *elts;
ngx_uint_t    nelts;
ngx_list_part_t *next;
};
```

elts: 节点中存放具体元素的内存的开始地址。

nelts: 节点中已有元素个数。这个值是不能大于链表头节点 `ngx_list_t` 类型中的 `nalloc` 字段的。

next: 指向下一个节点。

我们来看一下提供的一个操作的函数。

```
ngx_list_t *ngx_list_create(ngx_pool_t *pool, ngx_uint_t n, size_t size);
```

该函数创建一个 `ngx_list_t` 类型的对象,并对该 `list` 的第一个节点分配存放元素的内存空间。

pool: 分配内存使用的 `pool`。

n: 每个节点固定长度的数组的长度。

size: 存放的具体元素的个数。

返回值: 成功返回指向创建的 `ngx_list_t` 对象的指针, 失败返回 `NULL`。

```
void *ngx_list_push(ngx_list_t *list);
```

该函数在给定的 `list` 的尾部追加一个元素, 并返回指向新元素存放空间的指针。如果追加失败, 则返回 `NULL`。

```
static ngx_inline ngx_int_t
ngx_list_init(ngx_list_t *list, ngx_pool_t *pool, ngx_uint_t n, size_t size);
```

该函数是用于 `ngx_list_t` 类型的对象已经存在, 但是其第一个节点存放元素的内存空间还未分配的情况下, 可以调用此函数来给这个 `list` 的首节点来分配存放元素的内存空间。

那么什么时候会出现已经有了 `ngx_list_t` 类型的对象, 而其首节点存放元素的内存尚未分配的情况呢?

那就是这个 `ngx_list_t` 类型的变量并不是通过调用 `ngx_list_create` 函数创建的。例如: 如果某个结构体的一个成员变量是 `ngx_list_t` 类型的, 那么当这个结构体类型的对象被创建出来的时候, 这个成员变量也被创建出来了, 但是它的首节点的存放元素的内存并未被分配。

总之, 如果这个 `ngx_list_t` 类型的变量, 如果不是你通过调用函数 `ngx_list_create` 创建的, 那么就必须调用此函数去初始化, 否则, 你往这个 `list` 里追加元素就可能引发不可预知的行为, 亦或程序会崩溃!

ngx_queue_t(100%)

ngx_queue_t 是 nginx 中的双向链表，在 nginx 源码目录 src/core 下面的 ngx_queue.h 里面。它的原型如下：

```
typedef struct ngx_queue_s ngx_queue_t;

struct ngx_queue_s {
    ngx_queue_t  *prev;
    ngx_queue_t  *next;
};
```

不同于教科书中将链表节点的数据成员声明在链表节点的结构体中，ngx_queue_t 只是声明了前向和后向指针。在使用的时候，我们首先需要定义一个哨兵节点(对于后续具体存放数据的节点，我们称之为数据节点)，比如：

```
ngx_queue_t free;
```

接下来需要进行初始化，通过宏 ngx_queue_init()来实现：

```
ngx_queue_init(&free);
```

ngx_queue_init()的宏定义如下：

```
#define ngx_queue_init(q)    \
    (q)->prev = q;          \
    (q)->next = q;
```

可见初始的时候哨兵节点的 prev 和 next 都指向自己，因此其实是一个空链表。ngx_queue_empty() 可以用来判断一个链表是否为空，其实现也很简单，就是：

```
#define ngx_queue_empty(h)   \
    (h == (h)->prev)
```

那么如何声明一个具有数据元素的链表节点呢？只要在相应的结构体中加上一个 ngx_queue_t 的成员就行了。比如 ngx_http_upstream_keepalive_module 中的 ngx_http_upstream_keepalive_cache_t：

```
typedef struct {
    ngx_http_upstream_keepalive_srv_conf_t  *conf;

    ngx_queue_t                               queue;
```

```

    ngx_connection_t          *connection;

    socklen_t                 socklen;
    u_char                    sockaddr[NGX_SOCKADDRLEN];
} ngx_http_upstream_keepalive_cache_t;

```

对于每一个这样的数据节点，可以通过 ngx_queue_insert_head()来添加到链表中，第一个参数是哨兵节点，第二个参数是数据节点，比如：

```

ngx_http_upstream_keepalive_cache_t cache;
ngx_queue_insert_head(&free, &cache.queue);

```

相应的几个宏定义如下：

```

#define ngx_queue_insert_head(h, x) \
    (x)->next = (h)->next; \
    (x)->next->prev = x; \
    (x)->prev = h; \
    (h)->next = x

#define ngx_queue_insert_after    ngx_queue_insert_head

#define ngx_queue_insert_tail(h, x) \
    (x)->prev = (h)->prev; \
    (x)->prev->next = x; \
    (x)->next = h; \
    (h)->prev = x

```

ngx_queue_insert_head()和 ngx_queue_insert_after()都是往头部添加节点，ngx_queue_insert_tail()是往尾部添加节点。从代码可以看出哨兵节点的 prev 指向链表的尾数据节点，next 指向链表的头数据节点。另外 ngx_queue_head()和 ngx_queue_last()这两个宏分别可以得到头节点和尾节点。

那假如现在有一个 ngx_queue_t *q 指向的是链表中的数据节点的 queue 成员，如何得到

ngx_http_upstream_keepalive_cache_t 的数据呢？ nginx 提供了 ngx_queue_data()宏来得到

ngx_http_upstream_keepalive_cache_t 的指针，例如：

```

ngx_http_upstream_keepalive_cache_t *cache = ngx_queue_data(q,
    ngx_http_upstream_keepalive_cache_t,
    queue);

```

也许您已经可以猜到 ngx_queue_data 是通过地址相减来得到的：

```
#define ngx_queue_data(q, type, link) \
    (type *) ((u_char *) q - offsetof(type, link))
```

另外 nginx 也提供了 ngx_queue_remove()宏来从链表中删除一个数据节点，以及 ngx_queue_add()用来将一个链表添加到另一个链表。

nginx 的配置系统(100%)

nginx 的配置系统由一个主配置文件和其他一些辅助的配置文件构成。这些配置文件均是纯文本文件，全部位于 nginx 安装目录下的 conf 目录下。

配置文件中以#开始的行，或者是前面有若干空格或者 TAB，然后再跟#的行，都被认为是注释，也就是只对编辑查看文件的用户有意义，程序在读取这些注释行的时候，其实际的内容是被忽略的。

由于除主配置文件 nginx.conf 以外的文件都是在某些情况下才使用的，而只有主配置文件是在任何情况下都被使用的。所以在这里我们就以主配置文件为例，来解释 nginx 的配置系统。

在 nginx.conf 中，包含若干配置项。每个配置项由配置指令和指令参数 2 个部分构成。指令参数也就是配置指令对应的配置值。

指令概述

配置指令是一个字符串，可以用单引号或者双引号括起来，也可以不括。但是如果配置指令包含空格，一定要引起来。

指令参数

指令的参数使用一个或者多个空格或者 TAB 字符与指令分开。指令的参数有一个或者多个 TOKEN 串组成。TOKEN 串之间由空格或者 TAB 键分隔。

TOKEN 串分为简单字符串或者是复合配置块。复合配置块即是由大括号括起来的一堆内容。一个复合配置块中可能包含若干其他的配置指令。

如果一个配置指令的参数全部由简单字符串构成，也就是不包含复合配置块，那么我们就说这个配置指令是一个简单配置项，否则称之为复杂配置项。例如下面这个是一个简单配置项：

```
error_page 500 502 503 504 /50x.html;
```

对于简单配置，配置项的结尾使用分号结束。对于复杂配置项，包含多个 **TOKEN** 串的，一般都是简单 **TOKEN** 串放在前面，复合配置块一般位于最后，而且其结尾，并不需要再添加分号。例如下面这个复杂配置项：

```
location / {  
    root    /home/jizhao/nginx-book/build/html;  
    index  index.html index.htm;  
}
```

指令上下文

nginx.conf 中的配置信息，根据其逻辑上的意义，对它们进行了分类，也就是分成了多个作用域，或者称之为配置指令上下文。不同的作用域含有一个或者多个配置项。

当前 nginx 支持的几个指令上下文：

- main:** nginx 在运行时与具体业务功能（比如 http 服务或者 email 服务代理）无关的一些参数，比如工作进程数，运行的身份等。
- http:** 与提供 http 服务相关的一些配置参数。例如：是否使用 keepalive 啊，是否使用 gzip 进行压缩等。
- server:** http 服务上支持若干虚拟主机。每个虚拟主机一个对应的 server 配置项，配置项里面包含该虚拟主机相关的配置。在提供 mail 服务的代理时，也可以建立若干 server. 每个 server 通过监听的地址来区分。
- location:** http 服务中，某些特定的 URL 对应的一系列配置项。
- mail:** 实现 email 相关的 SMTP/IMAP/POP3 代理时，共享的一些配置项（因为可能实现多个代理，工作在多个监听地址上）。

指令上下文，可能有包含的情况出现。例如：通常 http 上下文和 mail 上下文一定是出现在 main 上下文里的。在一个上下文里，可能包含另外一种类型的上下文多次。例如：如果 http 服务，支持了多个虚拟主机，那么在 http 上下文里，就会出现多个 server 上下文。

我们来看一个示例配置：

```
user  nobody;  
worker_processes  1;  
error_log  logs/error.log  info;  
  
events {  
    worker_connections  1024;  
}
```

```

http {
    server {
        listen      80;
        server_name  www.linuxidc.com;
        access_log   logs/linuxidc.access.log main;
        location / {
            index index.html;
            root /var/www/linuxidc.com/htdocs;
        }
    }

    server {
        listen      80;
        server_name  www.Androidj.com;
        access_log   logs/androidj.access.log main;
        location / {
            index index.html;
            root /var/www/androidj.com/htdocs;
        }
    }
}

mail {
    auth_http 127.0.0.1:80/auth.php;
    pop3_capabilities "TOP" "USER";
    imap_capabilities "IMAP4rev1" "UIDPLUS";

    server {
        listen      110;
        protocol     pop3;
        proxy        on;
    }
    server {
        listen      25;
        protocol     smtp;
        proxy        on;
        smtp_auth    login plain;
        xclient       off;
    }
}

```

在这个配置中，上面提到个五种配置指令上下文都存在。

存在于 main 上下文中的配置指令如下：

- user

- worker_processes
- error_log
- events
- http
- mail

存在于 http 上下文中的指令如下:

- server

存在于 mail 上下文中的指令如下:

- server
- auth_http
- imap_capabilities

存在于 server 上下文中的配置指令如下:

- listen
- server_name
- access_log
- location
- protocol
- proxy
- smtp_auth
- xclient

存在于 location 上下文中的指令如下:

- index
- root

当然，这里只是一些示例。具体有哪些配置指令，以及这些配置指令可以出现在什么样的上下文中，需要参考 nginx 的使用文档。

nginx 的模块化体系结构

nginx 的内部结构是由核心部分和一系列的功能模块所组成。这样划分是为了使得每个模块的功能相对简单，便于开发，同时也便于对系统进行功能扩展。为了便于描述，下文中我们将使用 `nginx core` 来称呼 nginx 的核心功能部分。

nginx 提供了 web 服务器的基础功能，同时提供了 web 服务反向代理，email 服务反向代理功能。`nginx core` 实现了底层的通讯协议，为其他模块和 nginx 进程构建了基本的运行时环境，并且构建了其他各模块的协作基础。除此之外，或者说大部分与协议相关的，或者应用相关的功能都是在这些模块中所实现的。

模块概述

nginx 将各功能模块组织成一条链，当有请求到达的时候，请求依次经过这条链上的部分或者全部模块，进行处理。每个模块实现特定的功能。例如，实现对请求解压缩的模块，实现 SSI 的模块，实现与上游服务器进行通讯的模块，实现与 FastCGI 服务进行通讯的模块。

有两个模块比较特殊，他们居于 `nginx core` 和各功能模块的中间。这两个模块就是 `http` 模块和 `mail` 模块。这 2 个模块在 `nginx core` 之上实现了另外一层抽象，处理与 HTTP 协议和 email 相关协议（SMTP/POP3/IMAP）有关的事件，并且确保这些事件能被以正确的顺序调用其他的一些功能模块。

目前 HTTP 协议是被实现在 `http` 模块中的，但是有可能将来被剥离到一个单独的模块中，以扩展 nginx 支持 SPDY 协议。

模块的分类

nginx 的模块根据其功能基本上可以分为以下几种类型：

event module:	搭建了独立于操作系统的事件处理机制的框架，及提供了各具体事件的处理。包括 ngx_events_module, ngx_event_core_module 和 ngx_epoll_module 等。nginx 具体使用何种事件处理模块，这依赖于具体的操作系统和编译选项。
phase handler:	此类型的模块也被直接称为 handler 模块。主要负责处理客户端请求并产生待响应内容，比如 ngx_http_static_module 模块，负责客户端的静态页面请求处理并将对应的磁盘文件准备为响应内容输出。
output filter:	也称为 filter 模块，主要是负责对输出的内容进行处理，可以对输出进行修改。例如，可以实现对输出的所有 html 页面增加预定义的 footer 一类的工作，或者对输出的图片的 URL 进行替换之类的工作。
upstream:	upstream 模块实现反向代理的功能，将真正的请求转发到后端服务器上，并从后端服务器上读取响应，发回客户端。upstream 模块是一种特殊的 handler，只不过响应内容不是真正由自己产生的，而是从后端服务器上读取的。
load-balancer:	负载均衡模块，实现特定的算法，在众多的后端服务器中，选择一个服务器出来作为某个请求的转发服务器。

nginx 的请求处理

nginx 使用一个多进程模型来对外提供服务，其中一个 master 进程，多个 worker 进程。master 进程负责管理 nginx 本身和其他 worker 进程。

所有实际上的业务处理逻辑都在 worker 进程。worker 进程中有一个函数，执行无限循环，不断处理收到的来自客户端的请求，并进行处理，直到整个 nginx 服务被停止。

worker 进程中，ngx_worker_process_cycle()函数就是这个无限循环的处理函数。在这个函数中，一个请求的简单处理流程如下：

1. 操作系统提供的机制（例如 epoll, kqueue 等）产生相关的事件。
2. 接收和处理这些事件，如是接受到数据，则产生更高层的 request 对象。
3. 处理 request 的 header 和 body。
4. 产生响应，并发送回客户端。
5. 完成 request 的处理。
6. 重新初始化定时器及其他事件。

请求的处理流程

为了让大家更好的了解 nginx 中请求处理过程，我们以 HTTP Request 为例，来做一下详细地说明。

从 nginx 的内部来看，一个 HTTP Request 的处理过程涉及到以下几个阶段。

1. 初始化 HTTP Request（读取来自客户端的数据，生成 HTTP Request 对象，该对象含有该请求所有的信息）。
2. 处理请求头。
3. 处理请求体。
4. 如果有的话，调用与此请求（URL 或者 Location）关联的 handler。
5. 依次调用各 phase handler 进行处理。

在这里，我们需要了解一下 phase handler 这个概念。phase 字面的意思，就是阶段。所以 phase handlers 也就好理解了，就是包含若干个处理阶段的一些 handler。

在每一个阶段，包含有若干个 handler，再处理到某个阶段的时候，依次调用该阶段的 handler 对 HTTP Request 进行处理。

通常情况下，一个 phase handler 对这个 request 进行处理，并产生一些输出。通常 phase handler 是与定义在配置文件中的某个 location 相关联的。

一个 phase handler 通常执行以下几项任务：

1. 获取 location 配置。
2. 产生适当的响应。
3. 发送 response header。
4. 发送 response body。

当 nginx 读取到一个 HTTP Request 的 header 的时候，nginx 首先查找与这个请求关联的虚拟主机的配置。如果找到了这个虚拟主机的配置，那么通常情况下，这个 HTTP Request 将会经过以下几个阶段的处理（phase handlers）：

NGX_HTTP_POST_READ_PHASE:

读取请求内容阶段

NGX_HTTP_SERVER_REWRITE_PHASE:

Server 请求地址重写阶段

NGX_HTTP_FIND_CONFIG_PHASE:

配置查找阶段:

NGX_HTTP_REWRITE_PHASE:

Location 请求地址重写阶段

NGX_HTTP_POST_REWRITE_PHASE:

请求地址重写提交阶段

NGX_HTTP_PREACCESS_PHASE:

访问权限检查准备阶段

NGX_HTTP_ACCESS_PHASE:

访问权限检查阶段

NGX_HTTP_POST_ACCESS_PHASE:

访问权限检查提交阶段

NGX_HTTP_TRY_FILES_PHASE:

配置项 try_files 处理阶段

NGX_HTTP_CONTENT_PHASE:

内容产生阶段

NGX_HTTP_LOG_PHASE:

日志模块处理阶段

在内容产生阶段，为了给一个 request 产生正确的响应，nginx 必须把这个 request 交给一个合适的 content handler 去处理。如果这个 request 对应的 location 在配置文件中被明确指定了一个 content handler，那么 nginx 就可以通过对 location 的匹配，直接找到这个对应的 handler，并把这个 request 交给这个 content handler 去处理。这样的配置指令包括像，perl，flv，proxy_pass，mp4 等。

如果一个 request 对应的 location 并没有直接有配置的 content handler，那么 nginx 依次尝试:

1. 如果一个 location 里面有配置 random_index on，那么随机选择一个文件，发送给客户端。
2. 如果一个 location 里面有配置 index 指令，那么发送 index 指令指明的文件，给客户端。
3. 如果一个 location 里面有配置 autoindex on，那么就发送请求地址对应的服务端路径下的文件列表给客户端。
4. 如果这个 request 对应的 location 上有设置 gzip_static on，那么就查找是否有对应的.gz 文件存在，有的话，就发送这个给客户端（客户端支持 gzip 的情况下）。
5. 请求的 URI 如果对应一个静态文件，static module 就发送静态文件的内容到客户端。

内容产生阶段完成以后，生成的输出会被传递到 **filter** 模块去进行处理。**filter** 模块也是与 **location** 相关的。所有的 **filter** 模块都被组织成一条链。输出会依次穿越所有的 **filter**，直到有一个 **filter** 模块的返回值表明已经处理完成。

这里列举几个常见的 **filter** 模块，例如：

1. **server-side includes**。
2. **XSLT filtering**。
3. 图像缩放之类的。
4. **gzip 压缩**。

在所有的 **filter** 中，有几个 **filter** 模块需要关注一下。按照调用的顺序依次说明如下：

write: 写输出到客户端，实际上是写到连接对应的 **socket** 上。
postpone: 这个 **filter** 是负责 **subrequest** 的，也就是子请求的。
copy: 将一些需要复制的 **buf** (文件或者内存) 重新复制一份然后交给剩余的 **body filter** 处理。

handler 模块(100%)

handler 模块简介

相信大家在看了前一章的模块概述以后，都对 **nginx** 的模块有了一个基本的认识。基本上作为第三方开发者最可能开发的就是三种类型的模块，即 **handler**，**filter** 和 **load-balancer**。**Handler** 模块就是接受来自客户端的请求并产生输出的模块。有些地方说 **upstream** 模块实际上也是一种 **handler** 模块，只不过它产生的内容来自于从后端服务器获取的，而非在本机产生的。

在上一章提到，配置文件中使用 **location** 指令可以配置 **content handler** 模块，当 **Nginx** 系统启动的时候，每个 **handler** 模块都有一次机会把自己关联到对应的 **location** 上。如果有多个 **handler** 模块都关联了同一个 **location**，那么实际上只有一个 **handler** 模块真正会起作用。当然大多数情况下，模块开发人员都会避免出现这种情况。

handler 模块处理的结果通常有三种情况：处理成功，处理失败（处理的时候发生了错误）或者是拒绝去处理。在拒绝处理的情况下，这个 **location** 的处理就会由默认的 **handler** 模块来进行处理。例

如，当请求一个静态文件的时候，如果关联到这个 location 上的一个 handler 模块拒绝处理，就会由默认的 ngx_http_static_module 模块进行处理，该模块是一个典型的 handler 模块。

本章主要讲述的是如何编写 handler 模块，在研究 handler 模块编写之前先来了解一下模块的一些基本数据结构。

模块的基本结构

在这一节我们将会对通常的模块开发过程中，每个模块所包含的一些常用的部分进行说明。这些部分有些是必须的，有些不是必须的。同时这里所列出的这些东西对于其他类型的模块，例如 filter 模块等也都是相同的。

模块配置结构

基本上每个模块都会提供一些配置指令，以便于用户可以通过配置来控制该模块的行为。那么这些配置信息怎么存储呢？那就需要定义该模块的配置结构来进行存储。

大家都知道 Nginx 的配置信息分成了几个作用域(scope,有时也称作上下文)，这就是 main, server, 以及 location。同样的每个模块提供的配置指令也可以出现在这几个作用域里。那对于这三个作用域的配置信息，每个模块就需要定义三个不同的数据结构去进行存储。当然，不是每个模块都会在这三个作用域都提供配置指令的。那么也就不一定每个模块都需要定义三个数据结构去存储这些配置信息了。视模块的实现而言，需要几个就定义几个。

有一点需要特别注意的就是，在模块的开发过程中，我们最好使用 nginx 原有的命名习惯。这样跟原代码的契合度更高，看起来也更舒服。

对于模块配置信息的定义，命名习惯是 ngx_http_<module name>_(main|srv|loc)_conf_t。这里有个例子，就是从我们后面将要展示给大家的 hello module 中截取的。

```
typedef struct
{
    ngx_str_t hello_string;
    ngx_int_t hello_counter;
}ngx_http_hello_loc_conf_t;
```

模块配置指令

一个模块的配置指令是定义在一个静态数组中的。同样地，我们来看一下从 hello module 中截取的模块配置指令的定义。

```
static ngx_command_t ngx_http_hello_commands[] = {
    {
        ngx_string("hello_string"),
        NGX_HTTP_LOC_CONF|NGX_CONF_NOARGS|NGX_CONF_TAKE1,
        ngx_http_hello_string,
        NGX_HTTP_LOC_CONF_OFFSET,
        offsetof(ngx_http_hello_loc_conf_t, hello_string),
        NULL },

    {
        ngx_string("hello_counter"),
        NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
        ngx_http_hello_counter,
        NGX_HTTP_LOC_CONF_OFFSET,
        offsetof(ngx_http_hello_loc_conf_t, hello_counter),
        NULL },

    ngx_null_command
};
```

其实看这个定义，就基本能看出来一些信息。例如，我们是定义了两个配置指令，一个是叫 `hello_string`，可以接受一个参数，或者是没有参数。另外一个命令是 `hello_counter`，接受一个 `NGX_CONF_FLAG` 类型的参数。除此之外，似乎看起来有点迷惑。没有关系，我们来详细看一下 `ngx_command_t`，一旦我们了解这个结构的详细信息，那么我相信上述这个定义所表达的所有信息就不言自明了。

`ngx_command_t` 的定义，位于 `src/core/nginx_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: 配置指令的名称。

该配置的类型，其实更准确一点说，是该配置指令属性的集合。`nginx` 提供了很多预定义的属性值（一些宏

type: 定义），通过逻辑或运算符可组合在一起，形成对这个配置指令的详细的说明。下面列出可在这里使用的预定义属性值及说明。

- `NGX_CONF_NOARGS`: 配置指令不接受任何参数。
- `NGX_CONF_TAKE1`: 配置指令接受 1 个参数。

- NGX_CONF_TAKE2: 配置指令接受 2 个参数。
- NGX_CONF_TAKE3: 配置指令接受 3 个参数。
- NGX_CONF_TAKE4: 配置指令接受 4 个参数。
- NGX_CONF_TAKE5: 配置指令接受 5 个参数。
- NGX_CONF_TAKE6: 配置指令接受 6 个参数。
- NGX_CONF_TAKE7: 配置指令接受 7 个参数。

可以组合多个属性，比如一个指令即可以不填参数，也可以接受 1 个或者 2 个参数。那么就是 NGX_CONF_NOARGS|NGX_CONF_TAKE1|NGX_CONF_TAKE2。如果写上面三个属性在一起，你觉得麻烦，那么没有关系，nginx 提供了一些定义，使用起来更简洁。

- NGX_CONF_TAKE12: 配置指令接受 1 个或者 2 个参数。
- NGX_CONF_TAKE13: 配置指令接受 1 个或者 3 个参数。
- NGX_CONF_TAKE23: 配置指令接受 2 个或者 3 个参数。
- NGX_CONF_TAKE123: 配置指令接受 1 个或者 2 个或者 3 参数。
- NGX_CONF_TAKE1234: 配置指令接受 1 个或者 2 个或者 3 个或者 4 个参数。
- NGX_CONF_1MORE: 配置指令接受至少一个参数。
- NGX_CONF_2MORE: 配置指令接受至少两个参数。
- NGX_CONF_MULTI: 配置指令可以接受多个参数，即个数不定。
- NGX_CONF_BLOCK: 配置指令可以接受的值是一个配置信息块。也就是一对大括号括起来的内容。里面可以再包括很多的配置指令。比如常见的 **server** 指令就是这个属性的。
- NGX_CONF_FLAG: 配置指令可以接受的值是“on”或者“off”，最终会被转成 bool 值。

- `NGX_CONF_ANY`: 配置指令可以接受的任意的参数值。一个或者多个，或者“on”或者“off”，或者是配置块。

最后要说明的是，无论如何，`nginx` 的配置指令的参数个数不可以超过 `NGX_CONF_MAX_ARGS` 个。目前这个值被定义为 8，也就是不能超过 8 个参数值。

下面介绍一组说明配置指令可以出现的位置的属性。

- `NGX_DIRECT_CONF`: 可以出现在配置文件中最外层。例如已经提供的配置指令 `daemon`, `master_process` 等。
- `NGX_MAIN_CONF`: `http`、`mail`、`events`、`error_log` 等。
- `NGX_ANY_CONF`: 该配置指令可以出现在任意配置级别上。

对于我们编写的大多数模块而言，都是在处理 `http` 相关的事情，也就是所谓的都是 `NGX_HTTP_MODULE`，对于这样类型的模块，其配置可能出现的位置也是分为直接出现在 `http` 里面，以及其他位置。

- `NGX_HTTP_MAIN_CONF`: 可以直接出现在 `http` 配置指令里。
- `NGX_HTTP_SRV_CONF`: 可以出现在 `http` 里面的 `server` 配置指令里。
- `NGX_HTTP_LOC_CONF`: 可以出现在 `http server` 块里面的 `location` 配置指令里。
- `NGX_HTTP_UPS_CONF`: 可以出现在 `http` 里面的 `upstream` 配置指令里。
- `NGX_HTTP_SIF_CONF`: 可以出现在 `http` 里面的 `server` 配置指令里的 `if` 语句所在的 `block` 中。
- `NGX_HTTP_LMT_CONF`: 可以出现在 `http` 里面的 `limit_except` 指令的 `block` 中。
- `NGX_HTTP_LIF_CONF`: 可以出现在 `http server` 块里面的 `location` 配置指令里的 `if` 语句所在的 `block` 中。

这是一个函数指针，当 `nginx` 在解析配置的时候，如果遇到这个配置指令，将会把读取到的值传递给这个函数 **set**：进行分解处理。因为具体每个配置指令的值如何处理，只有定义这个配置指令的人是最清楚的。来看一下这个函数指针要求的函数原型。

```
char *(*set)(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
```

先看该函数的返回值，处理成功时，返回 `NGX_OK`，否则返回 `NGX_CONF_ERROR` 或者是一个自定义的错误信息的字符串。

再看一下这个函数被调用的时候，传入的三个参数。

- `cf`: 该参数里面保存从配置文件读取到的原始字符串以及相关的一些信息。特别注意的是这个参数的 `args` 字段是一个 `ngx_str_t` 类型的数组，该数组的首个元素是这个配置指令本身，第二个元素是指令的第一个参数，第三个元素是第二个参数，依次类推。
- `cmd`: 这个配置指令对应的 `ngx_command_t` 结构。
- `conf`: 就是定义的存储这个配置值的结构体，比如在上面展示的那个 `ngx_http_hello_loc_conf_t`。当解析这个 `hello_string` 变量的时候，传入的 `conf` 就指向一个 `ngx_http_hello_loc_conf_t` 类型的变量。用户在处理的时候可以使用类型转换，转换成自己知道的类型，再进行字段的赋值。

为了更加方便的实现对配置指令参数的读取，`nginx` 已经默认提供了对一些标准类型的参数进行读取的函数，可以直接赋值给 `set` 字段使用。下面来看一下这些已经实现的 `set` 类型函数。

- `ngx_conf_set_flag_slot`: 读取 `NGX_CONF_FLAG` 类型的参数。
- `ngx_conf_set_str_slot`: 读取字符串类型的参数。
- `ngx_conf_set_str_array_slot`: 读取字符串数组类型的参数。
- `ngx_conf_set_keyval_slot`: 读取键值对类型的参数。
- `ngx_conf_set_num_slot`: 读取整数类型(有符号整数 `ngx_int_t`)的参数。
- `ngx_conf_set_size_slot`: 读取 `size_t` 类型的参数，也就是无符号数。
- `ngx_conf_set_off_slot`: 读取 `off_t` 类型的参数。
- `ngx_conf_set_msec_slot`: 读取毫秒值类型的参数。
- `ngx_conf_set_sec_slot`: 读取秒值类型的参数。
- `ngx_conf_set_bufs_slot`: 读取的参数值是 2 个，一个是 `buf` 的个数，一个是 `buf` 的大小。例如: `output_buffers 1 128k`;
- `ngx_conf_set_enum_slot`: 读取枚举类型的参数，将其转换成整数 `ngx_uint_t` 类型。
- `ngx_conf_set_bitmask_slot`: 读取参数的值，并将这些参数的值以 `bit` 位的形式存储。例如: `HttpDavModule` 模块的 `dav_methods` 指令。

conf: 该字段被 `NGX_HTTP_MODULE` 类型模块所用 (我们编写的基本上都是 `NGX_HTTP_MODULE`，只有一些 `nginx` 核心模块是非 `NGX_HTTP_MODULE`)，该字段指定当前配置项存储的内存位置。实际上是使用哪个内存池的问题。因为 `http` 模块对所有 `http` 模块所要保存的配置信息，划分了 `main`, `server` 和 `location` 三个地方进行存储，每个地方都有一个内存池用来分配存储这些信息的内存。这里可能的值为 `NGX_HTTP_MAIN_CONF_OFFSET`、`NGX_HTTP_SRV_CONF_OFFSET` 或 `NGX_HTTP_LOC_CONF_OFFSET`。当然也可以直接置为 0，就是 `NGX_HTTP_MAIN_CONF_OFFSET`。

offset: 指定该配置项值的精确存放位置，一般指定为某一个结构体变量的字段偏移。因为对于配置信息的存储，一般我们都是定义个结构体来存储的。那么比如我们定义了一个结构体 `A`，该项配置的值需要存储到该结构体的 `b` 字段。那么在这里就可以填写为 `offsetof(A, b)`。对于有些配置项，它的值不需要保存或者是需要保存到更为复杂的结构中时，这里可以设置为 0。

post: 该字段存储一个指针。可以指向任何一个在读取配置过程中需要的数据，以便于进行配置读取的处理。大多数时候，都不需要，所以简单地设为 0 即可。

看到这里，应该就比较清楚了。ngx_http_hello_commands 这个数组每 5 个元素为一组，用来描述一个配置项的所有情况。那么如果有多个配置项，只要按照需要再增加 5 个对应的元素对新的配置项进行说明。

需要注意的是，就是在 **ngx_http_hello_commands** 这个数组定义的最后，都要加一个 **ngx_null_command** 作为结尾。

模块上下文结构

这是一个 ngx_http_module_t 类型的静态变量。这个变量实际上是提供一组回调函数指针，这些函数有在创建存储配置信息的对象的函数，也有在创建前和创建后会调用的函数。这些函数都将被 nginx 在合适的时间进行调用。

```
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;
```

preconfiguration:

在创建和读取该模块的配置信息之前被调用。

postconfiguration:

在创建和读取该模块的配置信息之后被调用。

create_main_conf:

调用该函数创建本模块位于 http block 的配置信息存储结构。该函数成功的时候，返回创建的配置对象。失败的话，返回 NULL。

init_main_conf:

调用该函数初始化本模块位于 http block 的配置信息存储结构。该函数成功的时候，返回 NGX_CONF_OK。失败的话，返回 NGX_CONF_ERROR 或错误字符串。

create_srv_conf:

调用该函数创建本模块位于 http server block 的配置信息存储结构，每个 server block 会创建一个。该函数成功的时候，返回创建的配置对象。失败的话，返回 NULL。

因为有些配置指令既可以出现在 http block，也可以出现在 http server block 中。那么遇到这种情况，每个 server 都会有自己存储结构来存储该 server 的配置，但是在这种情况下 http block 中的配置与 server block 中的配置信息发生冲突的时候，就需要调用此函数进行合并，该函数并非必须提供，当预计到绝对不会发生需要合并的情况的时候，就无需提供。当然为了安全起见还是建议提供。

merge_srv_conf:

该函数执行成功的时候，返回 `NGX_CONF_OK`。失败的话，返回 `NGX_CONF_ERROR` 或错误字符串。

create_loc_conf:

调用该函数创建本模块位于 `location block` 的配置信息存储结构。每个在配置中指明的 `location` 创建一个。该函数执行成功，返回创建的配置对象。失败的话，返回 `NULL`。

merge_loc_conf:

与 `merge_srv_conf` 类似，这个也是进行配置值合并的地方。该函数成功的时候，返回 `NGX_CONF_OK`。失败的话，返回 `NGX_CONF_ERROR` 或错误字符串。

Nginx 里面的配置信息都是上下一层层的嵌套的，对于具体某个 `location` 的话，对于同一个配置，如果当前层次没有定义，那么就使用上层的配置，否则使用当前层次的配置。

这些配置信息一般默认都应该设为一个未初始化的值，针对这个需求，Nginx 定义了一系列的宏定义来代表各种配置所对应数据类型的未初始化值，如下：

```
#define NGX_CONF_UNSET      -1
#define NGX_CONF_UNSET_UINT (ngx_uint_t) -1
#define NGX_CONF_UNSET_PTR  (void *) -1
#define NGX_CONF_UNSET_SIZE (size_t) -1
#define NGX_CONF_UNSET_MSEC (ngx_msec_t) -1
```

又因为对于配置项的合并，逻辑都类似，也就是前面已经说过的，如果在本层次已经配置了，也就是配置项的值已经被读取进来了（那么这些配置项的值就不会等于上面已经定义的那些 `UNSET` 的值），就使用本层次的值作为定义合并的结果，否则，使用上层的值，如果上层的值也是这些 `UNSET` 类的值，那就赋值为默认值，否则就使用上层的值作为合并的结果。对于这样类似的操作，Nginx 定义了一些宏操作来做这些事情，我们来看其中一个的定义。

```
#define ngx_conf_merge_uint_value(conf, prev, default) \
    if (conf == NGX_CONF_UNSET_UINT) { \
        conf = (prev == NGX_CONF_UNSET_UINT) ? default : prev; \
    }
```

显而易见，这个逻辑确实比较简单，所以其它的宏定义也类似，我们就列具其中的一部分吧。

```
ngx_conf_merge_value
ngx_conf_merge_ptr_value
ngx_conf_merge_uint_value
ngx_conf_merge_msec_value
ngx_conf_merge_sec_value
```

等等。

下面来看一下 `hello` 模块的模块上下文的定义，加深一下印象。

```
static ngx_http_module_t ngx_http_hello_module_ctx = {
    NULL,                                /* preconfiguration */
    ngx_http_hello_init,                 /* postconfiguration */

    NULL,                                /* create main configuration */
    NULL,                                /* init main configuration */

    NULL,                                /* create server configuration */
    NULL,                                /* merge server configuration */

    ngx_http_hello_create_loc_conf, /* create location configuration */
    NULL                                /* merge location configuration */
};
```

注意：这里并没有提供 **merge_loc_conf** 函数，因为我们这个模块的配置指令已经确定只出现在 **NGX_HTTP_LOC_CONF** 中这一个层次上，不会发生需要合并的情况。

模块的定义

对于开发一个模块来说，我们都需要定义一个 `ngx_module_t` 类型的变量来说明这个模块本身的信息，从某种意义上来说，这是这个模块最重要的一个信息，它告诉了 `nginx` 这个模块的一些信息，上面定义的配置信息，还有模块上下文信息，都是通过这个结构来告诉 `nginx` 系统的，也就是加载模块的上层代码，都需要通过定义的这个结构，来获取这些信息。

我们先来看下 `ngx_module_t` 的定义

```
typedef struct ngx_module_s      ngx_module_t;
struct ngx_module_s {
    ngx_uint_t      ctx_index;
    ngx_uint_t      index;
    ngx_uint_t      spare0;
    ngx_uint_t      spare1;
    ngx_uint_t      abi_compatibility;
    ngx_uint_t      major_version;
    ngx_uint_t      minor_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;
};

#define NGX_NUMBER_MAJOR 3
#define NGX_NUMBER_MINOR 1
#define NGX_MODULE_V1      0, 0, 0, 0, \
    NGX_DSO_ABI_COMPATIBILITY, NGX_NUMBER_MAJOR, NGX_NUMBER_MINOR
#define NGX_MODULE_V1_PADDING 0, 0, 0, 0, 0, 0, 0, 0

```

再看一下 `hello` 模块的模块定义。

```

ngx_module_t ngx_http_hello_module = {
    NGX_MODULE_V1,
    &ngx_http_hello_module_ctx,    /* module context */
    ngx_http_hello_commands,      /* module directives */
    NGX_HTTP_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
};

```

模块可以提供一些回调函数给 `nginx`，当 `nginx` 在创建进程线程或者结束进程线程时进行调用。但

大多数模块在这些时刻并不需要做什么，所以都简单赋值为 `NULL`。

handler 模块的基本结构

除了上一节介绍的模块的基本结构以外，`handler` 模块必须提供一个真正的处理函数，这个函数负责来自客户端请求的真正处理。这个函数的处理，既可以选择自己直接生成内容，也可以选择拒绝

处理，由后续的 handler 去进行处理，或者是选择丢给后续的 filter 进行处理。来看一下这个函数的原型申明。

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

r 是 http 请求。里面包含请求所有的信息，这里不详细说明了，可以参考别的章节的介绍。该函数处理成功返回 NGX_OK, 处理发生错误返回 NGX_ERROR, 拒绝处理（留给后续的 handler 进行处理）返回 NGX_DECLINE。 返回 NGX_OK 也就代表给客户端的响应已经生成好了，否则返回 NGX_ERROR 就发生错误了。

handler 模块的挂载

handler 模块真正的处理函数通过两种方式挂载到处理过程中，一种方式就是按处理阶段挂载;另外一种挂载方式就是按需挂载。

按处理阶段挂载

为了更精细地控制对于客户端请求的处理过程，nginx 把这个处理过程划分成了 11 个阶段。他们从前到后，依次列举如下：

NGX_HTTP_POST_READ_PHASE:

读取请求内容阶段

NGX_HTTP_SERVER_REWRITE_PHASE:

Server 请求地址重写阶段

NGX_HTTP_FIND_CONFIG_PHASE:

配置查找阶段:

NGX_HTTP_REWRITE_PHASE:

Location 请求地址重写阶段

NGX_HTTP_POST_REWRITE_PHASE:

请求地址重写提交阶段

NGX_HTTP_PREACCESS_PHASE:

访问权限检查准备阶段

NGX_HTTP_ACCESS_PHASE:

访问权限检查阶段

NGX_HTTP_POST_ACCESS_PHASE:

访问权限检查提交阶段

NGX_HTTP_TRY_FILES_PHASE:

配置项 try_files 处理阶段

NGX_HTTP_CONTENT_PHASE:

内容产生阶段

NGX_HTTP_LOG_PHASE:

日志模块处理阶段

一般情况下，我们自定义的模块，大多数是挂载在 NGX_HTTP_CONTENT_PHASE 阶段的。挂载的动作一般是在模块上下文调用的 postconfiguration 函数中。

注意:有几个阶段是特例,它不调用挂载地任何的 **handler**,也就是你就不用挂载到这几个阶段了:

- NGX_HTTP_FIND_CONFIG_PHASE
- NGX_HTTP_POST_ACCESS_PHASE
- NGX_HTTP_POST_REWRITE_PHASE
- NGX_HTTP_TRY_FILES_PHASE

所以其实真正是有 7 个 phase 你可以去挂载 handler。

挂载的代码如下（摘自 hello module）：

```
static ngx_int_t
ngx_http_hello_init(ngx_conf_t *cf)
{
    ngx_http_handler_pt      *h;
    ngx_http_core_main_conf_t *cmcf;

    cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);

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

    *h = ngx_http_hello_handler;

    return NGX_OK;
}
```

使用这种方式挂载的 handler 也被称为 **content phase handlers**。

按需挂载

以这种方式挂载的 handler 也被称为 **content handler**。

当一个请求进来以后，nginx 从 `NGX_HTTP_POST_READ_PHASE` 阶段开始依次执行每个阶段中所有 handler。执行到 `NGX_HTTP_CONTENT_PHASE` 阶段的时候，如果这个 location 有一个对应的 content handler 模块，那么就去执行这个 content handler 模块真正的处理函数。否则继续依次执行 `NGX_HTTP_CONTENT_PHASE` 阶段中所有 content phase handlers，直到某个函数处理返回 `NGX_OK` 或者 `NGX_ERROR`。

换句话说，当某个 location 处理到 `NGX_HTTP_CONTENT_PHASE` 阶段时，如果有 content handler 模块，那么 `NGX_HTTP_CONTENT_PHASE` 挂载的所有 content phase handlers 都不会被执行了。

但是使用这个方法挂载上去的 handler 有一个特点是必须在 `NGX_HTTP_CONTENT_PHASE` 阶段才能执行到。如果你想自己的 handler 在更早的阶段执行，那就不要使用这种挂载方式。

那么在什么情况会使用这种方式来挂载呢？一般情况下，某个模块对某个 location 进行了处理以后，发现符合自己处理的逻辑，而且也没有必要再调用 `NGX_HTTP_CONTENT_PHASE` 阶段的其它 handler 进行处理的时候，就动态挂载上这个 handler。

下面来看一下使用这种挂载方式的具体例子（摘自 *Emiller's Guide To Nginx Module Development*）。

```
static char *
ngx_http_circle_gif(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
{
    ngx_http_core_loc_conf_t *clcf;

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

    return NGX_CONF_OK;
}
```

handler 的编写步骤

好，到了这里，让我们稍微整理一下思路，回顾一下实现一个 handler 的步骤：

1. 编写模块基本结构。包括模块的定义，模块上下文结构，模块的配置结构等。

2. 实现 handler 的挂载函数。根据模块的需求选择正确的挂载方式。
3. 编写 handler 处理函数。模块的功能主要通过这个函数来完成。

看起来不是那么难，对吧？还是那句老话，世上无难事，只怕有心人！现在我们来完整的分析前面提到的 hello handler module 示例的功能和代码。

示例: hello handler 模块

在前面已经看到了这个 hello handler module 的部分重要的结构。该模块提供了 2 个配置指令，仅可以出现在 location 指令的作用域中。这两个指令是 hello_string, 该指令接受一个参数来设置显示的字符串。如果没有跟参数，那么就使用默认的字符串作为响应字符串。

另一个指令是 hello_counter, 如果设置为 on, 则会在响应的字符串后面追加 Visited Times: 的字样，以统计请求的次数。

这里有两点注意一下：

1. 对于 flag 类型的配置指令，当值为 off 的时候，使用 ngx_conf_set_flag_slot 函数，会转化为 0，为 on，则转化为非 0。
2. 另外一个，我提供了 merge_loc_conf 函数，但是却没有设置到模块的上下文定义中。这样有一个缺点，就是如果一个指令没有出现在配置文件中的时候，配置信息中的值，将永远会保持在 create_loc_conf 中的初始化的值。那如果，在类似 create_loc_conf 这样的函数中，对创建出来的配置信息的值，没有设置为合理的值的话，后面用户又没有配置，就会出现问题的。

下面来完整的给出 ngx_http_hello_module 模块的完整代码。

```
#include <ngx_config.h>
#include <ngx_core.h>
#include <ngx_http.h>

typedef struct
{
    ngx_str_t hello_string;
    ngx_int_t hello_counter;
}ngx_http_hello_loc_conf_t;

static ngx_int_t ngx_http_hello_init(ngx_conf_t *cf);

static void *ngx_http_hello_create_loc_conf(ngx_conf_t *cf);

static char *ngx_http_hello_string(ngx_conf_t *cf, ngx_command_t *cmd,
    void *conf);
static char *ngx_http_hello_counter(ngx_conf_t *cf, ngx_command_t *cmd,
```



```

    void *conf);

static ngx_command_t ngx_http_hello_commands[] = {
    {
        ngx_string("hello_string"),
        NGX_HTTP_LOC_CONF|NGX_CONF_NOARGS|NGX_CONF_TAKE1,
        ngx_http_hello_string,
        NGX_HTTP_LOC_CONF_OFFSET,
        offsetof(ngx_http_hello_loc_conf_t, hello_string),
        NULL },

    {
        ngx_string("hello_counter"),
        NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
        ngx_http_hello_counter,
        NGX_HTTP_LOC_CONF_OFFSET,
        offsetof(ngx_http_hello_loc_conf_t, hello_counter),
        NULL },

    ngx_null_command
};

/*
static u_char ngx_hello_default_string[] = "Default String: Hello, world!";
*/
static int ngx_hello_visited_times = 0;

static ngx_http_module_t ngx_http_hello_module_ctx = {
    NULL, /* preconfiguration */
    ngx_http_hello_init, /* postconfiguration */

    NULL, /* create main configuration */
    NULL, /* init main configuration */

    NULL, /* create server configuration */
    NULL, /* merge server configuration */

    ngx_http_hello_create_loc_conf, /* create location configuration */
    NULL /* merge location configuration */
};

ngx_module_t ngx_http_hello_module = {
    NGX_MODULE_V1,
    &ngx_http_hello_module_ctx, /* module context */
    ngx_http_hello_commands, /* module directives */
    NGX_HTTP_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
};

static ngx_int_t
ngx_http_hello_handler(ngx_http_request_t *r)
{
    ngx_int_t    rc;
    ngx_buf_t    *b;
    ngx_chain_t  out;
    ngx_http_hello_loc_conf_t* my_conf;
    u_char ngx_hello_string[1024] = {0};
    ngx_uint_t content_length = 0;

    ngx_log_error(NGX_LOG_EMERG, r->connection->log, 0,
"ngx_http_hello_handler is called!");

    my_conf = ngx_http_get_module_loc_conf(r, ngx_http_hello_module);
    if (my_conf->hello_string.len == 0 )
    {
        ngx_log_error(NGX_LOG_EMERG, r->connection->log, 0, "hello_string
is empty!");
        return NGX_DECLINED;
    }

    if (my_conf->hello_counter == NGX_CONF_UNSET
        || my_conf->hello_counter == 0)
    {
        ngx_sprintf(ngx_hello_string, "%s", my_conf->hello_string.data);
    }
    else
    {
        ngx_sprintf(ngx_hello_string, "%s Visited Times:%d",
my_conf->hello_string.data,
                    ++ngx_hello_visited_times);
    }
    ngx_log_error(NGX_LOG_EMERG, r->connection->log, 0, "hello_string:%s",
ngx_hello_string);
    content_length = ngx_strlen(ngx_hello_string);

```

```

/* we response to 'GET' and 'HEAD' requests only */
if (!(r->method & (NGX_HTTP_GET|NGX_HTTP_HEAD))) {
    return NGX_HTTP_NOT_ALLOWED;
}

/* discard request body, since we don't need it here */
rc = ngx_http_discard_request_body(r);

if (rc != NGX_OK) {
    return rc;
}

/* set the 'Content-type' header */
/*
 *r->headers_out.content_type.len = sizeof("text/html") - 1;
 *r->headers_out.content_type.data = (u_char *)"text/html";
 */
ngx_str_set(&r->headers_out.content_type, "text/html");

/* send the header only, if the request type is http 'HEAD' */
if (r->method == NGX_HTTP_HEAD) {
    r->headers_out.status = NGX_HTTP_OK;
    r->headers_out.content_length_n = content_length;

    return ngx_http_send_header(r);
}

/* allocate a buffer for your response body */
b = ngx_palloc(r->pool, sizeof(ngx_buf_t));
if (b == NULL) {
    return NGX_HTTP_INTERNAL_SERVER_ERROR;
}

/* attach this buffer to the buffer chain */
out.buf = b;
out.next = NULL;

/* adjust the pointers of the buffer */
b->pos = ngx_hello_string;
b->last = ngx_hello_string + content_length;
b->memory = 1;    /* this buffer is in memory */
b->last_buf = 1; /* this is the last buffer in the buffer chain */

/* set the status line */
r->headers_out.status = NGX_HTTP_OK;
r->headers_out.content_length_n = content_length;

```

```

    /* send the headers of your response */
    rc = ngx_http_send_header(r);

    if (rc == NGX_ERROR || rc > NGX_OK || r->header_only) {
        return rc;
    }

    /* send the buffer chain of your response */
    return ngx_http_output_filter(r, &out);
}

static void *ngx_http_hello_create_loc_conf(ngx_conf_t *cf)
{
    ngx_http_hello_loc_conf_t* local_conf = NULL;
    local_conf = ngx_palloc(cf->pool, sizeof(ngx_http_hello_loc_conf_t));
    if (local_conf == NULL)
    {
        return NULL;
    }

    ngx_str_null(&local_conf->hello_string);
    local_conf->hello_counter = NGX_CONF_UNSET;

    return local_conf;
}

/*
static char *ngx_http_hello_merge_loc_conf(ngx_conf_t *cf, void *parent, void
*child)
{
    ngx_http_hello_loc_conf_t* prev = parent;
    ngx_http_hello_loc_conf_t* conf = child;

    ngx_conf_merge_str_value(conf->hello_string, prev->hello_string,
ngx_hello_default_string);
    ngx_conf_merge_value(conf->hello_counter, prev->hello_counter, 0);

    return NGX_CONF_OK;
}*/

static char *
ngx_http_hello_string(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
{
    ngx_http_hello_loc_conf_t* local_conf;

    local_conf = conf;

```

```

        char* rv = ngx_conf_set_str_slot(cf, cmd, conf);

        ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "hello_string:%s",
local_conf->hello_string.data);

        return rv;
}

static char *ngx_http_hello_counter(ngx_conf_t *cf, ngx_command_t *cmd,
void *conf)
{
    ngx_http_hello_loc_conf_t* local_conf;

    local_conf = conf;

    char* rv = NULL;

    rv = ngx_conf_set_flag_slot(cf, cmd, conf);

    ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "hello_counter:%d",
local_conf->hello_counter);
    return rv;
}

static ngx_int_t
ngx_http_hello_init(ngx_conf_t *cf)
{
    ngx_http_handler_pt      *h;
    ngx_http_core_main_conf_t *cmcf;

    cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);

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

    *h = ngx_http_hello_handler;

    return NGX_OK;
}

```

通过上面一些介绍，我相信大家都能对整个示例模块有一个比较好的理解。唯一可能感觉有些理解困难的地方在于 `ngx_http_hello_handler` 函数里面产生和设置输出。但其实大家在本书的前面的相关章

节都可以看到对 `ngx_buf_t` 和 `request` 等相关数据结构的说明。如果仔细看了这些地方的说明的话，应该对这里代码的实现就比较容易理解了。因此，这里不再赘述解释。

handler 模块的编译和使用

模块的功能开发完了之后，模块的使用还需要编译才能够执行，下面我们来看下模块的编译和使用。

config 文件的编写

对于开发一个模块，我们是需要把这个模块的 C 代码组织到一个目录里，同时需要编写一个 config 文件。这个 config 文件的内容就是告诉 nginx 的编译脚本，该如何进行编译。我们来看一下 hello handler module 的 config 文件的内容，然后再做解释。

```
ngx_addon_name=ngx_http_hello_module
HTTP_MODULES="$HTTP_MODULES ngx_http_hello_module"
NGX_ADDON_SRCS="$NGX_ADDON_SRCS $ngx_addon_dir/nginx_http_hello_module.c"
```

其实文件很简单，几乎不需要做什么解释。大家一看都懂了。唯一需要说明的是，如果这个模块的实现有多个源文件，那么都在 `NGX_ADDON_SRCS` 这个变量里，依次写进去就可以。

编译

对于模块的编译，nginx 并不像 apache 一样，提供了单独的编译工具，可以在没有 apache 源代码的情况下单独编译一个模块的代码。nginx 必须去到 nginx 的源代码目录里，通过 `configure` 指令的参数，来进行编译。下面看一下 hello module 的 `configure` 指令：

```
./configure --prefix=/usr/local/nginx-1.3.1 --
add-module=/home/jizhao/open_source/book_module
```

我写的这个示例模块的代码和 config 文件都放在 `/home/jizhao/open_source/book_module` 这个目录下。所以一切都很明了，也没什么好说的了。

使用

使用一个模块需要根据这个模块定义的配置指令来做。比如我们这个简单的 hello handler module 的使用就很简单。在我的测试服务器的配置文件里，就是在 http 里面的默认的 server 里面加入如下的配置：

```
location /test {
    hello_string jizhao;
    hello_counter on;
}
```

当我们访问这个地址的时候, lynx <http://127.0.0.1/test> 的时候，就可以看到返回的结果。

jizhao Visited Times:1

当然你访问多次，这个次数是会增加的。

更多 handler 模块示例分析

http access module

该模块的代码位于 src/http/modules/nginx_http_access_module.c 中。该模块的作用是提供对于特定 host 的客户端的访问控制。可以限定特定 host 的客户端对于服务端全部，或者某个 server，或者是某个 location 的访问。 该模块的实现非常简单，总共也就只有几个函数。

```
static ngx_int_t ngx_http_access_handler(ngx_http_request_t *r);
static ngx_int_t ngx_http_access_inet(ngx_http_request_t *r,
    ngx_http_access_loc_conf_t *alcf, in_addr_t addr);
#if (NGX_HAVE_INET6)
static ngx_int_t ngx_http_access_inet6(ngx_http_request_t *r,
    ngx_http_access_loc_conf_t *alcf, u_char *p);
#endif
static ngx_int_t ngx_http_access_found(ngx_http_request_t *r, ngx_uint_t deny);
static char *ngx_http_access_rule(ngx_conf_t *cf, ngx_command_t *cmd,
    void *conf);
static void *ngx_http_access_create_loc_conf(ngx_conf_t *cf);
static char *ngx_http_access_merge_loc_conf(ngx_conf_t *cf,
    void *parent, void *child);
static ngx_int_t ngx_http_access_init(ngx_conf_t *cf);
```

对于与配置相关的几个函数都不需要做解释了，需要提一下的是函数 ngx_http_access_init，该函数在实现上把本模块挂载到了 NGX_HTTP_ACCESS_PHASE 阶段的 handler 上，从而使自己的被调用时

机发生在了 NGX_HTTP_CONTENT_PHASE 等阶段前。因为进行客户端地址的限制检查，根本不需要等到这么后面。

另外看一下这个模块的主处理函数 ngx_http_access_handler。这个函数的逻辑也非常简单，主要是根据客户端地址的类型，来分别选择 ipv4 类型的处理函数 ngx_http_access_inet 还是 ipv6 类型的处理函数 ngx_http_access_inet6。

而这个两个处理函数内部也非常简单，就是循环检查每个规则，检查是否有匹配的规则，如果有就返回匹配的结果，如果都没有匹配，就默认拒绝。

http static module

从某种程度上来说，此模块可以算的上是“最正宗的”，“最古老”的 content handler。因为本模块的作用就是读取磁盘上的静态文件，并把文件内容作为产生的输出。在 Web 技术发展的早期，只有静态页面，没有服务端脚本来动态生成 HTML 的时候。恐怕开发个 Web 服务器的时候，第一个要开发就是这样一个 content handler。

http static module 的代码位于 src/http/modules/nginx_http_static_module.c 中，总共只有两百多行近三百行。可以说是非常短小。

我们首先来看一下该模块的模块上下文的定义。

```
ngx_http_module_t  ngx_http_static_module_ctx = {
    NULL,                                /* preconfiguration */
    ngx_http_static_init,                /* postconfiguration */

    NULL,                                /* create main configuration */
    NULL,                                /* init main configuration */

    NULL,                                /* create server configuration */
    NULL,                                /* merge server configuration */

    NULL,                                /* create location configuration */
    NULL,                                /* merge location configuration */
};
```

是非常的简洁吧，连任何与配置相关的函数都没有。对了，因为该模块没有提供任何配置指令。大家想想也就知道了，这个模块做的事情实在是太简单了，也确实没什么好配置的。唯一需要调用的函数是一个 ngx_http_static_init 函数。好了，来看一下这个函数都干了写什么。


```

static ngx_int_t
ngx_http_static_init(ngx_conf_t *cf)
{
    ngx_http_handler_pt      *h;
    ngx_http_core_main_conf_t *cmcf;

    cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);

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

    *h = ngx_http_static_handler;

    return NGX_OK;
}

```

仅仅是挂载这个 handler 到 NGX_HTTP_CONTENT_PHASE 处理阶段。简单吧？

下面我们就看一下这个模块最核心的处理逻辑所在的 ngx_http_static_handler 函数。该函数大概占了这个模块代码量的百分之八九十。

```

static ngx_int_t
ngx_http_static_handler(ngx_http_request_t *r)
{
    u_char                *last, *location;
    size_t                 root, len;
    ngx_str_t              path;
    ngx_int_t              rc;
    ngx_uint_t             level;
    ngx_log_t              *log;
    ngx_buf_t              *b;
    ngx_chain_t             out;
    ngx_open_file_info_t   of;
    ngx_http_core_loc_conf_t *clcf;

    if (!(r->method & (NGX_HTTP_GET|NGX_HTTP_HEAD|NGX_HTTP_POST))) {
        return NGX_HTTP_NOT_ALLOWED;
    }

    if (r->uri.data[r->uri.len - 1] == '/') {
        return NGX_DECLINED;
    }

    log = r->connection->log;

```

```

/*
 * ngx_http_map_uri_to_path() allocates memory for terminating '\0'
 * so we do not need to reserve memory for '/' for possible redirect
 */

last = ngx_http_map_uri_to_path(r, &path, &root, 0);
if (last == NULL) {
    return NGX_HTTP_INTERNAL_SERVER_ERROR;
}

path.len = last - path.data;

ngx_log_debug1(NGX_LOG_DEBUG_HTTP, log, 0,
               "http filename: \"%s\"", path.data);

clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

ngx_memzero(&of, sizeof(ngx_open_file_info_t));

of.read_ahead = clcf->read_ahead;
of.directio = clcf->directio;
of.valid = clcf->open_file_cache_valid;
of.min_uses = clcf->open_file_cache_min_uses;
of.errors = clcf->open_file_cache_errors;
of.events = clcf->open_file_cache_events;

if (ngx_http_set_disable_symlinks(r, clcf, &path, &of) != NGX_OK) {
    return NGX_HTTP_INTERNAL_SERVER_ERROR;
}

if (ngx_open_cached_file(clcf->open_file_cache, &path, &of, r->pool)
    != NGX_OK)
{
    switch (of.err) {

        case 0:
            return NGX_HTTP_INTERNAL_SERVER_ERROR;

        case NGX_ENOENT:
        case NGX_ENOTDIR:
        case NGX_ENAMETOOLONG:

            level = NGX_LOG_ERR;
            rc = NGX_HTTP_NOT_FOUND;
            break;

        case NGX_EACCES:
#ifdef NGX_HAVE_OPENAT

```

```

        case NGX_EMLINK:
        case NGX_ELOOP:

#endif

        level = NGX_LOG_ERR;
        rc = NGX_HTTP_FORBIDDEN;
        break;

    default:

        level = NGX_LOG_CRIT;
        rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
        break;
    }

    if (rc != NGX_HTTP_NOT_FOUND || clcf->log_not_found) {
        ngx_log_error(level, log, of.err,
                      "%s \"%s\" failed", of.failed,
path.data);
    }

    return rc;
}

r->root_tested = !r->error_page;

ngx_log_debug1(NGX_LOG_DEBUG_HTTP, log, 0, "http static fd: %d", of.fd);

if (of.is_dir) {

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, log, 0, "http dir");

    ngx_http_clear_location(r);

    r->headers_out.location = ngx_palloc(r->pool,
sizeof(ngx_table_elt_t));
    if (r->headers_out.location == NULL) {
        return NGX_HTTP_INTERNAL_SERVER_ERROR;
    }

    len = r->uri.len + 1;

    if (!clcf->alias && clcf->root_lengths == NULL && r->args.len == 0)
{
        location = path.data + clcf->root.len;

        *last = '/';

```

```

    } else {
        if (r->args.len) {
            len += r->args.len + 1;
        }

        location = ngx_pnalloc(r->pool, len);
        if (location == NULL) {
            return NGX_HTTP_INTERNAL_SERVER_ERROR;
        }

        last = ngx_copy(location, r->uri.data, r->uri.len);

        *last = '/';

        if (r->args.len) {
            *++last = '?';
            ngx_memcpy(++last, r->args.data, r->args.len);
        }
    }

    /*
     * we do not need to set the r->headers_out.location->hash and
     * r->headers_out.location->key fields
     */

    r->headers_out.location->value.len = len;
    r->headers_out.location->value.data = location;

    return NGX_HTTP_MOVED_PERMANENTLY;
}

#if !(NGX_WIN32) /* the not regular files are probably Unix specific */

    if (!of.is_file) {
        ngx_log_error(NGX_LOG_CRIT, log, 0,
            "\"%s\" is not a regular file", path.data);

        return NGX_HTTP_NOT_FOUND;
    }

#endif

    if (r->method & NGX_HTTP_POST) {
        return NGX_HTTP_NOT_ALLOWED;
    }

    rc = ngx_http_discard_request_body(r);

```

```

if (rc != NGX_OK) {
    return rc;
}

log->action = "sending response to client";

r->headers_out.status = NGX_HTTP_OK;
r->headers_out.content_length_n = of.size;
r->headers_out.last_modified_time = of.mtime;

if (ngx_http_set_content_type(r) != NGX_OK) {
    return NGX_HTTP_INTERNAL_SERVER_ERROR;
}

if (r != r->main && of.size == 0) {
    return ngx_http_send_header(r);
}

r->allow_ranges = 1;

/* we need to allocate all before the header would be sent */

b = ngx_palloc(r->pool, sizeof(ngx_buf_t));
if (b == NULL) {
    return NGX_HTTP_INTERNAL_SERVER_ERROR;
}

b->file = ngx_palloc(r->pool, sizeof(ngx_file_t));
if (b->file == NULL) {
    return NGX_HTTP_INTERNAL_SERVER_ERROR;
}

rc = ngx_http_send_header(r);

if (rc == NGX_ERROR || rc > NGX_OK || r->header_only) {
    return rc;
}

b->file_pos = 0;
b->file_last = of.size;

b->in_file = b->file_last ? 1: 0;
b->last_buf = (r == r->main) ? 1: 0;
b->last_in_chain = 1;

b->file->fd = of.fd;
b->file->name = path;
b->file->log = log;

```

```
b->file->directio = of.is_directio;

out.buf = b;
out.next = NULL;

return ngx_http_output_filter(r, &out);
}
```

首先是检查客户端的 http 请求类型（`r->method`），如果请求类型为 `NGX_HTTP_GET|NGX_HTTP_HEAD|NGX_HTTP_POST`，则继续进行处理，否则一律返回 `NGX_HTTP_NOT_ALLOWED` 从而拒绝客户端的发起的请求。

其次是检查请求的 url 的结尾字符是不是斜杠 `'/'`，如果是说明请求的不是一个文件，给后续的 handler 去处理，比如后续的 `ngx_http_autoindex_handler`（如果是请求的是一个目录下面，可以列出这个目录的文件），或者是 `ngx_http_index_handler`（如果请求的路径下面有个默认的文件 `index`，直接返回 `index` 文件的内容）。

然后接下来调用了一个 `ngx_http_map_uri_to_path` 函数，该函数的作用是把请求的 http 协议的路径转化成一个文件系统的路径。

然后根据转化出来的具体路径，去打开文件，打开文件的时候做了 2 种检查，一种是，如果请求的文件是个 `symbol link`，根据配置，是否允许符号链接，不允许返回错误。还有一个检查是，如果请求的是一个名称，是一个目录的名字，也返回错误。如果都没有错误，就读取文件，返回内容。其实说返回内容可能不是特别准确，比较准确的说法是，把产生的内容传递给后续的 `filter` 去处理。

http log module

该模块提供了对于每一个 http 请求进行记录的功能，也就是我们见到的 `access.log`。当然这个模块对于 log 提供了一些配置指令，使得可以比较方便的定制 `access.log`。

这个模块的代码位于 `src/http/modules/nginx_http_log_module.c`，虽然这个模块的代码有接近 1400 行，但是主要的逻辑在于对日志本身格式啊，等细节的处理。我们在这里进行分析主要是关注，如何编写一个 log handler 的问题。

由于 log handler 的时候，拿到的参数也是 request 这个东西，那么也就意味着我们如果需要，可以好好研究下这个结构，把我们需要的所有信息都记录下来。

对于 log handler，有一点特别需要注意的就是，log handler 是无论如何都会被调用的，就是只要服务端接受到了一个客户端的请求，也就是产生了一个 request 对象，那么这些个 log handler 的处理函数都会被调用的，就是在释放 request 的时候被调用的（ngx_http_free_request 函数）。

那么当然绝对不能忘记的就是 log handler 最好，也是建议被挂载在 NGX_HTTP_LOG_PHASE 阶段。因为挂载在其他阶段，有可能在某些情况下被跳过，而没有执行到，导致你的 log 模块记录的信息不全。

还有一点要说明的是，由于 nginx 是允许在某个阶段有多个 handler 模块存在的，根据其处理结果，确定是否要调用下一个 handler。但是对于挂载在 NGX_HTTP_LOG_PHASE 阶段的 handler，则根本不关注这里 handler 的具体处理函数的返回值，所有的都被调用。如下，位于 src/http/nginx_http_request.c 中的 ngx_http_log_request 函数。

```
static void
ngx_http_log_request(ngx_http_request_t *r)
{
    ngx_uint_t          i, n;
    ngx_http_handler_pt *log_handler;
    ngx_http_core_main_conf_t *cmcf;

    cmcf = ngx_http_get_module_main_conf(r, ngx_http_core_module);

    log_handler = cmcf->phases[NGX_HTTP_LOG_PHASE].handlers.elts;
    n = cmcf->phases[NGX_HTTP_LOG_PHASE].handlers.nelts;

    for (i = 0; i < n; i++) {
        log_handler[i](r);
    }
}
```

过滤模块 (90%)

过滤模块简介 (90%)

执行时间和内容 (90%)

过滤（filter）模块是过滤响应头和内容的模块，可以对回复的头和内容进行处理。它的处理时间在获取回复内容之后，向用户发送响应之前。它的处理过程分为两个阶段，过滤 HTTP 回复的头部和主体，在这两个阶段可以分别对头部和主体进行修改。

在代码中有类似的函数：

```
ngx_http_top_header_filter(r);
ngx_http_top_body_filter(r, in);
```

就是分别对头部和主体进行过滤的函数。所有模块的响应内容要返回给客户端，都必须调用这两个接口。

执行顺序 (90%)

过滤模块的调用是有顺序的，它的顺序在编译的时候就决定了。控制编译的脚本位于 `auto/modules` 中，当你编译完 Nginx 以后，可以在 `objs` 目录下面看到一个 `ngx_modules.c` 的文件。打开这个文件，有类似的代码：

```
ngx_module_t *ngx_modules[] = {
    ...
    &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_ssi_filter_module,
    &ngx_http_charset_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
};
```

从 `write_filter` 到 `not_modified_filter`，模块的执行顺序是反向的。也就是说最早执行的是 `not_modified_filter`，然后各个模块依次执行。所有第三方的模块只能加入到 `copy_filter` 和 `headers_filter` 模块之间执行。

Nginx 执行的时候是怎么按照次序依次来执行各个过滤模块呢？它采用了一种很隐晦的方法，即通过局部的全局变量。比如，在每个 filter 模块，很可能看到如下代码：

```
static ngx_http_output_header_filter_pt  ngx_http_next_header_filter;
static ngx_http_output_body_filter_pt    ngx_http_next_body_filter;

...

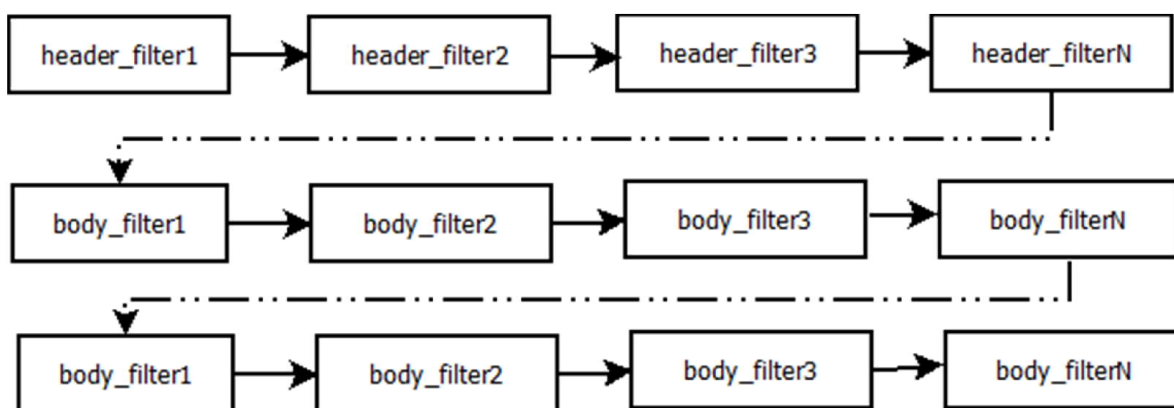
ngx_http_next_header_filter = ngx_http_top_header_filter;
ngx_http_top_header_filter = ngx_http_example_header_filter;

ngx_http_next_body_filter = ngx_http_top_body_filter;
ngx_http_top_body_filter = ngx_http_example_body_filter;
```

ngx_http_top_header_filter 是一个全局变量。当编译进一个 filter 模块的时候，就被赋值为当前 filter 模块的处理函数。而 ngx_http_next_header_filter 是一个局部全局变量，它保存了编译前上一个 filter 模块的处理函数。所以整体看来，就像用全局变量组成的一条单向链表。

每个模块想执行下一个过滤函数，只要调用一下 ngx_http_next_header_filter 这个局部变量。而整个过滤模块链的入口，需要调用 ngx_http_top_header_filter 这个全局变量。ngx_http_top_body_filter 的行为与 header filter 类似。

响应头和响应体过滤函数的执行顺序如下所示：



这图只表示了 head_filter 和 body_filter 之间的执行顺序，在 header_filter 和 body_filter 处理函数之间，在 body_filter 处理函数之间，可能还有其他执行代码。

模块编译 (90%)

Nginx 可以方便的加入第三方的过滤模块。在过滤模块的目录里，首先需要加入 config 文件，文件的内容如下：

```
ngx_addon_name=ngx_http_example_filter_module
HTTP_AUX_FILTER_MODULES="$HTTP_AUX_FILTER_MODULES
ngx_http_example_filter_module"
NGX_ADDON_SRCS="$NGX_ADDON_SRCS $ngx_addon_dir/ngx_http_example_filter_module.c"
```

说明把这个名为 ngx_http_example_filter_module 的过滤模块加入，

ngx_http_example_filter_module.c 是该模块的源代码。

注意 HTTP_AUX_FILTER_MODULES 这个变量与一般的内容处理模块不同。

过滤模块的分析 (90%)

相关结构体 (90%)

ngx_chain_t 结构非常简单，是一个单向链表：

```
typedef struct ngx_chain_s ngx_chain_t;

struct ngx_chain_s {
    ngx_buf_t      *buf;
    ngx_chain_t    *next;
};
```

在过滤模块中，所有输出的内容都是通过一条单向链表所组成。这种单向链表的设计，正好应和了 Nginx 流式的输出模式。每次 Nginx 都是读到一部分的内容，就放到链表，然后输出出去。这种设计的好处是简单，非阻塞，但是相应的问题就是跨链表的内容操作非常麻烦，如果需要跨链表，很多时候都只能缓存链表的内容。

单链表负载的就是 ngx_buf_t，这个结构体使用非常广泛，先让我们看下该结构体的代码：

```
struct ngx_buf_s {
    u_char          *pos;          /* 当前 buffer 真实内容的起始位置 */
    u_char          *last;         /* 当前 buffer 真实内容的结束位置 */
    off_t           file_pos;      /* 在文件中真实内容的起始位置 */
    off_t           file_last;     /* 在文件中真实内容的结束位置 */

    u_char          *start;        /* buffer 内存的开始分配的位置 */
    u_char          *end;          /* buffer 内存的结束分配的位置 */
    ngx_buf_tag_t   tag;          /* buffer 属于哪个模块的标志 */
};
```

```

ngx_file_t      *file;      /* buffer 所引用的文件 */

/* 用来引用替换过后的 buffer，以便当所有 buffer 输出以后，
 * 这个影子 buffer 可以被释放。
 */
ngx_buf_t      *shadow;

/* the buf's content could be changed */
unsigned        temporary:1;

/*
 * the buf's content is in a memory cache or in a read only memory
 * and must not be changed
 */
unsigned        memory:1;

/* the buf's content is mmap()ed and must not be changed */
unsigned        mmap:1;

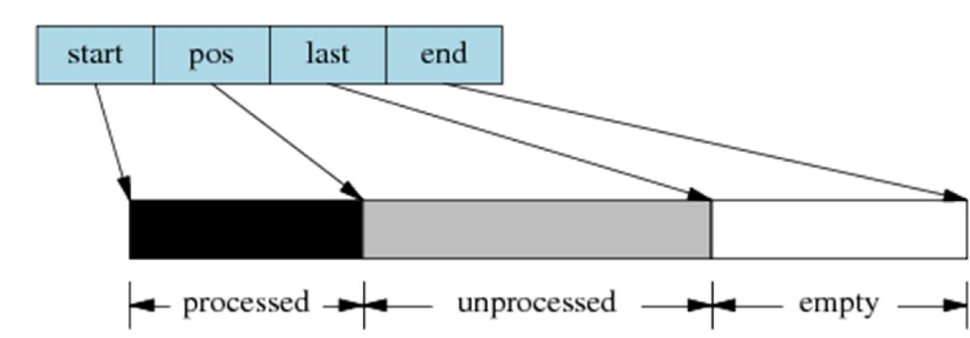
unsigned        recycled:1; /* 内存可以被输出并回收 */
unsigned        in_file:1; /* buffer 的内容在文件中 */
/* 马上全部输出 buffer 的内容，gzip 模块里面用得比较多 */
unsigned        flush:1;
/* 基本上是一段输出链的最后一个 buffer 带的标志，标示可以输出，
 * 有些零长度的 buffer 也可以置该标志
 */
unsigned        sync:1;
/* 所有请求里面最后一块 buffer，包含子请求 */
unsigned        last_buf:1;
/* 当前请求输出链的最后一块 buffer */
unsigned        last_in_chain:1;
/* shadow 链里面的最后 buffer，可以释放 buffer 了 */
unsigned        last_shadow:1;
/* 是否是暂存文件 */
unsigned        temp_file:1;

/* 统计用，表示使用次数 */
/* STUB */ int    num;
};

```

一般 buffer 结构体可以表示一块内存，内存的起始和结束地址分别用 start 和 end 表示，pos 和 last 表示实际的内容。如果内容已经处理过了，pos 的位置就可以往后移动。如果读取到新的内容，last 的位置就会往后移动。所以 buffer 可以在多次调用过程中使用。如果 last 等于 end，就说明这块内存已

经用完了。如果 pos 等于 last，说明内存已经处理完了。下面是一个简单的示意图，说明 buffer 中指针的用法：



响应头过滤函数 (90%)

响应头过滤函数主要的用处就是处理 HTTP 响应的头，可以根据实际情况对于响应头进行修改或者添加删除。响应头过滤函数先于响应体过滤函数，而且只调用一次，所以一般可作过滤模块的初始化工作。

响应头过滤函数的入口只有一个：

```
ngx_int_t
ngx_http_send_header(ngx_http_request_t *r)
{
    ...

    return ngx_http_top_header_filter(r);
}
```

该函数向客户端发送回复的时候调用，然后按前一节所述的执行顺序。该函数的返回值一般是 NGX_OK，NGX_ERROR 和 NGX_AGAIN，分别表示处理成功，失败和未完成。

你可以把 HTTP 响应头的存储方式想象成一个 hash 表，在 Nginx 内部可以很方便地查找和修改各个响应头部，ngx_http_header_filter_module 过滤模块把所有的 HTTP 头组合成一个完整的 buffer，最终 ngx_http_write_filter_module 过滤模块把 buffer 输出。

按照前一节过滤模块的顺序，依次讲解如下：

filter module	description
ngx_http_not_modified_filter_module	默认打开，如果请求的 if-modified-since 等于回复的 last-modified 间值，说明回复没有变化，清空所有回复的内容，返回 304。
ngx_http_range_body_filter_module	默认打开，只是响应体过滤函数，支持 range 功能，如果请求包含 range 请求，那就只发送 range 请求的一段内容。
ngx_http_copy_filter_module	始终打开，只是响应体过滤函数， 主要工作是把文件中内容读到内存中，以便

filter module	description
	进行处理。
ngx_http_headers_filter_module	始终打开，可以设置 expire 和 Cache-control 头，可以添加任意名称的头
ngx_http_userid_filter_module	默认关闭，可以添加统计用的识别用户的 cookie。
ngx_http_charset_filter_module	默认关闭，可以添加 charset，也可以将内容从一种字符集转换到另外一种字符集，不支持多字节字符集。
ngx_http_ssi_filter_module	默认关闭，过滤 SSI 请求，可以发起子请求，去获取 include 进来的文件
ngx_http_postpone_filter_module	始终打开，用来将子请求和主请求的输出链合并
ngx_http_gzip_filter_module	默认关闭，支持流式的压缩内容
ngx_http_range_header_filter_module	默认打开，只是响应头过滤函数，用来解析 range 头，并产生 range 响应的头。
ngx_http_chunked_filter_module	默认打开，对于 HTTP/1.1 和缺少 content-length 的回复自动打开。
ngx_http_header_filter_module	始终打开，用来将所有 header 组成一个完整的 HTTP 头。
ngx_http_write_filter_module	始终打开，将输出链拷贝到 r->out 中，然后输出内容。

响应体过滤函数 (90%)

响应体过滤函数是过滤响应主体的函数。ngx_http_top_body_filter 这个函数每个请求可能会被执行多次，它的入口函数是 ngx_http_output_filter，比如：

```
ngx_int_t
ngx_http_output_filter(ngx_http_request_t *r, ngx_chain_t *in)
{
    ngx_int_t      rc;
    ngx_connection_t *c;

    c = r->connection;

    rc = ngx_http_top_body_filter(r, in);

    if (rc == NGX_ERROR) {
        /* NGX_ERROR may be returned by any filter */
        c->error = 1;
    }

    return rc;
}
```

ngx_http_output_filter 可以被一般的静态处理模块调用，也有可能是在 upstream 模块里面被调用，对于整个请求的处理阶段来说，他们处于的用处都是一样的，就是把响应内容过滤，然后发给客户端。

具体模块的响应体过滤函数的格式类似这样：

```
static int
ngx_http_example_body_filter(ngx_http_request_t *r, ngx_chain_t *in)
{
```

```
...  
  
    return ngx_http_next_body_filter(r, in);  
}
```

该函数的返回值一般是 `NGX_OK`, `NGX_ERROR` 和 `NGX_AGAIN`, 分别表示处理成功, 失败和未完成。

主要功能介绍 (90%)

响应的主体内容就存于单链表 `in`, 链表一般不会太长, 有时 `in` 参数可能为 `NULL`。 `in` 中存有 `buf` 结构体中, 对于静态文件, 这个 `buf` 大小默认是 32K; 对于反向代理的应用, 这个 `buf` 可能是 4k 或者 8k。 为了保持内存的低消耗, Nginx 一般不会分配过大的内存, 处理的原则是收到一定的数据, 就发送出去。 一个简单的例子, 可以看看 Nginx 的 `chunked_filter` 模块, 在没有 `content-length` 的情况下, `chunk` 模块可以流式 (`stream`) 的加上长度, 方便浏览器接收和显示内容。

在响应体过滤模块中, 尤其要注意的是 `buf` 的标志位, 完整描述可以在“相关结构体”这个节中看到。 如果 `buf` 中包含 `last` 标志, 说明是最后一块 `buf`, 可以直接输出并结束请求了。 如果有 `flush` 标志, 说明这块 `buf` 需要马上输出, 不能缓存。 如果整块 `buffer` 经过处理完以后, 没有数据了, 你可以把 `buffer` 的 `sync` 标志置上, 表示只是同步的用处。

当所有的过滤模块都处理完毕时, 在最后的 `write_filtrer` 模块中, Nginx 会将 `in` 输出链拷贝到 `r->out` 输出链的末尾, 然后调用 `sendfile` 或者 `writetv` 接口输出。 由于 Nginx 是非阻塞的 `socket` 接口, 写操作并不一定会成功, 可能会有部分数据还残存在 `r->out`。 在下次的调用中, Nginx 会继续尝试发送, 直至成功。

发出子请求 (90%)

Nginx 过滤模块一大特色就是可以发出子请求, 也就是在过滤响应内容的时候, 你可以发送新的请求, Nginx 会根据你调用的先后顺序, 将多个回复的内容拼接成正常的响应主体。 一个简单的例子可以参考 `addition` 模块。

Nginx 是如何保证父请求和子请求的顺序呢？当 Nginx 发出子请求时，就会调用 ngx_http_subrequest 函数，将子请求插入父请求的 r->postponed 链表中。子请求会在主请求执行完毕时获得依次调用。子请求同样会有一个请求所有的生存期和处理过程，也会进入过滤模块流程。

关键点是在 postpone_filter 模块中，它会拼接主请求和子请求的响应内容。r->postponed 按次序保存有父请求和子请求，它是一个链表，如果前面一个请求未完成，那后一个请求内容就不会输出。当前一个请求完成时并输出时，后一个请求才可输出，当所有的子请求都完成时，所有的响应内容也就输出完毕了。

一些优化措施 (90%)

Nginx 过滤模块涉及到的结构体，主要就是 chain 和 buf，非常简单。在日常的过滤模块中，这两类结构使用非常频繁，Nginx 采用类似 freelist 重复利用的原则，将使用完毕的 chain 或者 buf 结构体，放置到一个固定的空闲链表里，以待下次使用。

比如，在通用内存池结构体中，pool->chain 变量里面就保存着释放的 chain。而一般的 buf 结构体，没有模块间公用的空闲链表池，都是保存在各模块的缓存空闲链表池里面。对于 buf 结构体，还有一种 busy 链表，表示该链表中的 buf 都处于输出状态，如果 buf 输出完毕，这些 buf 就可以释放并重复利用了。

功能	函数名
chain 分配	ngx_alloc_chain_link
chain 释放	ngx_free_chain
buf 分配	ngx_chain_get_free_buf
buf 释放	ngx_chain_update_chains

过滤内容的缓存 (90%)

由于 Nginx 设计流式的输出结构，当我们需要对响应内容作全文过滤的时候，必须缓存部分的 buf 内容。该类过滤模块往往比较复杂，比如 sub，ssi，gzip 等模块。这类模块的设计非常灵活，我简单讲一下设计原则：

1. 输入链 in 需要拷贝操作，经过缓存的过滤模块，输入输出链往往已经完全不一样了，所以需要拷贝，通过 ngx_chain_add_copy 函数完成。
2. 一般有自己的 free 和 busy 缓存链表池，可以提高 buf 分配效率。
3. 如果需要分配大块内容，一般分配固定大小的内存卡，并设置 recycled 标志，表示可以重复利用。
4. 原有的输入 buf 被替换缓存时，必须将其 buf->pos 设为 buf->last，表明原有的 buf 已经被输出完毕。或者在新建立的 buf，将 buf->shadow 指向旧的 buf，以便输出完毕时及时释放旧的 buf。

upstream 模块

upstream 模块 (100%)

nginx 模块一般被分成三大类: handler、filter 和 upstream。前面的章节中,读者已经了解了 handler、filter。利用这两类模块,可以使 nginx 轻松完成任何单机工作。而本章介绍的 upstream 模块,将使 nginx 跨越单机的限制,完成网络数据的接收、处理和转发。

数据转发功能,为 nginx 提供了跨越单机的横向处理能力,使 nginx 摆脱只能为终端节点提供单一功能的限制,而使它具备了网路应用级别的拆分、封装和整合的战略功能。在云模型大行其道的今天,数据转发是 nginx 有能力构建一个网络应用的关键组件。当然,鉴于开发成本的问题,一个网络应用的关键组件一开始往往会采用高级编程语言开发。但是当系统到达一定规模,并且需要更重视性能的时候,为了达到所要求的性能目标,高级语言开发出的组件必须进行结构化修改。此时,对于修改代价而言,nginx 的 upstream 模块呈现出极大的吸引力,因为它天生就快。作为附带,nginx 的配置系统提供的层次化和松耦合使得系统的扩展性也达到比较高的程度。

言归正传,下面介绍 upstream 的写法。

upstream 模块接口

从本质上说,upstream 属于 handler,只是他不产生自己的内容,而是通过请求后端服务器得到内容,所以才称为 upstream(上游)。请求并取得响应内容的整个过程已经被封装到 nginx 内部,所以 upstream 模块只需要开发若干回调函数,完成构造请求和解析响应等具体的工作。

这些回调函数如下表所示:

create_request	生成发送到后端服务器的请求缓冲(缓冲链),在初始化 upstream 时使用。
reinit_request	在某台后端服务器出错的情况,nginx 会尝试另一台后端服务器。nginx 选定新的服务器以后,会先调用此函数,以重新初始化 upstream 模块的工作状态,然后再次进行 upstream 连接。
process_header	处理后端服务器返回的信息头部。所谓头部是与 upstream server 通信的协议规定的,比如 HTTP 协议的 header 部分,或者 memcached 协议的响应状态部分。
abort_request	在客户端放弃请求时被调用。不需要在函数中实现关闭后端服务器连接的功能,系统会自动完成关闭连接的步骤,所以一般此函数不会进行任何具体工作。
finalize_request	正常完成与后端服务器的请求后调用该函数,与 abort_request 相同,一般也不会进行任何具体工作。
input_filter	处理后端服务器返回的响应正文。nginx 默认的 input_filter 会将收到的内容封装成为缓冲区链 ngx_chain。该链由 upstream 的 out_bufs 指针域定位,所以开发人员可以在模块以外通过该指针得

	到后端服务器返回的正文数据。memcached 模块实现了自己的 input_filter，在后面会具体分析这个模块。
input_filter_init	初始化 input filter 的上下文。nginx 默认的 input_filter_init 直接返回。

memcached 模块分析

memcache 是一款高性能的分布式 cache 系统，得到了非常广泛的应用。memcache 定义了一套私有通信协议，使得不能通过 HTTP 请求来访问 memcache。但协议本身简单高效，而且 memcache 使用广泛，所以大部分现代开发语言和平台都提供了 memcache 支持，方便开发者使用 memcache。

nginx 提供了 ngx_http_memcached 模块，提供从 memcache 读取数据的功能，而不提供向 memcache 写数据的功能。作为 web 服务器，这种设计是可以接受的。

下面，我们开始分析 ngx_http_memcached 模块，一窥 upstream 的奥秘。

Handler 模块？

初看 memcached 模块，大家可能觉得并无特别之处。如果稍微细看，甚至觉得有点像 handler 模块，当大家看到这段代码以后，必定疑惑为什么会跟 handler 模块一模一样。

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

因为 upstream 模块使用的就是 handler 模块的接入方式。同时，upstream 模块的指令系统的设计也是遵循 handler 模块的基本规则：配置该模块才会执行该模块。

```
{ ngx_string("memcached_pass"),
  NGX_HTTP_LOC_CONF|NGX_HTTP_LIF_CONF|NGX_CONF_TAKE1,
  ngx_http_memcached_pass,
  NGX_HTTP_LOC_CONF_OFFSET,
  0,
  NULL }
```

所以大家觉得眼熟是好事，说明大家对 Handler 的写法已经很熟悉了。

Upstream 模块！

那么，upstream 模块的特别之处究竟在哪里呢？答案是就在模块处理函数的实现中。upstream 模块的处理函数进行的操作都包含一个固定的流程。在 memcached 的例子中，可以观察 ngx_http_memcached_handler 的代码，可以发现，这个固定的操作流程是：

1. 创建 upstream 数据结构。

```
if (ngx_http_upstream_create(r) != NGX_OK) {  
    return NGX_HTTP_INTERNAL_SERVER_ERROR;  
}
```

2. 设置模块的 tag 和 schema。schema 现在只会用于日志，tag 会用于 buf_chain 管理。

```
u = r->upstream;  
  
ngx_str_set(&u->schema, "memcached://");  
u->output.tag = (ngx_buf_tag_t) &ngx_http_memcached_module;
```

3. 设置 upstream 的后端服务器列表数据结构。

```
mlcf = ngx_http_get_module_loc_conf(r, ngx_http_memcached_module);  
u->conf = &mlcf->upstream;
```

4. 设置 upstream 回调函数。在这里列出的代码稍稍调整了代码顺序。

```
u->create_request = ngx_http_memcached_create_request;  
u->reinit_request = ngx_http_memcached_reinit_request;  
u->process_header = ngx_http_memcached_process_header;  
u->abort_request = ngx_http_memcached_abort_request;  
u->finalize_request = ngx_http_memcached_finalize_request;  
u->input_filter_init = ngx_http_memcached_filter_init;  
u->input_filter = ngx_http_memcached_filter;
```

5. 创建并设置 upstream 环境数据结构。

```
ctx = ngx_palloc(r->pool, sizeof(ngx_http_memcached_ctx_t));  
if (ctx == NULL) {  
    return NGX_HTTP_INTERNAL_SERVER_ERROR;  
}  
  
ctx->rest = NGX_HTTP_MEMCACHED_END;  
ctx->request = r;
```

```
ngx_http_set_ctx(r, ctx, ngx_http_memcached_module);  
  
u->input_filter_ctx = ctx;
```

6. 完成 upstream 初始化并进行收尾工作。

```
r->main->count++;  
ngx_http_upstream_init(r);  
return NGX_DONE;
```

任何 upstream 模块，简单如 memcached，复杂如 proxy、fastcgi 都是如此。不同的 upstream 模块在这 6 步中的最大差别会出现在第 2、3、4、5 上。其中第 2、4 两步很容易理解，不同的模块设置的标志和使用的回调函数肯定不同。第 5 步也不难理解，只有第 3 步是最为晦涩的，不同的模块在取得后端服务器列表时，策略的差异非常大，有如 memcached 这样简单明了的，也有如 proxy 那样逻辑复杂的。这个问题先记下来，等把 memcached 剖析清楚了，再单独讨论。

第 6 步是一个常态。将 count 加 1，然后返回 NGX_DONE。nginx 遇到这种情况，虽然会认为当前请求的处理已经结束，但是不会释放请求使用的内存资源，也不会关闭与客户端的连接。之所以需要这样，是因为 nginx 建立了 upstream 请求和客户端请求之间一对一的关系，在后续使用 ngx_event_pipe 将 upstream 响应发送回客户端时，还要使用到这些保存着客户端信息的数据结构。这部分会在后面的原理篇做具体介绍，这里不再展开。

将 upstream 请求和客户端请求进行一对一绑定，这个设计有优势也有缺陷。优势就是简化模块开发，可以将精力集中在模块逻辑上，而缺陷同样明显，一对一的设计很多时候都不能满足复杂逻辑的需要。对于这一点，将会在后面的原理篇来阐述。

回调函数

前面剖析了 memcached 模块的骨架，现在开始逐个解决每个回调函数。

1. ngx_http_memcached_create_request: 很简单的按照设置的内容生成一个 key，接着生成一个“get \$key”的请求，放在 r->upstream->request_bufs 里面。

2. ngx_http_memcached_reinit_request: 无需初始化。

3. ngx_http_memcached_abort_request: 无需额外操作。

4. ngx_http_memcached_finalize_request: 无需额外操作。

5. ngx_http_memcached_process_header: 模块的业务重点函数。memcache 协议的头部信息被定义为第一行文本，可以找到这段代码证明：

```
for (p = u->buffer.pos; p < u->buffer.last; p++) {  
    if ( * p == LF) {  
        goto found;  
    }  
}
```

如果在已读入缓冲的数据中没有发现 LF('n')字符，函数返回 NGX_AGAIN，表示头部未完全读入，需要继续读取数据。nginx 在收到新的数据以后会再次调用该函数。

nginx 处理后端服务器的响应头时只会使用一块缓存，所有数据都在这块缓存中，所以解析头部信息时不需要考虑头部信息跨越多块缓存的情况。而如果头部过大，不能保存在这块缓存中，nginx 会返回错误信息给客户端，并记录 error log，提示缓存不够大。

process_header 的重要职责是将后端服务器返回的状态翻译成返回给客户端的状态。例如，在 ngx_http_memcached_process_header 中，有这样几段代码：

```
r->headers_out.content_length_n = ngx_atoof(len, p - len - 1);  
  
u->headers_in.status_n = 200;  
u->state->status = 200;  
  
u->headers_in.status_n = 404;  
u->state->status = 404;
```

u->state 用于计算 upstream 相关的变量。比如 u->state->status 将被用于计算变量“upstream_status”的值。u->headers_in 将被作为返回给客户端的响应返回状态码。而第一行则是设置返回给客户端的响应的长度。

在这个函数中不能忘记的一件事情是处理完头部信息以后需要将读指针 pos 后移，否则这段数据也将被复制到返回给客户端的响应的正文中，进而导致正文内容不正确。

```
u->buffer.pos = p + 1;
```

`process_header` 函数完成响应头的正确处理，应该返回 `NGX_OK`。如果返回 `NGX_AGAIN`，表示未读取完整数据，需要从后端服务器继续读取数据。返回 `NGX_DECLINED` 无意义，其他任何返回值都被认为是出错状态，`nginx` 将结束 `upstream` 请求并返回错误信息。

6. `ngx_http_memcached_filter_init`: 修正从后端服务器收到的内容长度。因为在处理 `header` 时没有加上这部分长度。

7. `ngx_http_memcached_filter`: `memcached` 模块是少有的带有处理正文的回调函数的模块。因为 `memcached` 模块需要过滤正文末尾 `CRLF "END" CRLF`，所以实现了自己的 `filter` 回调函数。处理正文的实际意义是将从后端服务器收到的正文有效内容封装成 `ngx_chain_t`，并加在 `u->out_bufs` 末尾。`nginx` 并不进行数据拷贝，而是建立 `ngx_buf_t` 数据结构指向这些数据内存区，然后由 `ngx_chain_t` 组织这些 `buf`。这种实现避免了内存大量搬迁，也是 `nginx` 高效的奥秘之一。

本节回顾

这一节介绍了 `upstream` 模块的基本组成。`upstream` 模块是从 `handler` 模块发展而来，指令系统和模块生效方式与 `handler` 模块无异。不同之处在于，`upstream` 模块在 `handler` 函数中设置众多回调函数。实际工作都是由这些回调函数完成的。每个回调函数都是在 `upstream` 的某个固定阶段执行，各司其职，大部分回调函数一般不会真正用到。`upstream` 最重要的回调函数是 `create_request`、`process_header` 和 `input_filter`，他们共同实现了与后端服务器的协议的解析部分。

负载均衡模块 (100%)

负载均衡模块用于从“`upstream`”指令定义的后端主机列表中选取一台主机。`nginx` 先使用负载均衡模块找到一台主机，再使用 `upstream` 模块实现与这台主机的交互。为了方便介绍负载均衡模块，做到言之有物，以下选取 `nginx` 内置的 `ip hash` 模块作为实际例子进行分析。

配置

要了解负载均衡模块的开发方法，首先需要了解负载均衡模块的使用方法。因为负载均衡模块与之前书中提到的模块差别比较大，所以我们从配置入手比较容易理解。

在配置文件中，我们如果需要使用 ip hash 的负载均衡算法。我们需要写一个类似下面的配置：

```
upstream test {
    ip_hash;

    server 192.168.0.1;
    server 192.168.0.2;
}
```

从配置我们可以看出负载均衡模块的使用场景： 1. 核心指令“ip_hash”只能在 upstream {}中使用。这条指令用于通知 nginx 使用 ip hash 负载均衡算法。如果没加这条指令，nginx 会使用默认的 round robin 负载均衡模块。请各位读者对比 handler 模块的配置，是不是有共同点？ 2. upstream {}中的指令可能出现在“server”指令前，可能出现在“server”指令后，也可能出现在两条“server”指令之间。各位读者可能会有疑问，有什么差别么？那么请各位读者尝试下面这个配置：

```
upstream test {
    server 192.168.0.1 weight=5;
    ip_hash;
    server 192.168.0.2 weight=7;
}
```

神奇的事情出现了：

```
nginx: [emerg] invalid parameter "weight=7" in nginx.conf:103
configuration file nginx.conf test failed
```

可见 ip_hash 指令的确能影响到配置的解析。

指令

配置决定指令系统，现在就来看 ip_hash 的指令定义：

```
static ngx_command_t  ngx_http_upstream_ip_hash_commands[] = {

    { ngx_string("ip_hash"),
      NGX_HTTP_UPS_CONF|NGX_CONF_NOARGS,
      ngx_http_upstream_ip_hash,
      0,
      0,
      NULL },

    ngx_null_command
```

```
};
```

没有特别的东西，除了指令属性是 `NGX_HTTP_UPS_CONF`。这个属性表示该指令的适用范围是 `upstream{}`。

钩子

以从前面的章节得到的经验，大家应该知道这里就是模块的切入点了。负载均衡模块的钩子代码都是有规律的，这里通过 `ip_hash` 模块来分析这个规律。

```
static char *
ngx_http_upstream_ip_hash(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
{
    ngx_http_upstream_srv_conf_t  *uscf;

    uscf = ngx_http_conf_get_module_srv_conf(cf, ngx_http_upstream_module);

    uscf->peer.init_upstream = ngx_http_upstream_init_ip_hash;

    uscf->flags = NGX_HTTP_UPSTREAM_CREATE
                 |NGX_HTTP_UPSTREAM_MAX_FAILS
                 |NGX_HTTP_UPSTREAM_FAIL_TIMEOUT
                 |NGX_HTTP_UPSTREAM_DOWN;

    return NGX_CONF_OK;
}
```

这段代码中有两点值得我们注意。一个是 `uscf->flags` 的设置，另一个是设置 `init_upstream` 回调。

设置 `uscf->flags`

1. `NGX_HTTP_UPSTREAM_CREATE`: 创建标志，如果含有创建标志的话，nginx 会检查重复创建，以及必要参数是否填写；
2. `NGX_HTTP_UPSTREAM_MAX_FAILS`: 可以在 `server` 中使用 `max_fails` 属性；
3. `NGX_HTTP_UPSTREAM_FAIL_TIMEOUT`: 可以在 `server` 中使用 `fail_timeout` 属性；
4. `NGX_HTTP_UPSTREAM_DOWN`: 可以在 `server` 中使用 `down` 属性；

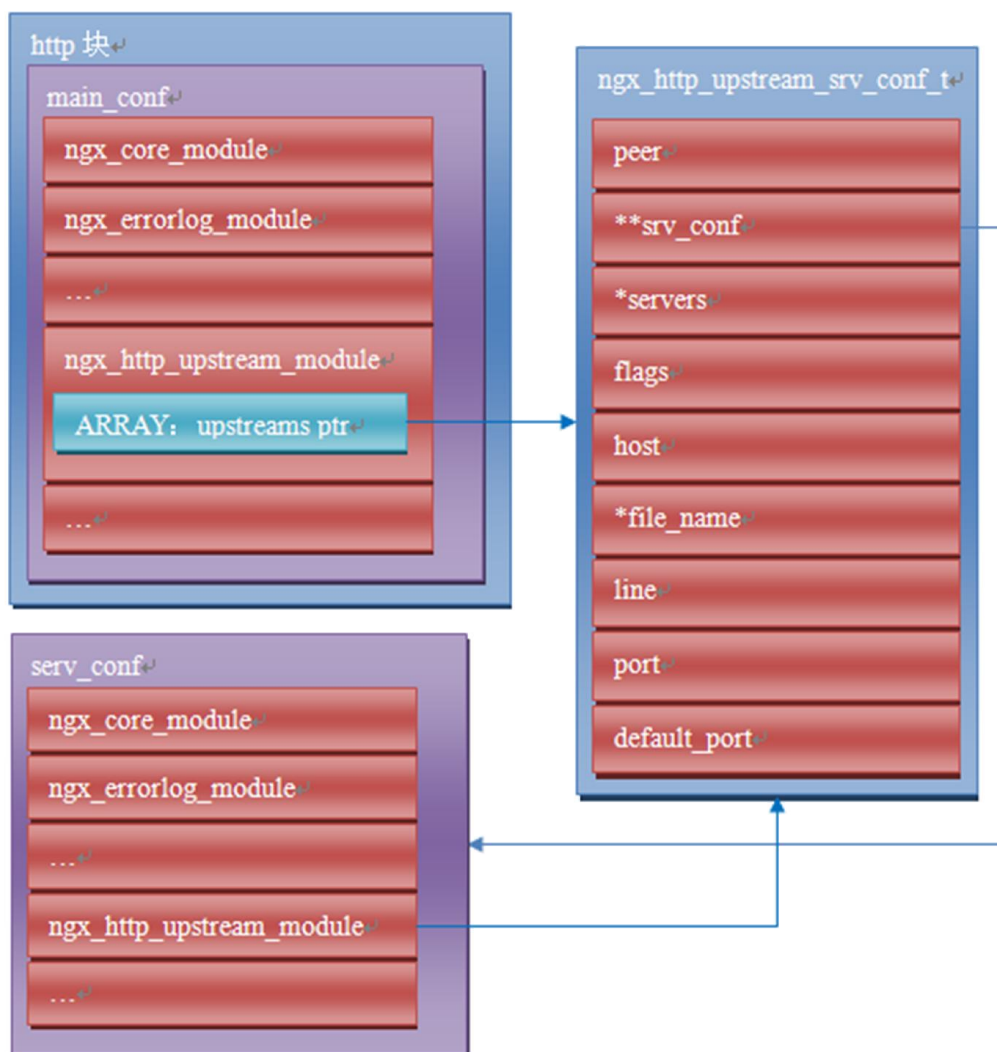
此外还有下面属性：

5. `NGX_HTTP_UPSTREAM_WEIGHT`: 可以在 `server` 中使用 `weight` 属性；
6. `NGX_HTTP_UPSTREAM_BACKUP`: 可以在 `server` 中使用 `backup` 属性。

聪明的读者如果联想到刚刚遇到的那个神奇的配置错误，可以得出一个结论：在负载均衡模块的指令处理函数中可以设置并修改 `upstream{}` 中“`server`”指令支持的属性。这是一个很重要的性质，因为不同的负载均衡模块对各种属性的支持情况都是不一样的，那么就需要在解析配置文件的时候检测出是否使用了不支持的负载均衡属性并给出错误提示，这对于提升系统维护性是很有意义的。但是，这种机制也存在缺陷，正如前面的例子所示，没有机制能够追加检查在更新支持属性之前已经配置了不支持属性的“`server`”指令。

设置 `init_upstream` 回调

nginx 初始化 `upstream` 时，会在 `ngx_http_upstream_init_main_conf` 函数中调用设置的回调函数初始化负载均衡模块。这里不太好理解的是 `uscf` 的具体位置。通过下面的示意图，说明 `upstream` 负载均衡模块的配置的内存布局。



从图上可以看出，MAIN_CONF 中 ngx_upstream_module 模块的配置项中有一个指针数组 upstreams，数组中的每个元素对应就是配置文件中每一个 upstream{} 的信息。更具体的将会在后面的原理篇讨论。

初始化配置

init_upstream 回调函数执行时需要初始化负载均衡模块的配置，还要设置一个新钩子，这个钩子函数会在 nginx 处理每个请求时作为初始化函数调用，关于这个新钩子函数的功能，后面会有详细的描述。这里，我们先分析 IP hash 模块初始化配置的代码：

```
ngx_http_upstream_init_round_robin(cf, us);  
us->peer.init = ngx_http_upstream_init_ip_hash_peer;
```

这段代码非常简单：IP hash 模块首先调用另一个负载均衡模块 Round Robin 的初始化函数，然后再设置自己的处理请求阶段初始化钩子。实际上几个负载均衡模块可以组成一条链表，每次都是从链首的模块开始进行处理。如果模块决定不处理，可以将处理权交给链表中的下一个模块。这里，IP hash 模块指定 Round Robin 模块作为自己的后继负载均衡模块，所以在自己的初始化配置函数中也对 Round Robin 模块进行初始化。

初始化请求

nginx 收到一个请求以后，如果发现需要访问 upstream，就会执行对应的 peer.init 函数。这是在初始化配置时设置的回调函数。这个函数最重要的作用是构造一张表，当前请求可以使用的 upstream 服务器被依次添加到这张表中。之所以需要这张表，最重要的原因是如果 upstream 服务器出现异常，不能提供服务时，可以从这张表中取得其他服务器进行重试操作。此外，这张表也可以用于负载均衡的计算。之所以构造这张表的行为放在这里而不是在前面初始化配置的阶段，是因为 upstream 需要为每一个请求提供独立隔离的环境。

为了讨论 peer.init 的核心，我们还是看 IP hash 模块的实现：

```
r->upstream->peer.data = &iphash->rrp;  
  
ngx_http_upstream_init_round_robin_peer(r, us);
```

```
r->upstream->peer.get = ngx_http_upstream_get_ip_hash_peer;
```

第一行是设置数据指针，这个指针就是指向前面提到的那张表；

第二行是调用 Round Robin 模块的回调函数对该模块进行请求初始化。面前已经提到，一个负载均衡模块可以调用其他负载均衡模块以提供功能的补充。

第三行是设置一个新的回调函数 `get`。该函数负责从表中取出某个服务器。除了 `get` 回调函数，还有另一个 `r->upstream->peer.free` 的回调函数。该函数在 `upstream` 请求完成后调用，负责做一些善后工作。比如我们需要维护一个 `upstream` 服务器访问计数器，那么可以在 `get` 函数中对其加 1，在 `free` 中对其减 1。如果是 SSL 的话，nginx 还提供两个回调函数 `peer.set_session` 和 `peer.save_session`。一般来说，有两个切入点实现负载均衡算法，其一是在这里，其二是在 `get` 回调函数中。

peer.get 和 peer.free 回调函数

这两个函数是负载均衡模块最底层的函数，负责实际获取一个连接和回收一个连接的预备操作。之所以说是预备操作，是因为在这两个函数中，并不实际进行建立连接或者释放连接的动作，而只是执行获取连接的地址或维护连接状态的操作。需要理解的清楚一点，在 `peer.get` 函数中获取连接的地址信息，并不代表这时连接一定没有被建立，相反的，通过 `get` 函数的返回值，nginx 可以了解是否存在可用连接，连接是否已经建立。这些返回值总结如下：

返回值	说明	nginx 后续动作
NGX_DONE	得到了连接地址信息，并且连接已经建立。	直接使用连接，发送数据。
NGX_OK	得到了连接地址信息，但连接并未建立。	建立连接，如连接不能立即建立，设置事件， 暂停执行本请求，执行别的请求。
NGX_BUSY	所有连接均不可用。	返回 502 错误至客户端。

各位读者看到上面这张表，可能会有几个问题浮现出来：

- Q:

什么时候连接是已经建立的？
- A:

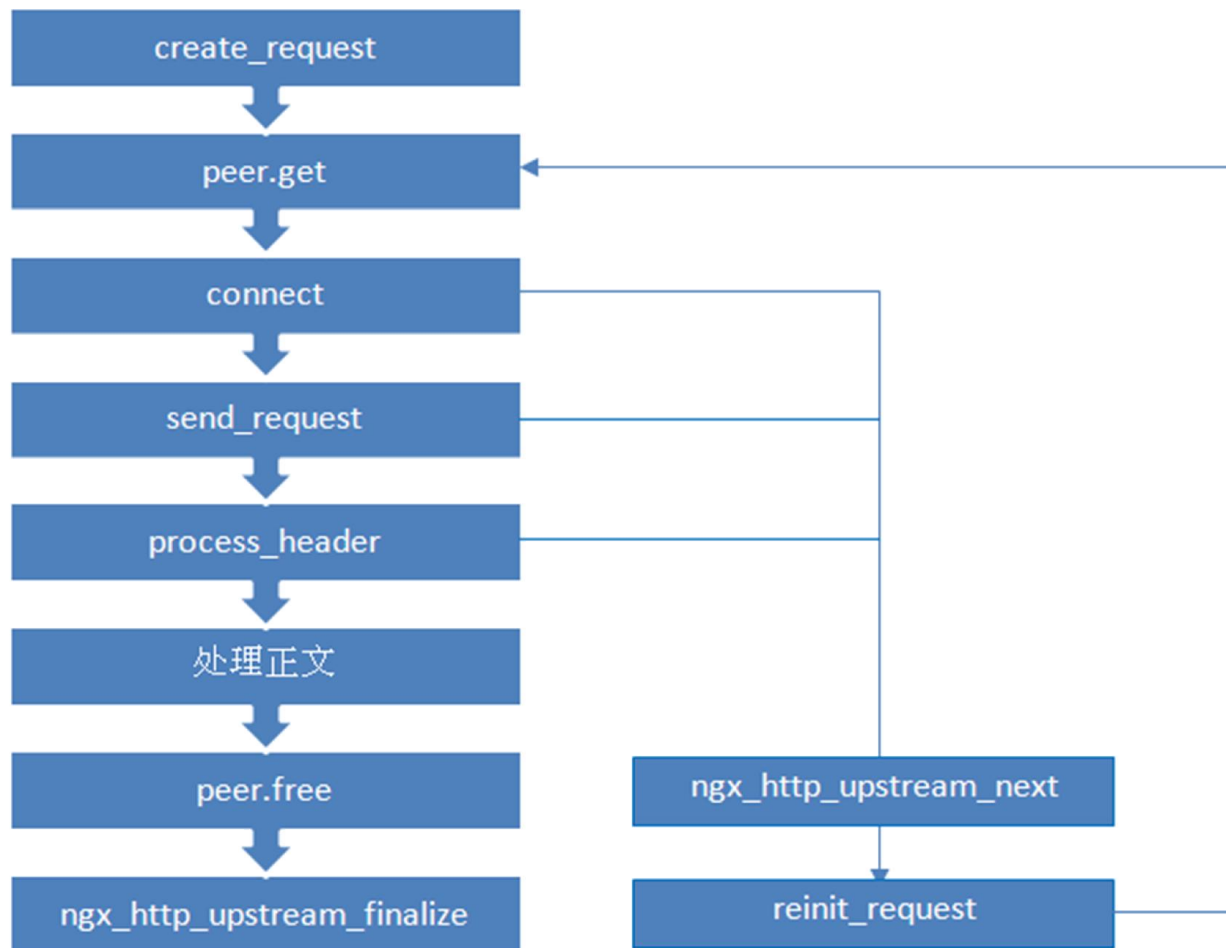
使用后端 `keepalive` 连接的时候，连接在使用完以后并不关闭，而是存放在一个队列中，新的请求只需要从队列中取出连接，这些连接都是已经准备好的。
- Q:

什么叫所有连接均不可用？
- A:

初始化请求的过程中，建立了一张表，`get` 函数负责每次从这张表中不重复的取出一个连接，当无法从表中取得一个新的连接时，即所有连接均不可用。
- Q:

对于一个请求，`peer.get` 函数可能被调用多次么？
- A:

正式如此。当某次 `peer.get` 函数得到的连接地址连接不上，或者请求对应的服务器得到异常响应，nginx 会执行 `ngx_http_upstream_next`，然后可能再次调用 `peer.get` 函数尝试别的连接。`upstream` 整体流程如下：



本节回顾

这一节介绍了负载均衡模块的基本组成。负载均衡模块的配置区集中在 `upstream{}` 块中。负载均衡模块的回调函数体系是以 `init_upstream` 为起点，经历 `init_peer`，最终到达 `peer.get` 和 `peer.free`。其中 `init_peer` 负责建立每个请求使用的 `server` 列表，`peer.get` 负责从 `server` 列表中选择某个 `server`（一般是不重复选择），而 `peer.free` 负责 `server` 释放前的资源释放工作。最后，这一节通过一张图将 `upstream` 模块和负载均衡模块在请求处理过程中的相互关系展现出来。

其他模块 (40%)

Nginx 的模块种类挺多的，除了 HTTP 模块，还有一些核心模块和 `mail` 系列模块。核心模块主要是做一些基础功能，比如 Nginx 的启动初始化，`event` 处理机制，错误日志的初始化，`ssl` 的初始化，正则处理初始化。

`mail` 模块可以对 `imap`，`pop3`，`smtp` 等协议进行反向代理，这些模块本身不对邮件内容进行处理。

core 模块 (40%)

Nginx 的启动模块 (40%)

启动模块从启动 Nginx 进程开始，做了一系列的初始化工作，源代码位于 `src/core/nginx.c`，从 `main` 函数开始：

1. 时间、正则、错误日志、ssl 等初始化
2. 读入命令行参数
3. OS 相关初始化
4. 读入并解析配置
5. 核心模块初始化
6. 创建各种暂时文件和目录
7. 创建共享内存
8. 打开 `listen` 的端口
9. 所有模块初始化
10. 启动 `worker` 进程

event 模块 (40%)

event 的类型和功能 (40%)

Nginx 是以 event（事件）处理模型为基础的模块。它为了支持跨平台，抽象出了 event 模块。它支持的 event 处理类型有：AIO（异步 IO），`/dev/poll`（Solaris 和 Unix 特有），`epoll`（Linux 特有），`eventport`（Solaris 10 特有），`kqueue`（BSD 特有），`poll`，`rtsig`（实时信号），`select` 等。

event 模块的主要功能就是，监听 `accept` 后建立的连接，对读写事件进行添加删除。事件处理模型和 Nginx 的非阻塞 IO 模型结合在一起使用。当 IO 可读可写的时候，相应的读写事件就会被唤醒，此时就会去处理事件的回调函数。

特别对于 Linux，Nginx 大部分 event 采用 `epoll EPOLLET`（边沿触发）的方法来触发事件，只有 `listen` 端口的读事件是 `EPOLL`（水平触发）。对于边沿触发，如果出现了可读事件，必须及时处理，否则可能会出现读事件不再触发，连接饿死的情况。

```
typedef struct {  
    /* 添加删除事件 */  
    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);

    /* 添加删除连接，会同时监听读写事件 */
    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);
    /* 处理事件的函数 */
    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;

```

上述是 event 处理抽象出来的关键结构体，可以看到，每个 event 处理模型，都需要实现部分功能。

最关键的是 add 和 del 功能，就是最基本的添加和删除事件的函数。

accept 锁 (40%)

Nginx 是多进程程序，80 端口是各进程所共享的，多进程同时 listen 80 端口，势必会产生竞争，也产生了所谓的“惊群”效应。当内核 accept 一个连接时，会唤醒所有等待中的进程，但实际上只有一个进程能获取连接，其他的进程都是被无效唤醒的。所以 Nginx 采用了自有的一套 accept 加锁机制，避免多个进程同时调用 accept。Nginx 多进程的锁在底层默认是通过 CPU 自旋锁来实现。如果操作系统不支持自旋锁，就采用文件锁。

Nginx 事件处理的入口函数是 ngx_process_events_and_timers()，下面是部分代码，可以看到其加锁的过程：

```

if (ngx_use_accept_mutex) {
    if (ngx_accept_disabled > 0) {
        ngx_accept_disabled--;
    } else {
        if (ngx_trylock_accept_mutex(cycle) == NGX_ERROR) {
            return;
        }
    }
}

```

```

        if (ngx_accept_mutex_held) {
            flags |= NGX_POST_EVENTS;

        } else {
            if (timer == NGX_TIMER_INFINITE
                || timer > ngx_accept_mutex_delay)
            {
                timer = ngx_accept_mutex_delay;
            }
        }
    }
}

```

在 `ngx_trylock_accept_mutex()` 函数里面，如果拿到了锁，Nginx 会把 listen 的端口读事件加入 event 处理，该进程在有新连接进来时就可以进行 accept 了。注意 accept 操作是一个普通的读事件。下面的代码说明了这点：

```

(void) ngx_process_events(cycle, timer, flags);

if (ngx_posted_accept_events) {
    ngx_event_process_posted(cycle, &ngx_posted_accept_events);
}

if (ngx_accept_mutex_held) {
    ngx_shmtx_unlock(&ngx_accept_mutex);
}

```

`ngx_process_events()` 函数是所有事件处理的入口，它会遍历所有的事件。抢到了 accept 锁的进程跟一般进程稍微不同的是，它被加上了 `NGX_POST_EVENTS` 标志，也就是说在 `ngx_process_events()` 函数里面只接受而不处理事件，并加入 `post_events` 的队列里面。直到 `ngx_accept_mutex` 锁去掉以后才去处理具体的事件。为什么这样？因为 `ngx_accept_mutex` 是全局锁，这样做可以尽量减少该进程抢到锁以后，从 accept 开始到结束的时间，以便其他进程继续接收新的连接，提高吞吐量。

`ngx_posted_accept_events` 和 `ngx_posted_events` 就分别是 accept 延迟事件队列和普通延迟事件队列。可以看到 `ngx_posted_accept_events` 还是放到 `ngx_accept_mutex` 锁里面处理的。该队列里面处理的都是 accept 事件，它会一口气把内核 backlog 里等待的连接都 accept 进来，注册到读写事件里。

而 `ngx_posted_events` 是普通的延迟事件队列。一般情况下，什么样的事件会放到这个普通延迟队列里面呢？我的理解是，那些 CPU 耗时比较多的都可以放进去。因为 Nginx 事件处理都是根据触发顺序在一个大循环里依次处理的，因为 Nginx 一个进程同时只能处理一个事件，所以有些耗时多的事件会把后面所有事件的处理都耽搁了。

除了加锁，Nginx 也对各进程的请求处理的均衡性作了优化，也就是说，如果在负载高的时候，进程抢到的锁过多，会导致这个进程被禁止接受请求一段时间。

比如，在 `ngx_event_accept` 函数中，有类似代码：

```
ngx_accept_disabled = ngx_cycle->connection_n / 8
                    - ngx_cycle->free_connection_n;
```

`ngx_cycle->connection_n` 是进程可以分配的连接总数，`ngx_cycle->free_connection_n` 是空闲的进程数。上述等式说明了，当前进程的空闲进程数小于 $1/8$ 的话，就会被禁止 `accept` 一段时间。

定时器 (40%)

Nginx 在需要用到超时的时候，都会用到定时器机制。比如，建立连接以后的那些读写超时。Nginx 使用红黑树来构造定期器，红黑树是一种有序的二叉平衡树，其查找插入和删除的复杂度都为 $O(\log n)$ ，所以是一种比较理想的二叉树。

定时器的机制就是，二叉树的值是其超时时间，每次查找二叉树的最小值，如果最小值已经过期，就删除该节点，然后继续查找，直到所有超时节点都被删除。

mail 模块

mail 模块的实现

mail 模块的功能

模块开发高级篇(30%)

变量(80%)

综述

在 Nginx 中同一个请求需要在模块之间数据的传递或者说在配置文件里面使用模块动态的数据一般来说都是使用变量, 比如在 HTTP 模块中导出了 `host/remote_addr` 等变量, 这样我们就可以在配置文件中以及其他的模块使用这个变量。在 Nginx 中, 有两种定义变量的方式, 一种是在配置文件中, 使用 `set` 指令, 一种就是上面我们提到的在模块中定义变量, 然后导出。

在 Nginx 中所有的变量都是与 HTTP 相关的(也就是说赋值都是在请求阶段), 并且基本上是同时保存在两个数据结构中, 一个就是 hash 表(可选), 另一个是数组。比如一些特殊的变量, 比如 `arg_xxx/cookie_xxx` 等, 这些变量的名字是不确定的(因此不能内置), 而且他们还是只读的(不能交由用户修改), 如果每个都要放到 hash 表中的话(不知道用户会取多少个), 会很占空间的, 因此这些变量就没有 hash, 只有索引。这里要注意, 用户不能定义这样的变量, 这样的变量只存在于 Nginx 内部。

对应的变量结构体是这样子(每一个变量都是一个 `ngx_http_variable_s` 结构体)的:

```
struct ngx_http_variable_s {
    ngx_str_t          name;    /* must be first to build the hash */
    ngx_http_set_variable_pt set_handler;
    ngx_http_get_variable_pt get_handler;
    uintptr_t          data;
    ngx_uint_t         flags;
    ngx_uint_t         index;
};
```

其中 `name` 表示对应的变量名字, `set/get_handler` 表示对应的设置以及读取回调, 而 `data` 是传递给回调的参数, `flags` 表示变量的属性, `index` 提供了一个索引(数组的脚标), 从而可以迅速定位到对应的变量。`set/get_handler` 只有在真正读取设置变量的时候才会被调用。

这里要注意 `flag` 属性, `flag` 属性就是由下面的几个属性组合而成:

```
#define NGX_HTTP_VAR_CHANGEABLE 1
#define NGX_HTTP_VAR_NOCACHEABLE 2
#define NGX_HTTP_VAR_INDEXED 4
#define NGX_HTTP_VAR_NOHASH 8
```


1. `NGX_HTTP_VAR_CHANGEABLE` 表示这个变量是可变的.Nginx 有很多内置变量是不可变的, 比如 `arg_xxx` 这类变量, 如果你使用 `set` 指令来修改, 那么 Nginx 就会报错.
2. `NGX_HTTP_VAR_NOCACHEABLE` 表示这个变量每次都要去取值, 而不是直接返回上次 `cache` 的值(配合对应的接口).
3. `NGX_HTTP_VAR_INDEXED` 表示这个变量是用索引读取的.
4. `NGX_HTTP_VAR_NOHASH` 表示这个变量不需要被 `hash`.

而变量在 Nginx 中的初始化流程是这样的:

1. 首先当解析 HTTP 之前会调用 `ngx_http_variables_add_core_vars(pre_config)`来将 HTTP core 模块导出的变量(`http_host/remote_addr...`)添加进全局的 `hash key` 链中.
2. 解析完 HTTP 模块之后, 会调用 `ngx_http_variables_init_vars` 来初始化所有的变量(不仅包括 HTTP core 模块的变量, 也包括其他的 HTTP 模块导出的变量,以及配置文件中使用 `set` 命令设置的变量),这里的初始化包括初始化 `hash` 表,以及初始化数组索引.
3. 当每次请求到来时会给每个请求创建一个变量数组(数组的个数就是上面第二步所保存的变量个数).然后只有取变量值的时候, 才会将变量保存在对应的变量数组位置.

创建变量

在 Nginx 中, 创建变量有两种方式, 分别是在配置文件中使用 `set` 指令, 和在模块中调用对应的接口, 在配置文件中创建变量比较简单, 因此我们主要来看如何在模块中创建自己的变量。

在 Nginx 中提供了下面的接口, 可以供模块调用来创建变量。

```
ngx_http_variable_t *ngx_http_add_variable(ngx_conf_t *cf, ngx_str_t *name,
ngx_uint_t flags);
```

这个函数所做的工作就是将变量 “name”添加进全局的 `hash key` 表中,然后初始化一些域, 不过这里要注意, 对应的变量的 `get/set` 回调, 需要当这个函数返回之后, 显式的设置,比如在 `split_clients` 模块中的例子:

```
var = ngx_http_add_variable(cf, &name, NGX_HTTP_VAR_CHANGEABLE);
if (var == NULL) {
    return NGX_CONF_ERROR;
}
//设置回调
var->get_handler = ngx_http_split_clients_variable;
var->data = (uintptr_t) ctx;
```

而对应的回调函数原型是这样的:

```
typedef void (*ngx_http_set_variable_pt) (ngx_http_request_t *r,
```

```
ngx_http_variable_value_t *v, uintptr_t data);
typedef ngx_int_t (*ngx_http_get_variable_pt) (ngx_http_request_t *r,
ngx_http_variable_value_t *v, uintptr_t data);
```

回调函数比较简单，第一个参数是当前请求，第二个是需要设置或者获取的变量值，第三个是初始化时的回调指针，这里我们着重来看一下 ngx_http_variable_value_t,下面就是这个结构体的原型:

```
typedef struct {
    unsigned    len:28;

    unsigned    valid:1;
    unsigned    no_cacheable:1;
    unsigned    not_found:1;
    unsigned    escape:1;
    u_char      *data;
} ngx_variable_value_t;
```

这里主要是 data 域，当我们在 get_handle 中设置变量值的时候，只需要将对应的值放入到 data 中就可以了，这里 data 需要在 get_handle 中分配内存,比如下面的例子

(ngx_http_fastcgi_script_name_variable),就是 fastcgi_script_name 变量的 get_handler 代码片段:

```
v->len = f->script_name.len + flcf->index.len;

v->data = ngx_pnalloc(r->pool, v->len);
if (v->data == NULL) {
    return NGX_ERROR;
}

p = ngx_copy(v->data, f->script_name.data, f->script_name.len);
ngx_memcpy(p, flcf->index.data, flcf->index.len);
```

使用变量

Nginx 的内部变量指的就是 Nginx 的官方模块中所导出的变量，在 Nginx 中，大部分常用的变量都是 CORE HTTP 模块导出的。而在 Nginx 中，不仅可以在模块代码中使用变量，而且还可以在配置文件中使用。

假设我们需要在配置文件中使用时 http 模块的 host 变量，那么只需要这样在变量名前加一个\$符号就可以了(\$host).而如果需要在模块中使用 host 变量，那么就比较麻烦，Nginx 提供了下面几个接口来取得变量:

```
ngx_http_variable_value_t *ngx_http_get_indexed_variable(ngx_http_request_t *r,
    ngx_uint_t index);
ngx_http_variable_value_t *ngx_http_get_flushed_variable(ngx_http_request_t *r,
    ngx_uint_t index);
ngx_http_variable_value_t *ngx_http_get_variable(ngx_http_request_t *r,
    ngx_str_t *name, ngx_uint_t key);
```

他们的区别是这样子的，`ngx_http_get_indexed_variable` 和 `ngx_http_get_flushed_variable` 都是用来取得有索引的变量，不过他们的区别是后一个会处理 `NGX_HTTP_VAR_NOCACHEABLE` 这个标记，也就是说如果你想要 cache 你的变量值，那么你的变量属性就不能设置 `NGX_HTTP_VAR_NOCACHEABLE`，并且通过 `ngx_http_get_flushed_variable` 来获取变量值。而 `ngx_http_get_variable` 和上面的区别就是它能够得到没有索引的变量值。

通过上面我们知道可以通过索引来得到变量值，可是这个索引该如何取得呢，Nginx 也提供了对应的接口：

```
ngx_int_t ngx_http_get_variable_index(ngx_conf_t *cf, ngx_str_t *name);
```

通过这个接口，就可以取得对应变量的索引值。

接下来来看对应的例子，比如在 `http_log` 模块中，如果在 `log_format` 中配置了对应的变量，那么它会调用 `ngx_http_get_variable_index` 来保存索引：

```
static ngx_int_t
ngx_http_log_variable_compile(ngx_conf_t *cf, ngx_http_log_op_t *op,
    ngx_str_t *value)
{
    ngx_int_t index;
    //得到变量的索引
    index = ngx_http_get_variable_index(cf, value);
    if (index == NGX_ERROR) {
        return NGX_ERROR;
    }

    op->len = 0;
    op->getlen = ngx_http_log_variable_getlen;
    op->run = ngx_http_log_variable;
    //保存索引值
    op->data = index;

    return NGX_OK;
}
```

```
}
```

然后 `http_log` 模块会使用 `ngx_http_get_indexed_variable` 来得到对应的变量值,这里要注意,就是使用这个接口的时候,判断返回值,不仅要判断是否为空,也需要判断 `value->not_found`,这是因为只有第一次调用才会返回空,后续返回就不是空,因此需要判断 `value->not_found`:

```
static u_char *
ngx_http_log_variable(ngx_http_request_t *r, u_char *buf, ngx_http_log_op_t *op)
{
    ngx_http_variable_value_t *value;
    //获取变量值
    value = ngx_http_get_indexed_variable(r, op->data);

    if (value == NULL || value->not_found) {
        *buf = '-';
        return buf + 1;
    }

    if (value->escape == 0) {
        return ngx_cpymem(buf, value->data, value->len);
    } else {
        return (u_char *) ngx_http_log_escape(buf, value->data, value->len);
    }
}
```

upstream

使用 **subrequest** 访问 **upstream**

超越 **upstream**

event 机制

例讲（主动健康检查模块）

使用 **lua** 模块

下篇：nginx 原理解析篇

高性能服务器设计

c10k 问题

高性能服务器编写的关键原则

事件驱动的核心引擎

定时器管理

nginx 架构详解(50%)

nginx 的下篇将会更加深入的介绍 nginx 的实现原理。上一章，我们了解到了如何设计一个高性能服务器，那这一章将会开始讲解，nginx 是如何一步一步实现高性能服务器的。

nginx 的源码目录结构(100%)

nginx 的优秀除了体现在程序结构以及代码风格上，nginx 的源码组织也同样简洁明了，目录结构层次结构清晰，值得我们去学习。nginx 的源码目录与 nginx 的模块化以及功能的划分是紧密结合，这也使得我们可以很方便地找到相关功能的代码。这节先介绍 nginx 源码的目录结构，先对 nginx 的源码有一个大致的认识，下节会讲解 nginx 如何编译。

下面是 nginx 源码的目录结构：

.	
— auto	自动检测系统环境以及编译相关的脚本
— cc	关于编译器相关的编译选项的检测脚本
— lib	nginx 编译所需要的一些库的检测脚本
— os	与平台相关的一些系统参数与系统调用相关的检测
— types	与数据类型相关的一些辅助脚本
— conf	存放默认配置文件，在 <code>make install</code> 后，会拷贝到安装目录中去
— contrib	存放一些实用工具，如 <code>geo</code> 配置生成工具（ <code>geo2nginx.pl</code> ）
— html	存放默认的网页文件，在 <code>make install</code> 后，会拷贝到安装目录中去

└─ man	nginx 的 man 手册
└─ src	存放 nginx 的源代码
└─ core	nginx 的核心源代码，包括常用数据结构的定义，以及 nginx 初始化运行的核心代码如 main 函数
└─ event	对系统事件处理机制的封装，以及定时器的实现相关代码
└─ modules	不同事件处理方式的模块化，如 select、poll、epoll、kqueue 等
└─ http	nginx 作为 http 服务器相关的代码
└─ modules	包含 http 的各种功能模块
└─ mail	nginx 作为邮件代理服务器相关的代码
└─ misc	一些辅助代码，测试 c++ 头的兼容性，以及对 google_perftools 的支持
└─ os	主要是对各种不同体系系统结构所提供的系统函数的封装，对外提供统一的系统调用接口

nginx 的 configure 原理(100%)

nginx 的编译旅程将从 configure 开始，configure 脚本将根据我们输入的选项、系统环境参与来生成所需的文件（包含源文件与 Makefile 文件）。configure 会调用一系列 auto 脚本来实现编译环境的初始化。

auto 脚本

auto 脚本由一系列脚本组成，他们有一些是实现一些通用功能由其它脚本来调用（如 have），有一些则是完成一些特定的功能（如 option）。脚本之间的主要执行顺序及调用关系如下图所示（由上到下，表示主流程的执行）：

接下来，我们结合代码来分析下 configure 的原理：

1. 初始化

```
. auto/options
. auto/init
. auto/sources
```

这是 configure 源码开始执行的前三行，依次交由 auto 目录下面的 option、init、sources 来处理。

2. auto/options 主是处理用户输入的 configure 选项，以及输出帮助信息等。读者可以结合 nginx 的源码来阅读本章内容。由于篇幅关系，这里大致列出此文件的结构：

##1. 设置选项对应的 shell 变量以及他们的初始值

```
help=no
NGX_PREFIX=
NGX_SBIN_PATH=
NGX_CONF_PREFIX=
NGX_CONF_PATH=
NGX_ERROR_LOG_PATH=
NGX_PID_PATH=
NGX_LOCK_PATH=
NGX_USER=
NGX_GROUP=
```

...

2, 处理每一个选项值, 并设置到对应的全局变量中

for option

do

```
opt="$opt `echo $option | sed -e \"s/\(--[^\]=\)\"(.\" .*)\"/\1'\2'/'\"`"
```

得到此选项的 value 部分

```
case "$option" in
```

```
  *=*) value=`echo "$option" | sed -e 's/[_a-zA-Z0-9]*=/'` ;;
  *) value="" ;;
```

```
esac
```

根据 option 内容进行匹配, 并设置相应的选项

```
case "$option" in
```

```
  --help)                help=yes                ;;
  --prefix=)             NGX_PREFIX="!"           ;;
  --prefix=*)            NGX_PREFIX="$value"       ;;
  --sbin-path=*)         NGX_SBIN_PATH="$value"    ;;
  --conf-path=*)         NGX_CONF_PATH="$value"    ;;
  --error-log-path=*)    NGX_ERROR_LOG_PATH="$value";;
  --pid-path=*)          NGX_PID_PATH="$value"     ;;
  --lock-path=*)         NGX_LOCK_PATH="$value"    ;;
  --user=*)              NGX_USER="$value"        ;;
  --group=*)             NGX_GROUP="$value"       ;;
```

...

```
*)
```

没有找到的对应选项

```
echo "$0: error: invalid option \"$option\""
```

```
exit 1
```

```
;;
```

```
esac
```

done

3. 对选项进行处理

```
# 如果有--help, 则输出帮助信息
if [ $help = yes ]; then
```

```
    cat << END
```

--help	print this message
--prefix=PATH	set installation prefix
--sbin-path=PATH	set nginx binary pathname
--conf-path=PATH	set nginx.conf pathname
--error-log-path=PATH	set error log pathname
--pid-path=PATH	set nginx.pid pathname
--lock-path=PATH	set nginx.lock pathname
--user=USER	set non-privileged user for
worker processes	
--group=GROUP	set non-privileged group for
	worker processes

```
END
```

```
    exit 1
```

```
fi
```

```
# 默认文件路径
```

```
NGX_CONF_PATH=${NGX_CONF_PATH:-conf/nginx.conf}
```

```
NGX_CONF_PREFIX=`dirname $NGX_CONF_PATH`
```

```
NGX_PID_PATH=${NGX_PID_PATH:-logs/nginx.pid}
```

```
NGX_LOCK_PATH=${NGX_LOCK_PATH:-logs/nginx.lock}
```

```
...
```

上面的代码中, 我们选用了文件中的部分代码进行了说明。大家可结合源码再进行分析。`auto/options` 的目的主要是处理用户选项, 并由选项生成一些全局变量的值, 这些值在其它文件中会用到。该文件也会输出 `configure` 的帮助信息。

3. auto/init

该文件的目录在于初始化一些临时文件的路径, 检查 `echo` 的兼容性, 并创建 `Makefile`。

```
# 生成最终执行编译的 makefile 文件路径
```

```
NGX_MAKEFILE=$NGX_OBJS/Makefile
```

```
# 动态生成 nginx 模块列表的路径, 由于 nginx 的的一些模块是可以选择编译的, 而且可以添加自己的模块, 所以模块列表是动态生成的
```



```

NGX_MODULES_C=$NGX_OBJS/nginx_modules.c

NGX_AUTO_HEADERS_H=$NGX_OBJS/nginx_auto_headers.h
NGX_AUTO_CONFIG_H=$NGX_OBJS/nginx_auto_config.h

# 自动测试目录与日志输出文件
NGX_AUTOTEST=$NGX_OBJS/autotest
# 如果 configure 出错，可用来查找出错的原因
NGX_AUTOCONF_ERR=$NGX_OBJS/autoconf.err

NGX_ERR=$NGX_OBJS/autoconf.err
MAKEFILE=$NGX_OBJS/Makefile


NGX_PCH=
NGX_USE_PCH=


# 检查 echo 是否支持 -n 或 \c

# check the echo's "-n" option and "\c" capability
if echo "test\c" | grep c >/dev/null; then

    # 不支持 -c 的方式，检查是否支持 -n 的方式

    if echo -n test | grep n >/dev/null; then
        ngx_n=
        ngx_c=

    else
        ngx_n=-n
        ngx_c=
    fi

else
    ngx_n=
    ngx_c='\c'
fi


# 创建最初始的 makefile 文件
# default 表示目前编译对象
# clean 表示执行 clean 工作时，需要删除 makefile 文件以及 objs 目录
# 整个过程中只会生成 makefile 文件以及 objs 目录，其它所有临时文件都在 objs 目录之下，
所以执行 clean 后，整个目录还原到初始状态
# 要再次执行编译，需要重新执行 configure 命令

# create Makefile

```

```
cat << END > Makefile

default:    build

clean:
    rm -rf Makefile $NGX_OBJS
END
```

4. auto/sources

该文件从文件名中就可以看出，它的主要功能是跟源文件相关的。它的主要作用是定义不同功能或系统所需要的文件的变量。根据功能，分为 CORE/REGEX/EVENT/UNIX/FREEBSD/HTTP 等。每一个功能将会由四个变量组成，“_MODULES”表示此功能相关的模块，最终会输出到 ngx_modules.c 文件中，即动态生成需要编译到 nginx 中的模块；“INCS”表示此功能依赖的源码目录，查找头文件的时候会用到，在编译选项中，会出现在“-I”中；“DEPS”显示指明在 Makefile 中需要依赖的文件名，即编译时，需要检查这些文件的更新时间；“SRCS”表示需要此功能编译需要的源文件。

拿 core 来说：

```
CORE_MODULES="ngx_core_module ngx_errlog_module ngx_conf_module
ngx_emp_server_module ngx_emp_server_core_module"
```

```
CORE_INCS="src/core"
```

```
CORE_DEPS="src/core/nginx.h \
    src/core/nginx_config.h \
    src/core/nginx_core.h \
    src/core/nginx_log.h \
    src/core/nginx_palloc.h \
    src/core/nginx_array.h \
    src/core/nginx_list.h \
    src/core/nginx_hash.h \
    src/core/nginx_buf.h \
    src/core/nginx_queue.h \
    src/core/nginx_string.h \
    src/core/nginx_parse.h \
    src/core/nginx_inet.h \
    src/core/nginx_file.h \
    src/core/nginx_crc.h \
    src/core/nginx_crc32.h \
    src/core/nginx_murmurhash.h \
    src/core/nginx_md5.h \
```

```
src/core/nginx_sha1.h \  
src/core/nginx_rbtrees.h \  
src/core/nginx_radix_tree.h \  
src/core/nginx_slab.h \  
src/core/nginx_times.h \  
src/core/nginx_shmtx.h \  
src/core/nginx_connection.h \  
src/core/nginx_cycle.h \  
src/core/nginx_conf_file.h \  
src/core/nginx_resolver.h \  
src/core/nginx_open_file_cache.h \  
src/core/nginx_emp_server.h \  
src/core/emp_server.h \  
src/core/task_thread.h \  
src/core/standard.h \  
src/core/dprint.h \  
src/core/nginx_crypt.h"
```

```
CORE_SRCS="src/core/nginx.c \  
src/core/nginx_log.c \  
src/core/nginx_palloc.c \  
src/core/nginx_array.c \  
src/core/nginx_list.c \  
src/core/nginx_hash.c \  
src/core/nginx_buf.c \  
src/core/nginx_queue.c \  
src/core/nginx_output_chain.c \  
src/core/nginx_string.c \  
src/core/nginx_parse.c \  
src/core/nginx_inet.c \  
src/core/nginx_file.c \  
src/core/nginx_crc32.c \  
src/core/nginx_murmurhash.c \  
src/core/nginx_md5.c \  
src/core/nginx_rbtrees.c \  
src/core/nginx_radix_tree.c \  
src/core/nginx_slab.c \  
src/core/nginx_times.c \  
src/core/nginx_shmtx.c \  
src/core/nginx_connection.c \  
src/core/nginx_cycle.c \  
src/core/nginx_spinlock.c \  
src/core/nginx_cpuinfo.c \  
src/core/nginx_conf_file.c \  
src/core/nginx_resolver.c \  
src/core/nginx_open_file_cache.c \  
src/core/nginx_emp_server.c \  
src/core/emp_server.c \  
"
```

```
src/core/standard.c \  
src/core/task_thread.c \  
src/core/dprint.c \  
src/core/nginx_crypt.c"
```

如果我们自己写一个第三方模块，我们可能会引用到这些变量的值，或对这些变量进行修改，比如添加我们自己的模块，或添加自己的一个头文件查找目录(在第三方模块的 `config` 中)，在后面，我们会看到它是如何加框第三方模块的。在继续分析执行流程之前，我们先介绍一些工具脚本。

5. auto/have

```
cat << END >> $NGX_AUTO_CONFIG_H  
  
#ifndef $have  
#define $have 1  
#endif  
  
END
```

从代码中，我们可以看到，这个工具的作用是，将 `$have` 变量的值，宏定义为 `1`，并输出到 `auto_config` 文件中。通常我们通过这个工具来控制是否打开某个特性。这个工具在使用前，需要先定义宏的名称，即 `$have` 变量。

6. 再回到 `configure` 文件中来:

```
# NGX_DEBUG 是在 auto/options 文件中处理的，如果有 --with-debug 选项，则其值是 YES  
if [ $NGX_DEBUG = YES ]; then  
    # 当有 debug 选项时，会定义 NGX_DEBUG 宏  
    have=NGX_DEBUG . auto/have  
fi
```

这段代码中，可以看出，`configure` 是如何定义一个特性的：通过宏定义，输出到 `config` 头文件中，然后在程序中可以判断这个宏是否有定义，来实现不同的特性。

`configure` 文件中继续向下:

```
# 编译器选项  
. auto/cc/conf  
  
# 头文件支持宏定义  
if [ "$NGX_PLATFORM" != win32 ]; then  
    . auto/headers  
fi
```

```
# 操作系统相关的配置的检测
. auto/os/conf

# unix 体系下的通用配置检测
if [ "$NGX_PLATFORM" != win32 ]; then
    . auto/unix
fi
```

configure 会依次调用其它几个文件，来进行环境的检测，包括编译器、操作系统相关。

7. auto/feature

nginx 的 configure 会自动检测不同平台的特性，神奇之处就是 auto/feature 的实现，在继续向下分析之前，我们先来看看这个工具的实现原理。此工具的核心思想是，输出一小段代表性 c 程序，然后设置好编译选项，再进行编译连接运行，再对结果进行分析。例如，如果想检测某个库是否存在，就在小段 c 程序里面调用库里面的某个函数，再进行编译链接，如果出错，则表示库的环境不正常，如果编译成功，且运行正常，则库的环境检测正常。我们在写 nginx 第三方模块时，也常使用此工具来进行环境的检测，所以，此工具的作用贯穿整个 configure 过程。

先看一小段使用例子：

```
ngx_feature="poll()"
ngx_feature_name=
ngx_feature_run=no
ngx_feature_incs="#include <poll.h>"
ngx_feature_path=
ngx_feature_libs=
ngx_feature_test="int  n; struct pollfd  pl;
                  pl.fd = 0;
                  pl.events = 0;
                  pl.revents = 0;
                  n = poll(&pl, 1, 0);
                  if (n == -1) return 1"

. auto/feature

if [ $ngx_found = no ]; then
    # 如果没有找到 poll，就设置变量的值
    EVENT_POLL=NONE
fi
```

这段代码在 auto/unix 里面实现，用来检测当前操作系统是否支持 poll 函数调用。在调用 auto/feature 之前，需要先设置几个输入参数变量的值，然后结果会存在\$ngx_found 变量里面，并输出宏定义以表示支持此特性:

```
$ngx_feature      特性名称
$ngx_feature_name 特性的宏定义名称，如果特性测试成功，则会定义该宏定义
$ngx_feature_path 编译时要查找头文件目录
$ngx_feature_test 要执行的测试代码
$ngx_feature_incs 在代码中要 include 的头文件
$ngx_feature_libs 编译时需要 link 的库文件选项
$ngx_feature_run  编译成功后，对二进制文件需要做的动作，可以是 yes value bug 其它

#ngx_found 如果找到，并测试成功，其值为 yes，否则其值为 no
```

看看 ngx_feature 的关键代码:

```
# 初始化输出结果为 no
ngx_found=no

#将特性名称小写转换成大写
if test -n "$ngx_feature_name"; then
    # 小写转大写
    ngx_have_feature=`echo $ngx_feature_name \
                        | tr abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ`
fi

# 将所有 include 目录转换成编译选项
if test -n "$ngx_feature_path"; then
    for ngx_temp in $ngx_feature_path; do
        ngx_feature_inc_path="$ngx_feature_inc_path -I $ngx_temp"
    done
fi

# 生成临时的小段 c 程序代码。
# $ngx_feature_incs 变量是程序需要 include 的头文件
# $ngx_feature_test 是测试代码
cat << END > $NGX_AUTOTEST.c

#include <sys/types.h>
$NGX_INCLUDE_UNISTD_H
$ngx_feature_incs

int main() {
    $ngx_feature_test;
```

```

    return 0;
}

END

# 编译命令
# 编译之后的目标文件是 $NGX_AUTOTEST，后面会判断这个文件是否存在来判断是否编译成功
ngx_test="$CC $CC_TEST_FLAGS $CC_AUX_FLAGS $ngx_feature_inc_path \
    -o $NGX_AUTOTEST $NGX_AUTOTEST.c $NGX_TEST_LD_OPT $ngx_feature_libs"

# 执行编译过程
# 编译成功后，会生成$NGX_AUTOTEST 命名的文件
eval "/bin/sh -c \"$ngx_test\" >> $NGX_AUTOCONF_ERR 2>&1"

# 如果文件存在，则编译成功
if [ -x $NGX_AUTOTEST ]; then

    case "$ngx_feature_run" in

        # 需要运行来判断是否支持特性
        # 测试程序能否正常执行（即程序退出后的状态码是否是 0），如果正常退出，则特性
        # 测试成功，设置 ngx_found 为 yes，并添加名为 ngx_feature_name 的宏定义，宏的值为 1
        yes)
            # 如果程序正常退出，退出码为 0，则程序执行成功，我们可以在测试代码里面手
            # 动返回非 0 来表示程序出错
            # /bin/sh is used to intercept "Killed" or "Abort trap" messages
            if /bin/sh -c $NGX_AUTOTEST >> $NGX_AUTOCONF_ERR 2>&1; then
                echo " found"
                ngx_found=yes

                # 添加宏定义，宏的值为 1
                if test -n "$ngx_feature_name"; then
                    have=$ngx_have_feature . auto/have
                fi

            else
                echo " found but is not working"
            fi
        ;;

        # 需要运行程序来判断是否支持特性，如果支持，将程序标准输出的结果作为宏的值
        # value)
        # /bin/sh is used to intercept "Killed" or "Abort trap" messages
        if /bin/sh -c $NGX_AUTOTEST >> $NGX_AUTOCONF_ERR 2>&1; then
            echo " found"
            ngx_found=yes

```

```

        # 与 yes 不一样的是，value 会将程序从标准输出里面打印出来的值，设置为
ngx_feature_name 宏变量的值
        # 在此种情况下，程序需要设置 ngx_feature_name 变量名
        cat << END >> $NGX_AUTO_CONFIG_H

#ifdef $ngx_feature_name
#define $ngx_feature_name ` $NGX_AUTOTEST `
#endif

END

        else
            echo " found but is not working"
        fi
    ;;

    # 与 yes 正好相反
    bug)
        # /bin/sh is used to intercept "Killed" or "Abort trap" messages
        if /bin/sh -c $NGX_AUTOTEST >> $NGX_AUTOCONF_ERR 2>&1; then
            echo " not found"

        else
            echo " found"
            ngx_found=yes

            if test -n "$ngx_feature_name"; then
                have=$ngx_have_feature . auto/have
            fi
        fi
    ;;

    # 不需要运行程序，最后定义宏变量
    *)
        echo " found"
        ngx_found=yes

        if test -n "$ngx_feature_name"; then
            have=$ngx_have_feature . auto/have
        fi
    ;;

esac
else
    # 编译失败
    echo " not found"

    # 编译失败，会保存信息到日志文件中
    echo "-----" >> $NGX_AUTOCONF_ERR

```



```

# 保留编译文件的内容
cat $NGX_AUTOTEST.c >> $NGX_AUTOCONF_ERR
echo "-----" >> $NGX_AUTOCONF_ERR
# 保留编译文件的选项
echo $ngx_test >> $NGX_AUTOCONF_ERR
echo "-----" >> $NGX_AUTOCONF_ERR
fi

# 最后删除生成的临时文件
rm $NGX_AUTOTEST*

```

8. auto/cc/conf

在了解了工具 `auto/feature` 后，继续我们的主流程，`auto/cc/conf` 的代码就很好理解了，这一步主要是检测编译器，并设置编译器相关的选项。它先调用 `auto/cc/name` 来得到编译器的名称，然后根据编译器选择执行不同的编译器相关的文件如 `gcc` 执行 `auto/cc/gcc` 来设置编译器相关的一些选项。

9. auto/include

这个工具用来检测是头文件是否支持。需要检测的头文件放在 `$ngx_include` 里面，如果支持，则 `$ngx_found` 变量的值为 `yes`，并且会产生 `NGX_HAVE_{ngx_include}` 的宏定义。

10. auto/headers

生成头文件的宏定义。生成的定义放在 `objs/nginx_auto_headers.h` 里面：

```

#ifndef NGX_HAVE_UNISTD_H
#define NGX_HAVE_UNISTD_H 1
#endif

#ifndef NGX_HAVE_INTTYPES_H
#define NGX_HAVE_INTTYPES_H 1
#endif

#ifndef NGX_HAVE_LIMITS_H
#define NGX_HAVE_LIMITS_H 1
#endif

#ifndef NGX_HAVE_SYS_FILIO_H
#define NGX_HAVE_SYS_FILIO_H 1

```

```
#endif
```

```
#ifndef NGX_HAVE_SYS_PARAM_H  
#define NGX_HAVE_SYS_PARAM_H 1  
#endif
```

11. auto/os/conf

针对不同的操作系统平台特性的检测，并针对不同的操作系统，设置不同的 CORE_INCS、CORE_DEPS、CORE_SRCS 变量。nginx 跨平台的支持就是在这个地方体现出来的。

12. auto/unix

针对 unix 体系的通用配置或系统调用的检测，如 poll 等事件处理系统调用的检测等。

13. 回到 configure 里面

```
# 生成模块列表  
. auto/modules  
# 配置库的依赖  
. auto/lib/conf
```

14. auto/modules

该脚本根据不同的条件，输出不同的模块列表，最后输出的模块列表的文件在 objs/nginx_modules.c:

```
#include <ngx_config.h>  
#include <ngx_core.h>  
  
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_emp_server_module;  
  
...  
  
ngx_module_t *ngx_modules[] = {  
    &ngx_core_module,  
    &ngx_errlog_module,  
    &ngx_conf_module,  
    ...  
};
```

```
&ngx_emp_server_module,  
...  
NULL  
};
```

这个文件会决定所有模块的顺序，这会直接影响到最后的功能，下一小节我们将讨论模块间的顺序。这个文件会加载我们的第三方模块，这也是我们值得关注的地方：

```
if test -n "$NGX_ADDONS"; then  
  
    echo configuring additional modules  
  
    for ngx_addon_dir in $NGX_ADDONS  
    do  
        echo "adding module in $ngx_addon_dir"  
  
        if test -f $ngx_addon_dir/config; then  
            # 执行第三方模块的配置  
            . $ngx_addon_dir/config  
  
            echo " + $ngx_addon_name was configured"  
  
        else  
            echo "$0: error: no $ngx_addon_dir/config was found"  
            exit 1  
        fi  
    done  
fi
```

这段代码比较简单，确实实现了 nginx 很强大的扩展性，加载第三方模块。`$ngx_addon_dir` 变量是在 `configure` 执行时，命令行参数 `--add-module` 加入的，它是一个目录列表，每一个目录，表示一个第三方模块。从代码中，我们可以看到，它就是针对每一个第三方模块执行其目录下的 `config` 文件。于是我们可以在 `config` 文件里面执行我们自己的检测逻辑，比如检测库依赖，添加编译选项等。

15. auto/lib/conf

该文件会针对 nginx 编译所需要的基础库的检测，比如 `rewrite` 模块需要的 `PCRE` 库的检测支持。

16. configure 接下来定义一些宏常量，主要是是文件路径方面的：

```
case ".$NGX_PREFIX" in  
    .)  
        NGX_PREFIX=${NGX_PREFIX:-/usr/local/nginx}  
        have=NGX_PREFIX value="\\"$NGX_PREFIX/\"" . auto/define
```

```

;;

.!)
    NGX_PREFIX=
;;

*)
    have=NGX_PREFIX value="\$NGX_PREFIX/" . auto/define
;;
esac

if [ ".$NGX_CONF_PREFIX" != "." ]; then
    have=NGX_CONF_PREFIX value="\$NGX_CONF_PREFIX/" . auto/define
fi

have=NGX_SBIN_PATH value="\$NGX_SBIN_PATH/" . auto/define
have=NGX_CONF_PATH value="\$NGX_CONF_PATH/" . auto/define
have=NGX_PID_PATH value="\$NGX_PID_PATH/" . auto/define
have=NGX_LOCK_PATH value="\$NGX_LOCK_PATH/" . auto/define
have=NGX_ERROR_LOG_PATH value="\$NGX_ERROR_LOG_PATH/" . auto/define

have=NGX_HTTP_LOG_PATH value="\$NGX_HTTP_LOG_PATH/" . auto/define
have=NGX_HTTP_CLIENT_TEMP_PATH value="\$NGX_HTTP_CLIENT_TEMP_PATH/" . auto/define
have=NGX_HTTP_PROXY_TEMP_PATH value="\$NGX_HTTP_PROXY_TEMP_PATH/" . auto/define
have=NGX_HTTP_FASTCGI_TEMP_PATH value="\$NGX_HTTP_FASTCGI_TEMP_PATH/" . auto/define
have=NGX_HTTP_UWSGI_TEMP_PATH value="\$NGX_HTTP_UWSGI_TEMP_PATH/" . auto/define
have=NGX_HTTP_SCGI_TEMP_PATH value="\$NGX_HTTP_SCGI_TEMP_PATH/" . auto/define

```

17. configure 最后的工作，生成编译安装的 makefile

```

# 生成 objs/makefile 文件
. auto/make

# 生成关于库的编译选项到 makefile 文件
. auto/lib/make
# 生成与安装相关的 makefile 文件内容，并生成最外层的 makefile 文件
. auto/install

# STUB
. auto/stubs

have=NGX_USER value="\$NGX_USER/" . auto/define

```

```
have=NGX_GROUP value="\ "$NGX_GROUP\" . auto/define
```

```
# 编译的最后阶段，汇总信息  
. auto/summary
```

模块编译顺序

上一节中，提到过，nginx 模块的顺序很重要，会直接影响到程序的功能。而且，nginx 和部分模块，也有着自己特定的顺序要求，比如 ngx_http_write_filter_module 模块一定要在 filter 模块的最后一步执行。想查看模块的执行顺序，可以在 objs/nginx_modules.c 这个文件中找到，这个文件在 configure 之后生成，上一节中，我们看过这个文件里面的内容。

下面是一个 ngx_modules.c 文件的示例：

```
ngx_module_t *ngx_modules[] = {  
    // 全局 core 模块  
    &ngx_core_module,  
    &ngx_errlog_module,  
    &ngx_conf_module,  
    &ngx_emp_server_module,  
    &ngx_emp_server_core_module,  
  
    // event 模块  
    &ngx_events_module,  
    &ngx_event_core_module,  
    &ngx_kqueue_module,  
  
    // 正则模块  
    &ngx_regex_module,  
  
    // http 模块  
    &ngx_http_module,  
    &ngx_http_core_module,  
    &ngx_http_log_module,  
    &ngx_http_upstream_module,  
  
    // http handler 模块  
    &ngx_http_static_module,  
    &ngx_http_autoindex_module,  
    &ngx_http_index_module,  
    &ngx_http_auth_basic_module,  
    &ngx_http_access_module,  
    &ngx_http_limit_conn_module,  
    &ngx_http_limit_req_module,  
}
```

```

    &ngx_http_geo_module,
    &ngx_http_map_module,
    &ngx_http_split_clients_module,
    &ngx_http_referer_module,
    &ngx_http_rewrite_module,
    &ngx_http_proxy_module,
    &ngx_http_fastcgi_module,
    &ngx_http_uwsgi_module,
    &ngx_http_scgi_module,
    &ngx_http_memcached_module,
    &ngx_http_empty_gif_module,
    &ngx_http_browser_module,
    &ngx_http_upstream_ip_hash_module,
    &ngx_http_upstream_keepalive_module,
    //此处是第三方 handler 模块

    // http filter 模块
    &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_ssi_filter_module,
    &ngx_http_charset_filter_module,
    &ngx_http_userid_filter_module,
    &ngx_http_headers_filter_module,
    // 第三方 filter 模块
    &ngx_http_copy_filter_module,
    &ngx_http_range_body_filter_module,
    &ngx_http_not_modified_filter_module,
    NULL
};

```

http handler 模块与 http filter 模块的顺序很重要，这里我们主要关注一下这两类模块。

http handler 模块，在后面的章节里面会讲到多阶段请求的处理链。对于 content phase 之前的 handler，同一个阶段的 handler，模块是顺序执行的。比如上面的示例代码中，ngx_http_auth_basic_module 与 ngx_http_access_module 这两个模块都是在 access phase 阶段，由于 ngx_http_auth_basic_module 在前面，所以会先执行。由于 content phase 只会有一个执行，所以不存在顺序问题。另外，我们加载的第三方 handler 模块永远是在最后执行。

http filter 模块, filter 模块会将所有的 filter handler 排成一个倒序链,所以在最前面的最后执行。上面的例子中, ngx_http_write_filter_module 最后执行, ngx_http_not_modified_filter_module 最先执行。注意,我们加载的第三方 filter 模块是在 copy_filter 模块之后, headers_filter 模块之前执行。

nginx 的事件机制

event 框架及非阻塞模型

定时器实现

信号处理

惊群问题

nginx 的进程机制

master 进程

worker 进程

进程间通讯

nginx 基础设施

内存池

简介:

Nginx 里内存的使用大都十分有特色:申请了永久保存,抑或伴随着请求的结束而全部释放,还有写满了缓冲再从头接着写.这么做的原因也主要取决于 Web Server 的特殊的场景,内存的分配和请求相关,一条请求处理完毕,即可释放其相关的内存池,降低了开发中对内存资源管理的复杂度,也减少了内存碎片的存在.

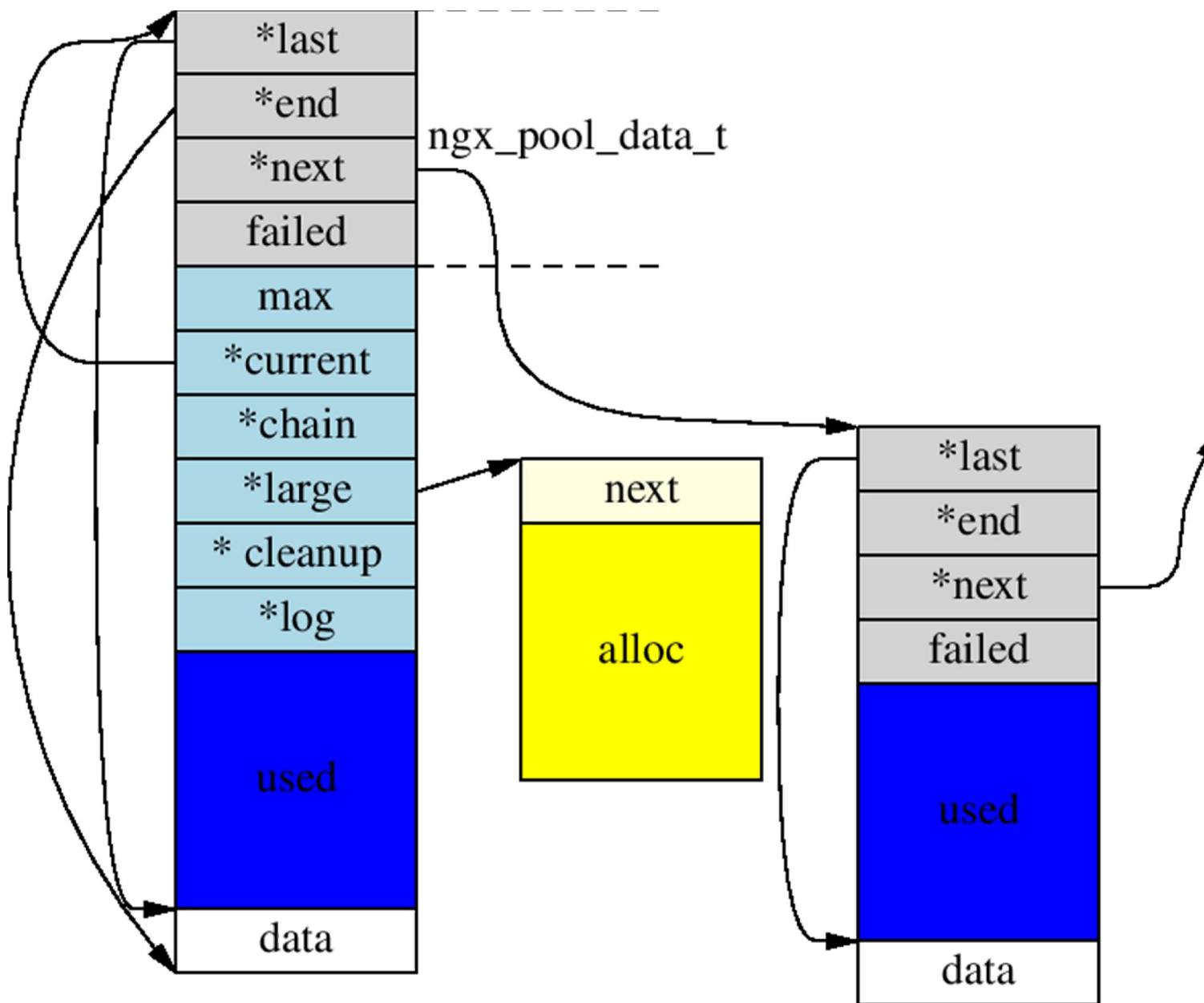
所以在 Nginx 使用内存池时总是只申请,不释放,使用完毕后直接 **destroy** 整个内存池.我们来看下内存池相关的实现。

结构:

```
struct ngx_pool_s {
    ngx_pool_data_t    d;
    size_t             max;
    ngx_pool_t         *current;
    ngx_chain_t        *chain;
    ngx_pool_large_t   *large;
    ngx_pool_cleanup_t *cleanup;
    ngx_log_t          *log;
};

struct ngx_pool_large_s {
    ngx_pool_large_t *next;
    void             *alloc;
};

typedef struct {
    u_char            *last;
    u_char            *end;
    ngx_pool_t        *next;
    ngx_uint_t        failed;
} ngx_pool_data_t;
```

实现:

这三个数据结构构成了基本的内存池的主体.通过 `ngx_create_pool` 可以创建一个内存池,通过 `ngx_palloc` 可以从内存池中分配指定大小的内存。

```
ngx_pool_t *
ngx_create_pool(size_t size, ngx_log_t *log)
{
    ngx_pool_t  *p;

    p = ngx_memalign(NGX_POOL_ALIGNMENT, size, log);
    if (p == NULL) {
        return NULL;
    }

    p->d.last = (u_char *) p + sizeof(ngx_pool_t);
```

```

    p->d.end = (u_char *) p + size;
    p->d.next = NULL;
    p->d.failed = 0;

    size = size - sizeof(ngx_pool_t);
    p->max = (size < NGX_MAX_ALLOC_FROM_POOL) ? size : NGX_MAX_ALLOC_FROM_POOL;

    p->current = p;
    p->chain = NULL;
    p->large = NULL;
    p->cleanup = NULL;
    p->log = log;

    return p;
}

```

这里首先申请了一块大小为 `size` 的内存区域，其前 `sizeof(ngx_pool_t)` 字节用来存储 `ngx_pool_t` 这个结构体自身。所以若 `size` 小于 `sizeof(ngx_pool_t)` 将会有 `coredump` 的可能性。

我们常用来分配内存的有三个接口：`ngx_palloc`，`ngx_pnalloc`，`ngx_pcalloc`。

分别来看下它们的实现：

```

void *
ngx_palloc(ngx_pool_t *pool, size_t size)
{
    u_char      *m;
    ngx_pool_t  *p;

    if (size <= pool->max) {

        p = pool->current;

        do {
            m = ngx_align_ptr(p->d.last, NGX_ALIGNMENT);

            if ((size_t) (p->d.end - m) >= size) {
                p->d.last = m + size;

                return m;
            }

            p = p->d.next;

        } while (p);
    }
}

```

```

        return ngx_palloc_block(pool, size);
    }

    return ngx_palloc_large(pool, size);
}

void *
ngx_pnalloc(ngx_pool_t *pool, size_t size)
{
    u_char      *m;
    ngx_pool_t   *p;

    if (size <= pool->max) {

        p = pool->current;

        do {
            m = p->d.last;

            if ((size_t) (p->d.end - m) >= size) {
                p->d.last = m + size;

                return m;
            }

            p = p->d.next;

        } while (p);

        return ngx_palloc_block(pool, size);
    }

    return ngx_palloc_large(pool, size);
}

void *
ngx_pcalloc(ngx_pool_t *pool, size_t size)
{
    void *p;

    p = ngx_palloc(pool, size);
    if (p) {
        ngx_memzero(p, size);
    }

    return p;
}

```

```
}
```

ngx_pccalloc 其只是 ngx_palloc 的一个封装，将申请到的内存全部初始化为 0。

ngx_palloc 相对 ngx_pnalloc，其会将申请的内存大小向上扩增到 NGX_ALIGNMENT 的倍数，以方便内存对齐，减少内存访问次数。

Nginx 的内存池不仅用于内存方面的管理，还可以通过`ngx_pool_cleanup_add`来添加内存池释放时的回调函数，以便用来释放自己申请的其他相关资源。

从代码中可以看出，这些由自己添加的释放回调是以链表形式保存的，也就是说你可以添加多个回调函数来管理不同的资源。

共享内存

slab 算法

buffer 管理

buffer 重用机制

buffer 防拷贝机制

chain 管理

chain 重用机制

aio 原理

锁实现

基本数据结构

时间缓存

文件缓存

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，我们想看懂 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。

2. 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;

```

3. 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
```

2. 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;
    }
}

```

3. 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。

4. 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储区地址。

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_palloc(cf->pool, sizeof(ngx_http_conf_ctx_t));

*(ngx_http_conf_ctx_t **) conf = ctx;

...

ctx->main_conf = ngx_palloc(cf->pool,
                             sizeof(void *) * ngx_http_max_module);

ctx->srv_conf = ngx_palloc(cf->pool, sizeof(void *) * ngx_http_max_module);

ctx->loc_conf = ngx_palloc(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，合并到路径上下文的操作是在 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;
#ifdef (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
#ifdef (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	checker	可自定义 handler
NGX_HTTP_POST_READ_PHASE	ngx_http_core_generic_phase	是
NGX_HTTP_SERVER_REWRITE_PHASE	ngx_http_core_rewrite_phase	是
NGX_HTTP_FIND_CONFIG_PHASE	ngx_http_core_find_config_phase	否
NGX_HTTP_REWRITE_PHASE	ngx_http_core_rewrite_phase	是
NGX_HTTP_POST_REWRITE_PHASE	ngx_http_core_post_rewrite_phase	否
NGX_HTTP_PREACCESS_PHASE	ngx_http_core_generic_phase	是
NGX_HTTP_ACCESS_PHASE	ngx_http_core_access_phase	是
NGX_HTTP_POST_ACCESS_PHASE	ngx_http_core_post_access_phase	否
NGX_HTTP_TRY_FILES_PHASE	ngx_http_core_try_files_phase	否
NGX_HTTP_CONTENT_PHASE	ngx_http_core_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;
```

2. 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。
3. 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

nginx 的请求处理阶段 (90%)

接收请求流程 (99%)

http 请求格式简介 (99%)

首先介绍一下 rfc2616 中定义的 http 请求基本格式:

```
Request = Request-Line
        * (( general-header
            | request-header
            | entity-header ) CRLF)
        CRLF
        [ message-body ]
```

第一行是请求行 (request line)，用来说明请求方法，要访问的资源以及所使用的 HTTP 版本:

```
Request-Line    = Method SP Request-URI SP HTTP-Version CRLF
```

请求方法 (Method) 的定义如下，其中最常用的是 GET, POST 方法:

```
Method = "OPTIONS"
| "GET"
| "HEAD"
| "POST"
| "PUT"
| "DELETE"
| "TRACE"
| "CONNECT"
```



```
| extension-method
extension-method = token
```

要访问的资源由统一资源地位符 URI(Uniform Resource Identifier)确定，它的一个比较通用的组成格式（rfc2396）如下：

```
<scheme>://<authority><path>?<query>
```

一般来说根据请求方法（Method）的不同，请求 URI 的格式会有所不同，通常只需写出 path 和 query 部分。

http 版本(version)定义如下，现在用的一般为 1.0 和 1.1 版本：

```
HTTP/<major>.<minor>
```

请求行的下一行则是请求头，rfc2616 中定义了 3 种不同类型的请求头，分别为 general-header，request-header 和 entity-header，每种类型 rfc 中都定义了一些通用的头，其中 entity-header 类型可以包含自定义的头。

请求头读取 (99%)

这一节介绍 nginx 中请求头的解析，nginx 的请求处理流程中，会涉及到 2 个非常重要的数据结构，ngx_connection_t 和 ngx_http_request_t，分别用来表示连接和请求，这 2 个数据结构在本书的前篇中已经做了比较详细的介绍，没有印象的读者可以翻回去复习一下，整个请求处理流程从头到尾，对应着这 2 个数据结构的分配，初始化，使用，重用和销毁。

nginx 在初始化阶段，具体是在 init process 阶段的 ngx_event_process_init 函数中会为每一个监听套接字分配一个连接结构（ngx_connection_t），并将该连接结构的读事件成员（read）的事件处理函数设置为 ngx_event_accept，并且如果没有使用 accept 互斥锁的话，在这个函数中会将该读事件挂载到 nginx 的事件处理模型上（poll 或者 epoll 等），反之则会等到 init process 阶段结束，在工作进程的事件处理循环中，某个进程抢到了 accept 锁才能挂载该读事件。

```
static ngx_int_t
ngx_event_process_init(ngx_cycle_t *cycle)
{
```

```

...

/* 初始化用来管理所有定时器的红黑树 */
if (ngx_event_timer_init(cycle->log) == NGX_ERROR) {
    return NGX_ERROR;
}
/* 初始化事件模型 */
for (m = 0; ngx_modules[m]; m++) {
    if (ngx_modules[m]->type != NGX_EVENT_MODULE) {
        continue;
    }

    if (ngx_modules[m]->ctx_index != ecf->use) {
        continue;
    }

    module = ngx_modules[m]->ctx;

    if (module->actions.init(cycle, ngx_timer_resolution) != NGX_OK) {
        /* fatal */
        exit(2);
    }

    break;
}

...

/* for each listening socket */
/* 为每个监听套接字分配一个连接结构 */
ls = cycle->listening.elts;
for (i = 0; i < cycle->listening.nelts; i++) {

    c = ngx_get_connection(ls[i].fd, cycle->log);

    if (c == NULL) {
        return NGX_ERROR;
    }

    c->log = &ls[i].log;

    c->listening = &ls[i];
    ls[i].connection = c;

    rev = c->read;

    rev->log = c->log;
    /* 标识此读事件为新请求连接事件 */

```

```

    rev->accept = 1;

    ...

#ifdef (NGX_WIN32)

    /* windows 环境下不做分析，但原理类似 */

#else

    /* 将读事件结构的处理函数设置为 ngx_event_accept */
    rev->handler = ngx_event_accept;
    /* 如果使用 accept 锁的话，要在后面抢到锁才能将监听句柄挂载上事件处理模型上 */
    if (ngx_use_accept_mutex) {
        continue;
    }
    /* 否则，将该监听句柄直接挂载上事件处理模型 */
    if (ngx_event_flags & NGX_USE_RTSIG_EVENT) {
        if (ngx_add_conn(c) == NGX_ERROR) {
            return NGX_ERROR;
        }

    } else {
        if (ngx_add_event(rev, NGX_READ_EVENT, 0) == NGX_ERROR) {
            return NGX_ERROR;
        }
    }

#endif

}

return NGX_OK;
}

```

当一个工作进程在某个时刻将监听事件挂载上事件处理模型之后，nginx 就可以正式接收并处理客户端过来的请求了。这时如果有一个用户在浏览器的地址栏内输入一个域名，并且域名解析服务器将该域名解析到一台由 nginx 监听的服务器上，nginx 的事件处理模型接收到这个读事件之后，会交给之前注册好的事件处理函数 `ngx_event_accept` 来处理。

在 `ngx_event_accept` 函数中，nginx 调用 `accept` 函数，从已连接队列得到一个连接以及对应的套接字，接着分配一个连接结构（`ngx_connection_t`），并将新得到的套接字保存在该连接结构中，这里还会做一些基本的连接初始化工作：

- 1, 首先给该连接分配一个内存池，初始大小默认为 256 字节，可通过 `connection_pool_size` 指令设置；

- 2, 分配日志结构, 并保存在其中, 以便后续的日志系统使用;
- 3, 初始化连接相应的 io 收发函数, 具体的 io 收发函数和使用的的事件模型及操作系统相关;
- 4, 分配一个套接口地址 (sockaddr), 并将 accept 得到的对端地址拷贝在其中, 保存在 sockaddr 字段;
- 5, 将本地套接口地址保存在 local_sockaddr 字段, 因为这个值是从监听结构 ngx_listening_t 中可得, 而监听结构中保存的只是配置文件中设置的监听地址, 但是配置的监听地址可能是通配符*, 即监听在所有的地址上, 所以连接中保存的这个值最终可能还会变动, 会被确定为真正的接收地址;
- 6, 将连接的写事件设置为已就绪, 即设置 ready 为 1, nginx 默认连接第一次为可写;
- 7, 如果监听套接字设置了 TCP_DEFER_ACCEPT 属性, 则表示该连接上已经有数据包过来, 于是设置读事件为就绪;
- 8, 将 sockaddr 字段保存的对端地址格式化为可读字符串, 并保存在 addr_text 字段;

最后调用 ngx_http_init_connection 函数初始化该连接结构的其他部分。

ngx_http_init_connection 函数最重要的工作是初始化读写事件的处理函数: 将该连接结构的写事件的处理函数设置为 ngx_http_empty_handler, 这个事件处理函数不会做任何操作, 实际上 nginx 默认连接第一次可写, 不会挂载写事件, 如果有数据需要发送, nginx 会直接写到这个连接, 只有在发生一次写不完的情况下, 才会挂载写事件到事件模型上, 并设置真正的写事件处理函数, 这里后面的章节还会做详细介绍; 读事件的处理函数设置为 ngx_http_init_request, 此时如果该连接上已经有数据过来(设置了 deferred accept), 则会直接调用 ngx_http_init_request 函数来处理该请求, 反之则设置一个定时器并在事件处理模型上挂载一个读事件, 等待数据到来或者超时。当然这里不管是已经有数据到来, 或者需要等待数据到来, 又或者等待超时, 最终都会进入读事件的处理函数-ngx_http_init_request。

ngx_http_init_request 函数主要工作即是初始化请求, 由于它是一个事件处理函数, 它只有唯一一个 ngx_event_t *类型的参数, ngx_event_t 结构在 nginx 中表示一个事件, 事件处理的上下文类似于一个中断处理的上下文, 为了在这个上下文得到相关的信息, nginx 中一般会将连接结构的引用保存在事件结构的 data 字段, 请求结构的引用则保存在连接结构的 data 字段, 这样在事件处理函数中可以方便

的得到对应的连接结构和请求结构。进入函数内部看一下，首先判断该事件是否是超时事件，如果是的话直接关闭连接并返回；反之则是指之前 `accept` 的连接上有请求过来需要处理。

`ngx_http_init_request` 函数首先在连接的内存池中为该请求分配一个 `ngx_http_request_t` 结构，这个结构将用来保存该请求所有的信息。分配完之后，这个结构的引用会被包存在连接的 `hc` 成员的 `request` 字段，以便于在长连接或 `pipelined` 请求中复用该请求结构。在这个函数中，`nginx` 根据该请求的接收端口和地址找到一个默认虚拟服务器配置（`listen` 指令的 `default_server` 属性用来标识一个默认虚拟服务器，否则监听在相同端口和地址的多个虚拟服务器，其中第一个定义的则为默认）。

`nginx` 配置文件中可以设置多个监听在不同端口和地址的虚拟服务器（每个 `server` 块对应一个虚拟服务器），另外还根据域名（`server_name` 指令可以配置该虚拟服务器对应的域名）来区分监听在相同端口和地址的虚拟服务器，每个虚拟服务器可以拥有不同的配置内容，而这些配置内容决定了 `nginx` 在接收到一个请求之后如何处理该请求。找到之后，相应的配置被保存在该请求对应的 `ngx_http_request_t` 结构中。注意这里根据端口和地址找到的默认配置只是临时使用一下，最终 `nginx` 会根据域名找到真正的虚拟服务器配置，随后的初始化工作还包括：

- 1, 将连接的读事件的处理函数设置为 `ngx_http_process_request_line` 函数，这个函数用来解析请求行，将请求的 `read_event_handler` 设置为 `ngx_http_block_reading` 函数，这个函数实际上什么都不做（当然在事件模型设置为水平触发时，唯一做的事情就是将事件从事件模型监听列表中删除，防止该事件一直被触发），后面会说到这里为什么会将 `read_event_handler` 设置为此函数；
- 2, 为这个请求分配一个缓冲区用来保存它的请求头，地址保存在 `header_in` 字段，默认大小为 1024 个字节，可以使用 `client_header_buffer_size` 指令修改，这里需要注意一下，`nginx` 用来保存请求头的缓冲区是在该请求所在连接的内存池中分配，而且会将地址保存一份在连接的 `buffer` 字段中，这样做的目的也是为了给该连接的下一次请求重用这个缓冲区，另外如果客户端发过来的请求头大于 1024 个字节，`nginx` 会重新分配更大的缓存区，默认用于大请求的头的缓冲区最大为 8K，最多 4 个，这 2 个值可以用 `large_client_header_buffers` 指令设置，后面还会说到请求行和一个请求头都不能超过一个最大缓冲区的大小；

3, 为这个请求分配一个内存池，后续所有与该请求相关的内存分配一般都会使用该内存池，默认大小为 4096 个字节，可以使用 `request_pool_size` 指令修改；

4, 为这个请求分配响应头链表，初始大小为 20；

5, 创建所有模块的上下文 `ctx` 指针数组，变量数据；

6, 将该请求的 `main` 字段设置为它本身，表示这是一个主请求，nginx 中对应的还有子请求概念，后面的章节会做详细的介绍；

7, 将该请求的 `count` 字段设置为 1，`count` 字段表示请求的引用计数；

8, 将当前时间保存在 `start_sec` 和 `start_msec` 字段，这个时间是该请求的起始时刻，将被用来计算一个请求的处理时间（request time），nginx 使用的这个起始点和 apache 略有差别，nginx 中请求的起始点是接收到客户端的第一个数据包的事件开始，而 apache 则是接收到客户端的整个 request line 后开始算起；

9, 初始化请求的其他字段，比如将 `uri_changes` 设置为 11，表示最多可以将该请求的 uri 改写 10 次，`subrequests` 被设置为 201，表示一个请求最多可以发起 200 个子请求；

做完所有这些初始化工作之后，`ngx_http_init_request` 函数会调用读事件的处理函数来真正的解析客户端发过来的数据，也就是会进入 `ngx_http_process_request_line` 函数中处理。

解析请求行 (99%)

`ngx_http_process_request_line` 函数的主要作用即是解析请求行，同样由于涉及到网络 IO 操作，即使是很短的一行请求行可能也不能被一次读完，所以在之前的 `ngx_http_init_request` 函数中，

`ngx_http_process_request_line` 函数被设置为读事件的处理函数，它也只拥有一个唯一的 `ngx_event_t *` 类型参数，并且在函数的开头，同样需要判断是否是超时事件，如果是的话，则关闭这个请求和连接；否则开始正常的解析流程。先调用 `ngx_http_read_request_header` 函数读取数据。

由于可能多次进入 `ngx_http_process_request_line` 函数，`ngx_http_read_request_header` 函数首先检查请求的 `header_in` 指向的缓冲区内是否有数据，有的话直接返回；否则从连接读取数据并保存在请求的

header_in 指向的缓存区，而且只要缓冲区有空间的话，会一次尽可能多的读数据，读到多少返回多少；

如果客户端暂时没有发任何数据过来，并返回 NGX_AGAIN，返回之前会做 2 件事情：

- 1，设置一个定时器，时长默认为 60s，可以通过指令 client_header_timeout 设置，如果定时事件到达之前没有任何可读事件，nginx 将会关闭此请求；

- 2，调用 ngx_handle_read_event 函数处理一下读事件-如果该连接尚未在事件处理模型上挂载读事件，则将其挂载上；

如果客户端提前关闭了连接或者读取数据发生了其他错误，则给客户端返回一个 400 错误（当然这里并不保证客户端能够接收到响应数据，因为客户端可能都已经关闭了连接），最后函数返回 NGX_ERROR；

如果 ngx_http_read_request_header 函数正常的读取到了数据，ngx_http_process_request_line 函数将调用 ngx_http_parse_request_line 函数来解析，这个函数根据 http 协议规范中对请求行的定义实现了一个有限状态机，经过这个状态机，nginx 会记录请求行中的请求方法（Method），请求 uri 以及 http 协议版本在缓冲区中的起始位置，在解析过程中还会记录一些其他有用的信息，以便后面的处理过程中使用。如果解析请求行的过程中没有产生任何问题，该函数会返回 NGX_OK；如果请求行不满足协议规范，该函数会立即终止解析过程，并返回相应错误号；如果缓冲区数据不够，该函数返回 NGX_AGAIN。

在整个解析 http 请求的状态机中始终遵循着两条重要的原则：减少内存拷贝和回溯。

内存拷贝是一个相对比较昂贵的操作，大量的内存拷贝会带来较低的运行时效率。nginx 在需要做内存拷贝的地方尽量只拷贝内存的起始和结束地址而不是内存本身，这样做的话仅仅只需要两个赋值操作而已，大大降低了开销，当然这样带来的影响是后续的操作不能修改内存本身，如果修改的话，会影响到所有引用到该内存区间的地方，所以必须很小心的管理，必要的时候需要拷贝一份。

这里不得不提到 nginx 中最能体现这一思想的数据结构，ngx_buf_t，它用来表示 nginx 中的缓存，在很多情况下，只需要将一块内存的起始地址和结束地址分别保存在它的 pos 和 last 成员中，再将它的 memory 标志置 1，即可表示一块不能修改的内存区间，在另外的需要一块能够修改的缓存的情形中，则必须分配一块所需大小的内存并保存其起始地址，再将 ngx_buf_t 的 temporary 标志置 1，表示这是一块能够被修改的内存区域。

再回到 `ngx_http_process_request_line` 函数中，如果 `ngx_http_parse_request_line` 函数返回了错误，则直接给客户端返回 400 错误； 如果返回 `NGX_AGAIN`，则需要判断一下是否是由于缓冲区空间不够，还是已读数据不够。如果是缓冲区大小不够了，nginx 会调用 `ngx_http_alloc_large_header_buffer` 函数来分配另一块大缓冲区，如果大缓冲区还不够装下整个请求行，nginx 则会返回 414 错误给客户端，否则分配了更大的缓冲区并拷贝之前的数据之后，继续调用 `ngx_http_read_request_header` 函数读取数据来进入请求行自动机处理，直到请求行解析结束；

如果返回了 `NGX_OK`，则表示请求行被正确的解析出来了，这时先记录好请求行的起始地址以及长度，并将请求 uri 的 path 和参数部分保存在请求结构的 uri 字段，请求方法起始位置和长度保存在 `method_name` 字段，http 版本起始位置和长度记录在 `http_protocol` 字段。还要从 uri 中解析出参数以及请求资源的拓展名，分别保存在 `args` 和 `exten` 字段。接下来将要解析请求头，将在下一小节中接着介绍。

解析请求头 (99%)

在 `ngx_http_process_request_line` 函数中，解析完请求行之后，如果请求行的 uri 里面包含了域名部分，则将其保存在请求结构的 `headers_in` 成员的 `server` 字段，`headers_in` 用来保存所有请求头，它的类型为 `ngx_http_headers_in_t`：

```
typedef struct {
    ngx_list_t          headers;

    ngx_table_elt_t     *host;
    ngx_table_elt_t     *connection;
    ngx_table_elt_t     *if_modified_since;
    ngx_table_elt_t     *if_unmodified_since;
    ngx_table_elt_t     *user_agent;
    ngx_table_elt_t     *referer;
    ngx_table_elt_t     *content_length;
    ngx_table_elt_t     *content_type;

    ngx_table_elt_t     *range;
    ngx_table_elt_t     *if_range;

    ngx_table_elt_t     *transfer_encoding;
    ngx_table_elt_t     *expect;
```



```

#if (NGX_HTTP_GZIP)
    ngx_table_elt_t          *accept_encoding;
    ngx_table_elt_t          *via;
#endif

    ngx_table_elt_t          *authorization;

    ngx_table_elt_t          *keep_alive;

#if (NGX_HTTP_PROXY || NGX_HTTP_REALIP || NGX_HTTP_GEO)
    ngx_table_elt_t          *x_forwarded_for;
#endif

#if (NGX_HTTP_REALIP)
    ngx_table_elt_t          *x_real_ip;
#endif

#if (NGX_HTTP_HEADERS)
    ngx_table_elt_t          *accept;
    ngx_table_elt_t          *accept_language;
#endif

#if (NGX_HTTP_DAV)
    ngx_table_elt_t          *depth;
    ngx_table_elt_t          *destination;
    ngx_table_elt_t          *overwrite;
    ngx_table_elt_t          *date;
#endif

    ngx_str_t                user;
    ngx_str_t                passwd;

    ngx_array_t              cookies;

    ngx_str_t                server;
    off_t                    content_length_n;
    time_t                   keep_alive_n;

    unsigned                 connection_type:2;
    unsigned                 msie:1;
    unsigned                 msie6:1;
    unsigned                 opera:1;
    unsigned                 gecko:1;
    unsigned                 chrome:1;
    unsigned                 safari:1;
    unsigned                 konqueror:1;
} ngx_http_headers_in_t;

```

接着，该函数会检查进来的请求是否使用的是 http0.9，如果是的话则使用从请求行里得到的域名，调用 ngx_http_find_virtual_server（）函数来查找用来处理该请求的虚拟服务器配置，之前通过端口和地址找到的默认配置不再使用，找到相应的配置之后，则直接调用 ngx_http_process_request（）函数处理该请求，因为 http0.9 是最原始的 http 协议，它里面没有定义任何请求头，显然就不需要读取请求头的操作。

```
if (r->host_start && r->host_end) {

    host = r->host_start;
    n = ngx_http_validate_host(r, &host,
                               r->host_end - r->host_start, 0);

    if (n == 0) {
        ngx_log_error(NGX_LOG_INFO, c->log, 0,
                      "client sent invalid host in request line");
        ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
        return;
    }

    if (n < 0) {
        ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
        return;
    }

    r->headers_in.server.len = n;
    r->headers_in.server.data = host;
}

if (r->http_version < NGX_HTTP_VERSION_10) {

    if (ngx_http_find_virtual_server(r, r->headers_in.server.data,
                                     r->headers_in.server.len)
        == NGX_ERROR)
    {
        ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
        return;
    }

    ngx_http_process_request(r);
    return;
}
```

当然，如果是 1.0 或者更新的 http 协议，接下来要做的就是读取请求头了，首先 nginx 会为请求头分配空间，ngx_http_headers_in_t 结构的 headers 字段为一个链表结构，它被用来保存所有请求头，初始为它分配了 20 个节点，每个节点的类型为 ngx_table_elt_t，保存请求头的 name/value 值对，还可以看到 ngx_http_headers_in_t 结构有很多类型为 ngx_table_elt_t* 的指针成员，而且从它们的命名可以看出是一些常见的请求头名字，nginx 对这些常用的请求头在 ngx_http_headers_in_t 结构里面保存了一份引用，后续需要使用的话，可以直接通过这些成员得到，另外也事先为 cookie 头分配了 2 个元素的数组空间，做完这些内存准备工作之后，该请求对应的读事件结构的处理函数被设置为 ngx_http_process_request_headers，并随后马上调用了该函数。

```
if (ngx_list_init(&r->headers_in.headers, r->pool, 20,
                 sizeof(ngx_table_elt_t))
    != NGX_OK)
{
    ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
    return;
}

if (ngx_array_init(&r->headers_in.cookies, r->pool, 2,
                  sizeof(ngx_table_elt_t *))
    != NGX_OK)
{
    ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
    return;
}

c->log->action = "reading client request headers";

rev->handler = ngx_http_process_request_headers;
ngx_http_process_request_headers(rev);
```

ngx_http_process_request_headers 函数循环的读取所有的请求头，并保存和初始化和请求头相关的结构，下面详细分析一下该函数：

因为 nginx 对读取请求头有超时限制，ngx_http_process_request_headers 函数作为读事件处理函数，一并处理了超时事件，如果读超时了，nginx 直接给该请求返回 408 错误：

```
if (rev->timedout) {
    ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT, "client timed out");
    c->timedout = 1;
```

```
    ngx_http_close_request(r, NGX_HTTP_REQUEST_TIME_OUT);  
    return;  
}
```

读取和解析请求头的逻辑和处理请求行差不多，总的流程也是循环的调用

`ngx_http_read_request_header()` 函数读取数据，然后再调用一个解析函数来从读取的数据中解析请求头，直到解析完所有请求头，或者发生解析错误为主。当然由于涉及到网络 io，这个流程可能发生在多个 io 事件的上下文中。

接着来细看该函数，先调用了 `ngx_http_read_request_header()` 函数读取数据，如果当前连接并没有数据过来，再直接返回，等待下一次读事件到来，如果读到了一些数据则调用

`ngx_http_parse_header_line()` 函数来解析，同样的该解析函数实现为一个有限状态机，逻辑很简单，

只是根据 http 协议来解析请求头，每次调用该函数最多解析出一个请求头，该函数返回 4 种不同返回值，表示不同解析结果：

1, 返回 `NGX_OK`，表示解析出了一行请求头，这时还要判断解析出的请求头名字里面是否有非法字符，名字里面合法的字符包括字母，数字和连字符 (-)，另外如果设置了 `underscores_in_headers` 指令为 `on`，则下划线也是合法字符，但是 `nginx` 默认下划线不合法，当请求头里面包含了非法的字符，`nginx` 默认只是忽略这一行请求头；如果一切都正常，`nginx` 会将该请求头及请求头名字的 hash 值保存在请求结构体的 `headers_in` 成员的 `headers` 链表,而且对于一些常见的请求头，如 `Host`, `Connection`, `nginx` 采用了类似于配置指令的方式，事先给这些请求头分配了一个处理函数，当解析出一个请求头时，会检查该请求头是否有设置处理函数，有的话则调用之，`nginx` 所有有处理函数的请求头都记录在 `ngx_http_headers_in` 全局数组中：

```
typedef struct {  
    ngx_str_t          name;  
    ngx_uint_t         offset;  
    ngx_http_header_handler_pt handler;  
} ngx_http_header_t;  
  
ngx_http_header_t  ngx_http_headers_in[] = {  
    { ngx_string("Host"), offsetof(ngx_http_headers_in_t, host),  
      ngx_http_process_host },  
  
    { ngx_string("Connection"), offsetof(ngx_http_headers_in_t, connection),  
      ngx_http_process_connection },
```

```

{ ngx_string("If-Modified-Since"),
  offsetof(ngx_http_headers_in_t, if_modified_since),
  ngx_http_process_unique_header_line },

{ ngx_string("If-Unmodified-Since"),
  offsetof(ngx_http_headers_in_t, if_unmodified_since),
  ngx_http_process_unique_header_line },

{ ngx_string("User-Agent"), offsetof(ngx_http_headers_in_t, user_agent),
  ngx_http_process_user_agent },

{ ngx_string("Referer"), offsetof(ngx_http_headers_in_t, referer),
  ngx_http_process_header_line },

{ ngx_string("Content-Length"),
  offsetof(ngx_http_headers_in_t, content_length),
  ngx_http_process_unique_header_line },

{ ngx_string("Content-Type"),
  offsetof(ngx_http_headers_in_t, content_type),
  ngx_http_process_header_line },

{ ngx_string("Range"), offsetof(ngx_http_headers_in_t, range),
  ngx_http_process_header_line },

{ ngx_string("If-Range"),
  offsetof(ngx_http_headers_in_t, if_range),
  ngx_http_process_unique_header_line },

{ ngx_string("Transfer-Encoding"),
  offsetof(ngx_http_headers_in_t, transfer_encoding),
  ngx_http_process_header_line },

{ ngx_string("Expect"),
  offsetof(ngx_http_headers_in_t, expect),
  ngx_http_process_unique_header_line },

#if (NGX_HTTP_GZIP)
{ ngx_string("Accept-Encoding"),
  offsetof(ngx_http_headers_in_t, accept_encoding),
  ngx_http_process_header_line },

{ ngx_string("Via"), offsetof(ngx_http_headers_in_t, via),
  ngx_http_process_header_line },
#endif

{ ngx_string("Authorization"),

```

```

        offsetof(ngx_http_headers_in_t, authorization),
        ngx_http_process_unique_header_line },

    { ngx_string("Keep-Alive"), offsetof(ngx_http_headers_in_t, keep_alive),
      ngx_http_process_header_line },

#ifdef NGX_HTTP_PROXY || NGX_HTTP_REALIP || NGX_HTTP_GEO
    { ngx_string("X-Forwarded-For"),
      offsetof(ngx_http_headers_in_t, x_forwarded_for),
      ngx_http_process_header_line },
#endif

#ifdef NGX_HTTP_REALIP
    { ngx_string("X-Real-IP"),
      offsetof(ngx_http_headers_in_t, x_real_ip),
      ngx_http_process_header_line },
#endif

#ifdef NGX_HTTP_HEADERS
    { ngx_string("Accept"), offsetof(ngx_http_headers_in_t, accept),
      ngx_http_process_header_line },

    { ngx_string("Accept-Language"),
      offsetof(ngx_http_headers_in_t, accept_language),
      ngx_http_process_header_line },
#endif

#ifdef NGX_HTTP_DAV
    { ngx_string("Depth"), offsetof(ngx_http_headers_in_t, depth),
      ngx_http_process_header_line },

    { ngx_string("Destination"), offsetof(ngx_http_headers_in_t, destination),
      ngx_http_process_header_line },

    { ngx_string("Overwrite"), offsetof(ngx_http_headers_in_t, overwrite),
      ngx_http_process_header_line },

    { ngx_string("Date"), offsetof(ngx_http_headers_in_t, date),
      ngx_http_process_header_line },
#endif

    { ngx_string("Cookie"), 0, ngx_http_process_cookie },

    { ngx_null_string, 0, NULL }
};

```

ngx_http_headers_in 数组当前包含了 25 个常用的请求头，每个请求头都设置了一个处理函数，其中一部分请求头设置的是公共处理函数，这里有 2 个公共处理函数，ngx_http_process_header_line 和 ngx_http_process_unique_header_line。 先来看一下处理函数的函数指针定义：

```
typedef ngx_int_t (*ngx_http_header_handler_pt)(ngx_http_request_t *r,  
        ngx_table_elt_t *h, ngx_uint_t offset);
```

它有 3 个参数，r 为对应的请求结构，h 为指向该请求头在 headers_in.headers 链表中对应节点的指针，offset 为该请求头对应字段在 ngx_http_headers_in_t 结构中的偏移。

再来看 ngx_http_process_header_line 函数：

```
static ngx_int_t  
ngx_http_process_header_line(ngx_http_request_t *r, ngx_table_elt_t *h,  
        ngx_uint_t offset)  
{  
    ngx_table_elt_t **ph;  
  
    ph = (ngx_table_elt_t **) ((char *) &r->headers_in + offset);  
  
    if (*ph == NULL) {  
        *ph = h;  
    }  
  
    return NGX_OK;  
}
```

这个函数只是简单将该请求头在 ngx_http_headers_in_t 结构中保存一份引用。

ngx_http_process_unique_header_line 功能类似，不同点在于该函数会检查这个请求头是否是重复的，如果是的话，则给该请求返回 400 错误。

ngx_http_headers_in 数组中剩下的请求头都有自己特殊的处理函数，这些特殊的函数根据对应的请求头有一些特殊的处理，下面拿 Host 头的处理函数 ngx_http_process_host 做一下介绍：

```
static ngx_int_t  
ngx_http_process_host(ngx_http_request_t *r, ngx_table_elt_t *h,  
        ngx_uint_t offset)  
{  
    u_char    *host;
```

```

ssize_t    len;

if (r->headers_in.host == NULL) {
    r->headers_in.host = h;
}

host = h->value.data;
len = ngx_http_validate_host(r, &host, h->value.len, 0);

if (len == 0) {
    ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
        "client sent invalid host header");
    ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
    return NGX_ERROR;
}

if (len < 0) {
    ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
    return NGX_ERROR;
}

if (r->headers_in.server.len) {
    return NGX_OK;
}

r->headers_in.server.len = len;
r->headers_in.server.data = host;

return NGX_OK;
}

```

此函数的目的也是保存 Host 头的快速引用，它会对 Host 头的值做一些合法性检查，并从中解析出域名，保存在 headers_in.server 字段，实际上前面在解析请求行时，headers_in.server 可能已经被赋值为从请求行中解析出来的域名，根据 http 协议的规范，如果请求行中的 uri 带有域名的话，则域名以它为准，所以这里需检查一下 headers_in.server 是否为空，如果不为空则不需要再赋值。

其他请求头的特殊处理函数，不再做介绍，大致都是根据该请求头在 http 协议中规定的意义及其值设置请求的一些属性，必备后续使用。

对一个合法的请求头的处理大致为如上所述；

2，返回 NGX_AGAIN，表示当前接收到的数据不够，一行请求头还未结束，需要继续下一轮循环。在下一轮循环中，nginx 首先检查请求头缓冲区 header_in 是否已满，如够满了，则调用

ngx_http_alloc_large_header_buffer() 函数分配更多缓冲区，下面分析一下

ngx_http_alloc_large_header_buffer 函数：

```
static ngx_int_t
ngx_http_alloc_large_header_buffer(ngx_http_request_t *r,
    ngx_uint_t request_line)
{
    u_char          *old, *new;
    ngx_buf_t       *b;
    ngx_http_connection_t *hc;
    ngx_http_core_srv_conf_t *cscf;

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
        "http alloc large header buffer");

    /*
     * 在解析请求行阶段，如果客户端在发送请求行之前发送了大量回车换行符将
     * 缓冲区塞满了，针对这种情况，nginx 只是简单的重置缓冲区，丢弃这些垃圾
     * 数据，不需要分配更大的内存。
     */
    if (request_line && r->state == 0) {

        /* the client fills up the buffer with "\r\n" */

        r->request_length += r->header_in->end - r->header_in->start;

        r->header_in->pos = r->header_in->start;
        r->header_in->last = r->header_in->start;

        return NGX_OK;
    }

    /* 保存请求行或者请求头在旧缓冲区中的起始地址 */
    old = request_line ? r->request_start : r->header_name_start;

    cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);

    /* 如果一个大缓冲区还装不下请求行或者一个请求头，则返回错误 */
    if (r->state != 0
        && (size_t) (r->header_in->pos - old)
            >= cscf->large_client_header_buffers.size)
    {
        return NGX_DECLINED;
    }

    hc = r->http_connection;
```

```

/* 首先在 ngx_http_connection_t 结构中查找是否有空闲缓冲区，有的话，直接取之 */
if (hc->nfree) {
    b = hc->free[--hc->nfree];

    ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
        "http large header free: %p %uz",
        b->pos, b->end - b->last);

/* 检查给该请求分配的请求头缓冲区个数是否已经超过限制，默认最大个数为 4 个 */
} else if (hc->nbusy < cscf->large_client_header_buffers.num) {

    if (hc->busy == NULL) {
        hc->busy = ngx_palloc(r->connection->pool,
            cscf->large_client_header_buffers.num * sizeof(ngx_buf_t *));
        if (hc->busy == NULL) {
            return NGX_ERROR;
        }
    }

/* 如果还没有达到最大分配数量，则分配一个新的大缓冲区 */
    b = ngx_create_temp_buf(r->connection->pool,
        cscf->large_client_header_buffers.size);

    if (b == NULL) {
        return NGX_ERROR;
    }

    ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
        "http large header alloc: %p %uz",
        b->pos, b->end - b->last);

} else {
    /* 如果已经达到最大的分配限制，则返回错误 */
    return NGX_DECLINED;
}

/* 将从空闲队列取得的或者新分配的缓冲区加入已使用队列 */
hc->busy[hc->nbusy++] = b;

/*
 * 因为 nginx 中，所有的请求头的保存形式都是指针（起始和结束地址），
 * 所以一行完整的请求头必须放在连续的内存块中。如果旧的缓冲区不能
 * 再放下整行请求头，则分配新缓冲区，并从旧缓冲区拷贝已经读取的部分请求头，
 * 拷贝完之后，需要修改所有相关指针指向到新缓冲区。
 * status 为 0 表示解析完一行请求头之后，缓冲区正好被用完，这种情况不需要拷贝
 */
if (r->state == 0) {
    /*
     * r->state == 0 means that a header line was parsed successfully

```

```

    * and we do not need to copy incomplete header line and
    * to relocate the parser header pointers
    */

    r->request_length += r->header_in->end - r->header_in->start;

    r->header_in = b;

    return NGX_OK;
}

ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
               "http large header copy: %d", r->header_in->pos - old);

r->request_length += old - r->header_in->start;

new = b->start;

/* 拷贝旧缓冲区中不完整的请求头 */
ngx_memcpy(new, old, r->header_in->pos - old);

b->pos = new + (r->header_in->pos - old);
b->last = new + (r->header_in->pos - old);

/* 修改相应的指针指向新缓冲区 */
if (request_line) {
    r->request_start = new;

    if (r->request_end) {
        r->request_end = new + (r->request_end - old);
    }

    r->method_end = new + (r->method_end - old);

    r->uri_start = new + (r->uri_start - old);
    r->uri_end = new + (r->uri_end - old);

    if (r->schema_start) {
        r->schema_start = new + (r->schema_start - old);
        r->schema_end = new + (r->schema_end - old);
    }

    if (r->host_start) {
        r->host_start = new + (r->host_start - old);
        if (r->host_end) {
            r->host_end = new + (r->host_end - old);
        }
    }
}

```

```

    if (r->port_start) {
        r->port_start = new + (r->port_start - old);
        r->port_end = new + (r->port_end - old);
    }

    if (r->uri_ext) {
        r->uri_ext = new + (r->uri_ext - old);
    }

    if (r->args_start) {
        r->args_start = new + (r->args_start - old);
    }

    if (r->http_protocol.data) {
        r->http_protocol.data = new + (r->http_protocol.data - old);
    }

} else {
    r->header_name_start = new;
    r->header_name_end = new + (r->header_name_end - old);
    r->header_start = new + (r->header_start - old);
    r->header_end = new + (r->header_end - old);
}

r->header_in = b;

return NGX_OK;
}

```

当 `ngx_http_alloc_large_header_buffer` 函数返回 `NGX_DECLINED` 时，表示客户端发送了一行过大的请求头，或者是整个请求头部超过了限制，nginx 会返回 494 错误，注意到 nginx 在返回 494 错误之前将请求的 `lingering_close` 标识置为了 1，这样做的目的是在返回响应之前丢弃掉客户端发过来的其他数据；

3，返回 `NGX_HTTP_PARSE_INVALID_HEADER`，表示请求头解析过程中遇到错误，一般为客户端发送了不符合协议规范的头部，此时 nginx 返回 400 错误；

4，返回 `NGX_HTTP_PARSE_HEADER_DONE`，表示所有请求头已经成功的解析，这时请求的状态被置为 `NGX_HTTP_PROCESS_REQUEST_STATE`，意味着结束了请求读取阶段，正式进入了请求处理阶段，但是实际上请求可能含有请求体，nginx 在请求读取阶段并不会去读取请求体，这个工作交给了后续的请求处理阶段的模块，这样做的目的是 nginx 本身并不知道这些请求体是否有用，如果后续模块并不

需要的话，一方面请求体一般较大，如果全部读取进内存，则白白耗费大量的内存空间，另一方面即使 nginx 将请求体写进磁盘，但是涉及到磁盘 io，会耗费比较多时间。所以交由后续模块来决定读取还是丢弃请求体是最明智的办法。

读取完请求头之后，nginx 调用了 `ngx_http_process_request_header()` 函数，这个函数主要做了两个方面的事情，一是调用 `ngx_http_find_virtual_server()` 函数查找虚拟服务器配置；二是对一些请求头做一些协议的检查。比如对那些使用 http1.1 协议但是却没有发送 Host 头的请求，nginx 给这些请求返回 400 错误。还有 nginx 现在的版本并不支持 chunked 格式的输入，如果某些请求申明自己使用了 chunked 格式的输入（请求带有值为 chunked 的 transfer_encoding 头部），nginx 给这些请求返回 411 错误。等等。

最后调用 `ngx_http_process_request()` 函数处理请求,至此，nginx 请求头接收流程就介绍完毕。

请求体读取(100%)

上节说到 nginx 核心本身不会主动读取请求体，这个工作是交给请求处理阶段的模块来做，但是 nginx 核心提供了 `ngx_http_read_client_request_body()` 接口来读取请求体，另外还提供了一个丢弃请求体的接口-`ngx_http_discard_request_body()`，在请求执行的各个阶段中，任何一个阶段的模块如果对请求体感兴趣或者希望丢掉客户端发过来的请求体，可以分别调用这两个接口来完成。这两个接口是 nginx 核心提供的处理请求体的标准接口，如果希望配置文件中一些请求体相关的指令（比如 `client_body_in_file_only`，`client_body_buffer_size` 等）能够预期工作，以及能够正常使用 nginx 内置的一些和请求体相关的变量（比如 `$request_body` 和 `$request_body_file`），一般来说所有模块都必须调用这些接口来完成相应操作，如果需要自定义接口来处理请求体，也应尽量兼容 nginx 默认的行为。

读取请求体

请求体的读取一般发生在 nginx 的 content handler 中，一些 nginx 内置的模块，比如 proxy 模块，fastcgi 模块，uwsgi 模块等，这些模块的行为必须将客户端过来的请求体（如果有的话）以相应协议完整的转发到后端服务进程，所有的这些模块都是调用了 `ngx_http_read_client_request_body()` 接口来完成请求体读取。值得注意的是这些模块会把客户端的请求体完整的读取后才开始往后端转发数据。

由于内存的限制，`ngx_http_read_client_request_body()`接口读取的请求体会部分或者全部写入一个临时文件中，根据请求体的大小以及相关的指令配置，请求体可能完整放置在一块连续内存中，也可能分别放置在两块不同内存中，还可能全部存在一个临时文件中，最后还可能一部分在内存，剩余部分在临时文件中。下面先介绍一下和这些不同存储行为相关的指令：

client_body_buffer_size:

设置缓存请求体的 `buffer` 大小，默认为系统页大小的 2 倍，当请求体的大小超过此大小时，`nginx` 会把请求体写入到临时文件中。可以根据业务需求设置合适的大小，尽量避免磁盘 `io` 操作；

client_body_in_single_buffer:

指示是否将请求体完整的存储在一块连续的内存中，默认为 `off`，如果此指令被设置为 `on`，则 `nginx` 会保证请求体在不大于 `client_body_buffer_size` 设置的值时，被存放在一块连续的内存中，但超过大小时会被整个写入一个临时文件；

client_body_in_file_only:

设置是否总是将请求体保存在临时文件中，默认为 `off`，当此指令被设置为 `on` 时，即使客户端显式指示了请求体长度为 0 时，`nginx` 还是会为请求创建一个临时文件。

接着介绍 `ngx_http_read_client_request_body()`接口的实现，它的定义如下：

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

该接口有 2 个参数，第 1 个为指向请求结构的指针，第 2 个为一个函数指针，当请求体读完时，它会被调用。之前也说到根据 `nginx` 现有行为，模块逻辑会在请求体读完后执行，这个回调函数一般就是模块的逻辑处理函数。`ngx_http_read_client_request_body()`函数首先将参数 `r` 对应的主请求的引用加 1，这样做的目的和该接口被调用的上下文有关，一般而言，模块是在 `content handler` 中调用此接口，一个典型的调用如下：

```
static ngx_int_t
ngx_http_proxy_handler(ngx_http_request_t *r)
{
    ...
    rc = ngx_http_read_client_request_body(r, ngx_http_upstream_init);

    if (rc >= NGX_HTTP_SPECIAL_RESPONSE) {
        return rc;
    }

    return NGX_DONE;
}
```

上面的代码是在 proxy 模块的 content handler, ngx_http_proxy_handler()中调用了

ngx_http_read_client_request_body()函数, 其中 ngx_http_upstream_init()被作为回调函数传入进接口中, 另外 nginx 中模块的 content handler 调用的上下文如下:

```
ngx_int_t
ngx_http_core_content_phase(ngx_http_request_t *r,
    ngx_http_phase_handler_t *ph)
{
    ...
    if (r->content_handler) {
        r->write_event_handler = ngx_http_request_empty_handler;
        ngx_http_finalize_request(r, r->content_handler(r));
        return NGX_OK;
    }
    ...
}
```

上面的代码中, content handler 调用之后, 它的返回值作为参数调用了 ngx_http_finalize_request()函数, 在请求体没有被接收完全时, ngx_http_read_client_request_body()函数返回值为 NGX_AGAIN, 此时 content handler, 比如 ngx_http_proxy_handler()会返回 NGX_DONE, 而 NGX_DONE 作为参数传给 ngx_http_finalize_request()函数会导致主请求的引用计数减 1, 所以正好抵消了 ngx_http_read_client_request_body()函数开头对主请求计数的加 1。

接下来回到 ngx_http_read_client_request_body()函数, 它会检查该请求的请求体是否已经被读取或者被丢弃了, 如果是的话, 则直接调用回调函数并返回 NGX_OK, 这里实际上是为子请求检查, 子请求是 nginx 中的一个概念, nginx 中可以在当前请求中发起另外一个或多个全新的子请求来访问其他的 location, 关于子请求的具体介绍会在后面的章节作详细分析, 一般而言子请求不需要自己去读取请求体。

函数接着调用 ngx_http_test_expect()检查客户端是否发送了 Expect: 100-continue 头, 是的话则给客户端回复“HTTP/1.1 100 Continue”, 根据 http 1.1 协议, 客户端可以发送一个 Expect 头来向服务器表明期望发送请求体, 服务器如果允许客户端发送请求体, 则会回复“HTTP/1.1 100 Continue”, 客户端收到时, 才会开始发送请求体。

接着继续为接收请求体做准备工作，分配一个 `ngx_http_request_body_t` 结构，并保存在

`r->request_body`，这个结构用来保存请求体读取过程用到的缓存引用，临时文件引用，剩余请求体大小等信息，它的定义如下：

```
typedef struct {
    ngx_temp_file_t      *temp_file;
    ngx_chain_t          *bufs;
    ngx_buf_t            *buf;
    off_t                 rest;
    ngx_chain_t          *to_write;
    ngx_http_client_body_handler_pt post_handler;
} ngx_http_request_body_t;
```

temp_file: 指向储存请求体的临时文件的指针；

bufs: 指向保存请求体的链表头；

buf: 指向当前用于保存请求体的内存缓存；

rest: 当前剩余的请求体大小；

post_handler: 保存传给 `ngx_http_read_client_request_body()` 函数的回调函数。

做好准备工作之后，函数开始检查请求是否带有 `content_length` 头，如果没有该头或者客户端发送了一个值为 0 的 `content_length` 头，表明没有请求体，这时直接调用回调函数并返回 `NGX_OK` 即可。当然如果 `client_body_in_file_only` 指令被设置为 `on`，且 `content_length` 为 0 时，该函数在调用回调函数之前，会创建一个空的临时文件。

进入到函数下半部分，表明客户端请求确实表明了要发送请求体，该函数会先检查是否在读取请求头时预读了请求体，这里的检查是通过判断保存请求头的缓存(`r->header_in`)中是否还有未处理的数据。

如果有预读数据，则分配一个 `ngx_buf_t` 结构，并将 `r->header_in` 中的预读数据保存在其中，并且如果 `r->header_in` 中还有剩余空间，并且能够容下剩余未读取的请求体，这些空间将被继续使用，而不用分配新的缓存，当然甚至如果请求体已经被整个预读了，则不需要继续处理了，此时调用回调函数后返回。

如果没有预读数据或者预读不完整，该函数会分配一块新的内存（除非 `r->header_in` 还有足够的剩余空间），另外如果 `request_body_in_single_buf` 指令被设置为 `no`，则预读的数据会被拷贝进新开辟的内存块中，真正读取请求体的操作是在 `ngx_http_do_read_client_request_body()` 函数，该函数循环的读取请求体并保存在缓存中，如果缓存被写满了，其中的数据会被清空并写回到临时文件中。当然这里有可能不能一次将数据读到，该函数会挂载读事件并设置读事件 handler 为

`ngx_http_read_client_request_body_handler`，另外 `nginx` 核心对两次请求体的读事件之间也做了超时

设置, `client_body_timeout` 指令可以设置这个超时时间, 默认为 60 秒, 如果下次读事件超时了, `nginx` 会返回 408 给客户端。

最终读完请求体后, `ngx_http_do_read_client_request_body()`会根据配置, 将请求体调整到预期的位置 (内存或者文件), 所有情况下请求体都可以从 `r->request_body` 的 `bufs` 链表得到, 该链表最多可能有 2 个节点, 每个节点为一个 `buffer`, 但是这个 `buffer` 的内容可能是保存在内存中, 也可能是保存在磁盘文件中。另外 `$request_body` 变量只在当请求体已经被读取并且是全部保存在内存中, 才能取得相应的数据。

丢弃请求体

一个模块想要主动的丢弃客户端发过的请求体, 可以调用 `nginx` 核心提供的

`ngx_http_discard_request_body()`接口, 主动丢弃的原因可能有很多种, 如模块的业务逻辑压根不需要请求体, 客户端发送了过大的请求体, 另外为了兼容 `http1.1` 协议的 `pipeline` 请求, 模块有义务主动丢弃不需要的请求体。总之为了保持良好的客户端兼容性, `nginx` 必须主动丢弃无用的请求体。下面开始分析 `ngx_http_discard_request_body()`函数:

```
ngx_int_t
ngx_http_discard_request_body(ngx_http_request_t *r)
{
    ssize_t      size;
    ngx_event_t  *rev;

    if (r != r->main || r->discard_body) {
        return NGX_OK;
    }

    if (ngx_http_test_expect(r) != NGX_OK) {
        return NGX_HTTP_INTERNAL_SERVER_ERROR;
    }

    rev = r->connection->read;

    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, rev->log, 0, "http set discard body");

    if (rev->timer_set) {
        ngx_del_timer(rev);
    }
}
```

```

if (r->headers_in.content_length_n <= 0 || r->request_body) {
    return NGX_OK;
}

size = r->header_in->last - r->header_in->pos;

if (size) {
    if (r->headers_in.content_length_n > size) {
        r->header_in->pos += size;
        r->headers_in.content_length_n -= size;

    } else {
        r->header_in->pos += (size_t) r->headers_in.content_length_n;
        r->headers_in.content_length_n = 0;
        return NGX_OK;
    }
}

r->read_event_handler = ngx_http_discarded_request_body_handler;

if (ngx_handle_read_event(rev, 0) != NGX_OK) {
    return NGX_HTTP_INTERNAL_SERVER_ERROR;
}

if (ngx_http_read_discarded_request_body(r) == NGX_OK) {
    r->lingering_close = 0;

} else {
    r->count++;
    r->discard_body = 1;
}

return NGX_OK;
}

```

由于函数不长，这里把它完整的列出来了，函数的开始同样先判断了不需要再做处理的情况：子请求不需要处理，已经调用过此函数的也不需要再处理。接着调用 `ngx_http_test_expect()` 处理 http1.1 expect 的情况，根据 http1.1 的 expect 机制，如果客户端发送了 expect 头，而服务端不希望接收请求体时，必须返回 417(Expectation Failed)错误。nginx 并没有这样做，它只是简单的让客户端把请求体发送过来，然后丢弃掉。接下来，函数删掉了读事件上的定时器，因为这时本身就不需要请求体，所以也无所谓客户端发送的快还是慢了，当然后面还会讲到，当 nginx 已经处理完该请求但客户端还没有发送完无用的请求体时，nginx 会在读事件上再挂上定时器。

客户端如果打算发送请求体，就必须发送 content-length 头，所以函数会检查请求头中的 content-length 头，同时还会查看其他地方是不是已经读取了请求体。如果确实有待处理的请求体，函数接着检查请求头 buffer 中预读的数据，预读的数据会直接被丢掉，当然如果请求体已经被全部预读，函数就直接返回了。

接下来，如果还有剩余的请求体未处理，该函数调用 ngx_handle_read_event() 在事件处理机制中挂载好读事件，并把读事件的处理函数设置为 ngx_http_discarded_request_body_handler。做好这些准备之后，该函数最后调用 ngx_http_read_discarded_request_body() 接口读取客户端过来的请求体并丢弃。如果客户端并没有一次将请求体发过来，函数会返回，剩余的数据等到下一次读事件过来时，交给 ngx_http_discarded_request_body_handler() 来处理，这时，请求的 discard_body 将被设置为 1 用来标识这种情况。另外请求的引用数(count)也被加 1，这样做的目的是客户端可能在 nginx 处理完请求之后仍未完整发送待发送的请求体，增加引用是防止 nginx 核心在处理完请求后直接释放了请求的相关资源。

ngx_http_read_discarded_request_body() 函数非常简单，它循环的从链接中读取数据并丢弃，直到读完接收缓冲区的所有数据，如果请求体已经被读完了，该函数会设置读事件的处理函数为 ngx_http_block_reading，这个函数仅仅删除水平触发的读事件，防止同一事件不断被触发。

最后看一下读事件的处理函数 ngx_http_discarded_request_body_handler，这个函数每次读事件来时会被调用，先看一下它的源码：

```
void
ngx_http_discarded_request_body_handler(ngx_http_request_t *r)
{
    ...

    c = r->connection;
    rev = c->read;

    if (rev->timedout) {
        c->timedout = 1;
        c->error = 1;
        ngx_http_finalize_request(r, NGX_ERROR);
        return;
    }

    if (r->lingering_time) {
        timer = (ngx_msec_t) (r->lingering_time - ngx_time());
```

```

        if (timer <= 0) {
            r->discard_body = 0;
            r->lingering_close = 0;
            ngx_http_finalize_request(r, NGX_ERROR);
            return;
        }

    } else {
        timer = 0;
    }

    rc = ngx_http_read_discarded_request_body(r);

    if (rc == NGX_OK) {
        r->discard_body = 0;
        r->lingering_close = 0;
        ngx_http_finalize_request(r, NGX_DONE);
        return;
    }

    /* rc == NGX_AGAIN */

    if (ngx_handle_read_event(rev, 0) != NGX_OK) {
        c->error = 1;
        ngx_http_finalize_request(r, NGX_ERROR);
        return;
    }

    if (timer) {

        clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

        timer *= 1000;

        if (timer > clcf->lingering_timeout) {
            timer = clcf->lingering_timeout;
        }

        ngx_add_timer(rev, timer);
    }
}

```

函数一开始就处理了读事件超时的情况，之前说到在 `ngx_http_discard_request_body()` 函数中已经删除了读事件的定时器，那么什么时候会设置定时器呢？答案就是在 `nginx` 已经处理完该请求，但是又没有完全将该请求的请求体丢弃的时候（客户端可能还没有发送过来），在 `ngx_http_finalize_connection()`

函数中，如果检查到还有未丢弃的请求体时，nginx 会添加一个读事件定时器，它的时长为 `lingering_timeout` 指令所指定，默认为 5 秒，不过这个时间仅仅两次读事件之间的超时时间，等待请求体的总时长为 `lingering_time` 指令所指定，默认为 30 秒。这种情况中，该函数如果检测到超时事件则直接返回并断开连接。同样，还需要控制整个丢弃请求体的时长不能超过 `lingering_time` 设置的时间，如果超过了最大时长，也会直接返回并断开连接。

如果读事件发生在请求处理完之前，则不用处理超时事件，也不用设置定时器，函数只是简单的调用 `ngx_http_read_discarded_request_body()` 来读取并丢弃数据。

多阶段处理请求

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

nginx 实际把请求处理流程划分为了 11 个阶段，这样划分的原因是将请求的执行逻辑细分，各阶段按照处理时机定义了清晰的执行语义，开发者可以很容易分辨自己需要开发的模块应该定义在什么阶段，下面介绍一下各阶段：

NGX_HTTP_POST_READ_PHASE:

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

NGX_HTTP_SERVER_REWRITE_PHASE:

server 级别的 uri 重写阶段，也就是该阶段执行处于 server 块内，location 块外的重写指令，前面的章节已经说明在读取请求头的过程中 nginx 会根据 host 及端口找到对应的虚拟主机配置；

NGX_HTTP_FIND_CONFIG_PHASE:

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

NGX_HTTP_REWRITE_PHASE:

location 级别的 uri 重写阶段，该阶段执行 location 基本的重写指令，也可能被执行多次；

NGX_HTTP_POST_REWRITE_PHASE:

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

NGX_HTTP_PREACCESS_PHASE:

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

NGX_HTTP_ACCESS_PHASE:

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

NGX_HTTP_POST_ACCESS_PHASE:

访问权限控制的最后一阶段，该阶段根据权限控制阶段的执行结果进行相应处理；

NGX_HTTP_TRY_FILES_PHASE:

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

NGX_HTTP_CONTENT_PHASE:

内容生成阶段，该阶段产生响应，并发送到客户端；

NGX_HTTP_LOG_PHASE:

日志记录阶段，该阶段记录访问日志。

多阶段执行链

nginx 按请求处理的执行顺序将处理流程划分为多个阶段，一般每个阶段又可以注册多个模块处理函数，

nginx 按阶段将这些处理函数组织成了一个执行链，这个执行链保存在 http 主配置

(ngx_http_core_main_conf_t) 的 phase_engine 字段中，phase_engine 字段的类型为

ngx_http_phase_engine_t:

```
typedef struct {
    ngx_http_phase_handler_t  *handlers;
    ngx_uint_t                 server_rewrite_index;
    ngx_uint_t                 location_rewrite_index;
} ngx_http_phase_engine_t;
```

其中 handlers 字段即为执行链，实际上它是一个数组，而每个元素之间又被串成链表，从而允许执行流程向前，或者向后的阶段跳转，执行链节点的数据结构定义如下：

```
struct ngx_http_phase_handler_s {
    ngx_http_phase_handler_pt  checker;
    ngx_http_handler_pt        handler;
    ngx_uint_t                 next;
};
```

其中 checker 和 handler 都是函数指针，相同阶段的节点具有相同的 checker 函数，handler 字段保存

的是模块处理函数，一般在 checker 函数中会执行当前节点的 handler 函数，但是例外的是

NGX_HTTP_FIND_CONFIG_PHASE, NGX_HTTP_POST_REWRITE_PHASE,

NGX_HTTP_POST_ACCESS_PHASE 和 NGX_HTTP_TRY_FILES_PHASE 这 4 个阶段不能注册模块函数。next 字段为快速跳跃索引，多数情况下，执行流程是按照执行链顺序的往前执行，但在某些执行阶段的 checker 函数中由于执行了某个逻辑可能需要回跳至之前的执行阶段，也可能需要跳过之后的某些执行阶段，next 字段保存的就是跳跃的目的索引。

和建立执行链相关的数据结构都保存在 http 主配置中，一个是 phases 字段，另外一个 phase_engine 字段。其中 phases 字段为一个数组，它的元素个数等于阶段数目，即每个元素对应一个阶段。而 phases 数组的每个元素又是动态数组（ngx_array_t），每次模块注册处理函数时只需要在对应阶段的动态数组增加一个元素用来保存处理函数的指针。由于在某些执行阶段可能需要向后，或者向前跳转，简单的使用 2 个数组并不方便，所以 nginx 又组织了一个执行链，保存在了 phase_engine 字段，其每个节点包含一个 next 域用来保存跳跃目的节点的索引，而执行链的建立则在 nginx 初始化的 post config 阶段之后调用 ngx_http_init_phase_handlers 函数完成，下面分析一下该函数：

```
static ngx_int_t
ngx_http_init_phase_handlers(ngx_conf_t *cf, ngx_http_core_main_conf_t *cmcf)
{
    ngx_int_t          j;
    ngx_uint_t          i, n;
    ngx_uint_t          find_config_index, use_rewrite, use_access;
    ngx_http_handler_pt *h;
    ngx_http_phase_handler_t *ph;
    ngx_http_phase_handler_pt checker;

    cmcf->phase_engine.server_rewrite_index = (ngx_uint_t) -1;
    cmcf->phase_engine.location_rewrite_index = (ngx_uint_t) -1;
    find_config_index = 0;
    use_rewrite = cmcf->phases[NGX_HTTP_REWRITE_PHASE].handlers.nelts ? 1 : 0;
    use_access = cmcf->phases[NGX_HTTP_ACCESS_PHASE].handlers.nelts ? 1 : 0;

    n = use_rewrite + use_access + cmcf->try_files + 1 /* find config phase */;

    for (i = 0; i < NGX_HTTP_LOG_PHASE; i++) {
        n += cmcf->phases[i].handlers.nelts;
    }

    ph = ngx_palloc(cf->pool,
                    n * sizeof(ngx_http_phase_handler_t) + sizeof(void *));
    if (ph == NULL) {
        return NGX_ERROR;
    }
}
```

```

cmcf->phase_engine.handlers = ph;
n = 0;

for (i = 0; i < NGX_HTTP_LOG_PHASE; i++) {
    h = cmcf->phases[i].handlers.elts;

    switch (i) {

case NGX_HTTP_SERVER_REWRITE_PHASE:
    if (cmcf->phase_engine.server_rewrite_index == (ngx_uint_t) -1) {
        cmcf->phase_engine.server_rewrite_index = n;
    }
    checker = ngx_http_core_rewrite_phase;

    break;

case NGX_HTTP_FIND_CONFIG_PHASE:
    find_config_index = n;

    ph->checker = ngx_http_core_find_config_phase;
    n++;
    ph++;

    continue;

case NGX_HTTP_REWRITE_PHASE:
    if (cmcf->phase_engine.location_rewrite_index == (ngx_uint_t) -1) {
        cmcf->phase_engine.location_rewrite_index = n;
    }
    checker = ngx_http_core_rewrite_phase;

    break;

case NGX_HTTP_POST_REWRITE_PHASE:
    if (use_rewrite) {
        ph->checker = ngx_http_core_post_rewrite_phase;
        ph->next = find_config_index;
        n++;
        ph++;
    }

    continue;

case NGX_HTTP_ACCESS_PHASE:
    checker = ngx_http_core_access_phase;
    n++;
    break;

```



```

case NGX_HTTP_POST_ACCESS_PHASE:
    if (use_access) {
        ph->checker = ngx_http_core_post_access_phase;
        ph->next = n;
        ph++;
    }

    continue;

case NGX_HTTP_TRY_FILES_PHASE:
    if (cmcf->try_files) {
        ph->checker = ngx_http_core_try_files_phase;
        n++;
        ph++;
    }

    continue;

case NGX_HTTP_CONTENT_PHASE:
    checker = ngx_http_core_content_phase;
    break;

default:
    checker = ngx_http_core_generic_phase;
}

n += cmcf->phases[i].handlers.nelts;

for (j = cmcf->phases[i].handlers.nelts - 1; j >=0; j--) {
    ph->checker = checker;
    ph->handler = h[j];
    ph->next = n;
    ph++;
}

return NGX_OK;
}

```

首先需要说明的是 `cmcf->phases` 数组中保存了在 `post config` 之前注册的所有模块函数，上面的函数先计算执行链的节点个数，并分配相应的空间，前面提到有 4 个阶段不能注册模块，并且 `POST_REWRITE` 和 `POST_ACCESS` 这 2 个阶段分别只有在 `REWRITE` 和 `ACCESS` 阶段注册了模块时才存在，另外 `TRY_FILES`

阶段只有在配置了 `try_files` 指令的时候才存在，最后 `FIND_CONFIG` 阶段虽然不能注册模块，但它是必须存在的，所以在计算执行链节点数时需要考虑这些因素。

分配好内存之后，开始建立链表，过程很简单，遍历每个阶段注册的模块函数，为每个阶段的节点赋值 `checker` 函数，`handler` 函数，以及 `next` 索引。最终建立好的执行链如下图：

(暂缺)

`SERVER_REWRITE` 阶段的节点的 `next` 域指向 `FIND_CONFIG` 阶段的第 1 个节点，`REWRITE` 阶段的 `next` 域指向 `POST_REWRITE` 阶段的第 1 个节点，而 `POST_REWRITE` 阶段的 `next` 域则指向 `FIND_CONFIG`，因为当出现 `location` 级别的 `uri` 重写时，可能需要重新匹配新的 `location`，`PREACCESS` 阶段的 `next` 域指向 `ACCESS` 域，`ACCESS` 和 `POST_ACCESS` 阶段的 `next` 域则是则是指向 `CONTENT` 阶段，当然如果 `TRY_FILES` 阶段存在的话，则是指向 `TRY_FILES` 阶段，最后 `CONTENT` 阶段的 `next` 域指向 `LOG` 阶段，当然 `next` 域是每个阶段的 `checker` 函数根据该阶段的需求来使用的，没有需要时，`checker` 函数可能都不会使用到它。

POST_READ 阶段

`POST_READ` 阶段是 `nginx` 处理请求流程中第一个可以添加模块函数的阶段，任何需要在接收完请求头之后立刻处理的逻辑可以在该阶段注册处理函数。`nginx` 源码中只有 `realip` 模块在该阶段注册了函数，当 `nginx` 前端多了一个 7 层负载均衡层，并且客户端的真实 `ip` 被前端保存在请求头中时，该模块用来将客户端的 `ip` 替换为请求头中保存的值。`realip` 模块之所以在 `POST_READ` 阶段执行的原因是它需要在其他模块执行之前悄悄的将客户端 `ip` 替换为真实值，而且它需要的信息仅仅只是请求头。一般很少有模块需要注册在 `POST_READ` 阶段，`realip` 模块默认没有编译进 `nginx`。

`POST_READ` 阶段的 `checker` 函数是 `ngx_http_core_generic_phase`，这个函数是 `nginx` phase 默认的 `checker` 函数，后面的 `PREACCESS` phase 也是用 `checker`，下面对它做一下介绍：

```
ngx_int_t
ngx_http_core_generic_phase(ngx_http_request_t *r, ngx_http_phase_handler_t *ph)
{
    ngx_int_t rc;
```

```

/*
 * generic phase checker,
 * used by the post read and pre-access phases
 */

ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
               "generic phase: %ui", r->phase_handler);

rc = ph->handler(r);

if (rc == NGX_OK) {
    r->phase_handler = ph->next;
    return NGX_AGAIN;
}

if (rc == NGX_DECLINED) {
    r->phase_handler++;
    return NGX_AGAIN;
}

if (rc == NGX_AGAIN || rc == NGX_DONE) {
    return NGX_OK;
}

/* rc == NGX_ERROR || rc == NGX_HTTP_... */

ngx_http_finalize_request(r, rc);

return NGX_OK;
}

```

这个函数逻辑非常简单，调用该 phase 注册的 handler 函数，需要注意的是该函数对 handler 返回值的处理，一般而言 handler 返回：

NGX_OK: 表示该阶段已经处理完成，需要转入下一个阶段；

NG_DECLINED: 表示需要转入本阶段的下一个 handler 继续处理；

NGX_AGAIN, NGX_DONE:

表示需要等待某个事件发生才能继续处理（比如等待网络 IO），此时 Nginx 为了不阻塞其他请求的处理，必须中断当前请求的执行链，等待事件发生之后继续执行该 handler；

NGX_ERROR: 表示发生了错误，需要结束该请求。

checker 函数根据 handler 函数的不同返回值，给上一层的 ngx_http_core_run_phases 函数返回

NGX_AGAIN 或者 NGX_OK，如果期望上一层继续执行后面的 phase 则需要确保 checker 函数不是返回

NGX_OK，不同 checker 函数对 handler 函数的返回值处理还不太一样，开发模块时需要确保相应阶段的 checker 函数对返回值的处理在你的预期之内。

SERVER_REWRITE 阶段

SERVER_REWRITE 阶段是 nginx 中第一个必须经历的重要 phase，请求进入此阶段时已经找到对应的虚拟主机（server）配置。nginx 的 rewrite 模块在这个阶段注册了一个 handler，rewrite 模块提供 url 重写指令 rewrite，变量设置指令 set，以及逻辑控制指令 if、break 和 return，用户可以在 server 配置里面，组合这些指令来满足自己的需求，而不需要另外写一个模块，比如将一些前缀满足特定模式的 uri 重定向到一个固定的 url，还可以根据请求的属性来决定是否需要重写或者给用户发送特定的返回码。rewrite 提供的逻辑控制指令能够满足一些简单的需求，针对一些较复杂的逻辑可能需要注册 handler 通过独立实现模块的方式来满足。

需要注意该阶段和后面的 REWRITE 阶段的区别，在 SERVER_REWRITE 阶段中，请求还未被匹配到一个具体的 location 中。该阶段执行的结果（比如改写后的 uri）会影响后面 FIND_CONFIG 阶段的执行。

另外这个阶段也是内部子请求执行的第一个阶段。 SERVER_REWRITE 阶段的 checker 函数是

ngx_http_core_rewrite_phase:

```
ngx_int_t
ngx_http_core_rewrite_phase(ngx_http_request_t *r, ngx_http_phase_handler_t *ph)
{
    ngx_int_t rc;

    ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
        "rewrite phase: %ui", r->phase_handler);

    rc = ph->handler(r);

    if (rc == NGX_DECLINED) {
        r->phase_handler++;
        return NGX_AGAIN;
    }

    if (rc == NGX_DONE) {
        return NGX_OK;
    }
}
```

```
/* NGX_OK, NGX_AGAIN, NGX_ERROR, NGX_HTTP_... */  
  
ngx_http_finalize_request(r, rc);  
  
return NGX_OK;  
}
```

这个函数和上面说的 `ngx_http_core_generic_phase` 函数流程基本一致，唯一的区别就是对 handler 返回值的处理稍有不同，比如这里对 `NGX_OK` 的处理是调用 `ngx_http_finalize_request` 结束请求，所以再强调一下，handler 函数的返回值一定要根据不同 phase 的 checker 函数来设置。Nginx 的 rewrite 模块会挂上一个名为 `ngx_http_rewrite_handler` 的 handler。

FIND_CONFIG 阶段

FIND_CONFIG 阶段顾名思义就是寻找配置阶段，具体一点就是根据 uri 查找 location 配置，实际上就是设置 `r->loc_conf`，在此之前 `r->loc_conf` 使用的 server 级别的，查找 location 过程由函数 `ngx_http_core_find_location` 完成，具体查找流程这里不再赘述，可以参考上一章关于 location 管理的内容，值得注意的是当 `ngx_http_core_find_location` 函数返回 `NGX_DONE` 时，Nginx 会返回 301，将用户请求做一个重定向，这种情况仅发生在该 location 使用了 `proxy_pass/fastcgi/scgi/uwsgi/memcached` 模块，且 location 的名字以/符号结尾，并且请求的 uri 为该 location 除/之外的前缀，比如对 location `/xx/`，如果某个请求 `/xx` 访问到该 location，则会被重定向为 `/xx/`。另外 Nginx 中 location 可以标识为 `internal`，即内部 location，这种 location 只能由子请求或者内部跳转访问。

找到 location 配置后，Nginx 调用了 `ngx_http_update_location_config` 函数来更新请求相关配置，其中最重要的是更新请求的 content handler，不同 location 可以有自己的 content handler。

最后，由于有 `REWRITE_PHASE` 的存在，FIND_CONFIG 阶段可能会被执行多次。

REWRITE 阶段

REWRITE 阶段为 location 级别的重写, 这个阶段的 checker 和 SERVER_REWRITE 阶段的是同一个函数, 而且 Nginx 的 rewrite 模块对这 2 个阶段注册的是同一个 handler, 2 者唯一区别就是执行时机不一样, REWRITE 阶段为 location 级别的重写, SERVER_REWRITE 执行之后是 FIND_CONFIG 阶段, REWRITE 阶段执行之后是 POST_REWRITE 阶段。

POST_REWRITE 阶段

该阶段不能注册 handler, 仅仅只是检查上一阶段是否做了 uri 重写, 如果没有重写的话, 直接进入下一阶段; 如果有重写的话, 则利用 next 跳转域往前跳转到 FIND_CONFIG 阶段重新执行。Nginx 对 uri 重写次数做了限制, 默认是 10 次。

PREACCESS 阶段

进入该阶段表明 Nginx 已经将请求确定到了某一个 location(当该 server 没有任何 location 时, 也可能是 server), 无论如何请求的 loc_conf 配置已经确定下来, 该阶段一般用来做资源控制, 默认情况下, 诸如 ngx_http_limit_conn_module, ngx_http_limit_req_module 等模块会在该阶段注册 handler, 用于控制连接数, 请求速率等。PREACCESS 阶段使用的 checker 是默认的 ngx_http_core_generic_phase 函数。

ACCESS 阶段

该阶段的首要目的是做权限控制, 默认情况下, Nginx 的 ngx_http_access_module 和 ngx_http_auth_basic_module 模块分别会在该阶段注册一个 handler。

ACCESS 阶段的 checker 是 ngx_http_core_access_phase 函数, 此函数对 handler 返回值的处理大致和 ngx_http_core_generic_phase 一致, 特殊的地方是当 clcf->satisfy 为 NGX_HTTP_SATISFY_ALL, 也就是需要满足该阶段注册的所有 handler 的验证时, 某个 handler 返回 NGX_OK 时还需要继续处理本阶段的其他 handler。clcf->satisfy 的值可以使用 satisfy 指令指定。

POST_ACCESS 阶段

POST_ACCESS 和 POST_REWRITE 阶段一样，只是处理一下上一阶段的结果，而不能挂载自己的 handler，具体为如果 ACCESS 阶段返回了 NGX_HTTP_FORBIDDEN 或 NGX_HTTP_UNAUTHORIZED（记录在 r->access_code 字段），该阶段会结束掉请求。

TRY_FILES 阶段

TRY_FILES 阶段仅当配置了 try_files 指令时生效，实际上该指令不常用，它的功能是指定一个或者多个文件或目录，最后一个参数可以指定为一个 location 或一个返回码，当设置了该指令时，TRY_FILES 阶段调用 checker 函数 ngx_http_core_try_files_phase 来依此检查指定的文件或目录是否存在，如果本地文件系统存在某个文件或目录则退出该阶段继续执行下面的阶段，否则内部重定向到最后一个参数指定的 location 或返回指定的返回码。

该阶段也不能注册 handler。

CONTENT 阶段

CONTENT 阶段可以说是整个执行链中最重要的阶段，请求从这里开始执行业务逻辑并产生响应，下面来分析一下它的 checker 函数：

```
ngx_int_t
ngx_http_core_content_phase(ngx_http_request_t *r,
    ngx_http_phase_handler_t *ph)
{
    size_t    root;
    ngx_int_t rc;
    ngx_str_t path;

    if (r->content_handler) {
        r->write_event_handler = ngx_http_request_empty_handler;
        ngx_http_finalize_request(r, r->content_handler(r));
        return NGX_OK;
    }

    ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
        "content phase: %ui", r->phase_handler);

    rc = ph->handler(r);

    if (rc != NGX_DECLINED) {
```

```

    ngx_http_finalize_request(r, rc);
    return NGX_OK;
}

/* rc == NGX_DECLINED */

ph++;

if (ph->checker) {
    r->phase_handler++;
    return NGX_AGAIN;
}

/* no content handler was found */

if (r->uri.data[r->uri.len - 1] == '/') {

    if (ngx_http_map_uri_to_path(r, &path, &root, 0) != NULL) {
        ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
            "directory index of \"%s\" is forbidden", path.data);
    }

    ngx_http_finalize_request(r, NGX_HTTP_FORBIDDEN);
    return NGX_OK;
}

ngx_log_error(NGX_LOG_ERR, r->connection->log, 0, "no handler found");

ngx_http_finalize_request(r, NGX_HTTP_NOT_FOUND);
return NGX_OK;
}

```

CONTENT 阶段有些特殊，它不像其他阶段只能执行固定的 handler 链，还有一个特殊的 content_handler，每个 location 可以有自己独立的 content handler，而且当有 content handler 时，CONTENT 阶段只会执行 content handler，不再执行本阶段的 handler 链。

默认情况下，Nginx 会在 CONTENT 阶段的 handler 链挂上 index 模块，静态文件处理模块等的 handler。另外模块还可以设置独立的 content handler，比如 ngx_http_proxy_module 的 proxy_pass 指令会设置一个名为 ngx_http_proxy_handler 的 content handler。

接下来看一下上面的 checker 函数的执行流程，首先检查是否设置了 r->content_handler，如果设置的话，则执行它，需要注意的是在执行它之前，Nginx 将 r->write_event_handler 设置为了

ngx_http_request_empty_handler，先看一下设置 r->write_event_handler 之前的值是什么，在 ngx_http_handler 函数中它被设置为 ngx_http_core_run_phases，而 ngx_http_core_run_phases 会运行每个阶段的 checker 函数。正常流程中，如果某个阶段需要等待某个写事件发生时，该阶段的 handler 会返回 NGX_OK 来中断 ngx_http_core_run_phases 的运行，等到下次写事件过来时，会继续执行之前阶段的 handler；当执行 r->content_handler 的流程时，Nginx 默认模块会去处理 r->write_event_handler 的值，也就是假设 r->content_handler 只能执行 1 次，如果模块设置的 content handler 涉及到 IO 操作，就需要合理的设置处理读写事件的 handler（r->read_event_handler 和 r->write_event_handler）。

还有一个需要注意的点是 r->content_handler 执行之后，Nginx 直接用其返回值调用了 ngx_http_finalize_request 函数，Nginx 将一大堆耦合的逻辑都集中在了这个函数当中，包括长连接，lingering_close，子请求等的处理都涉及到该函数，后面会有一节单独介绍这个函数。这里需要提醒的是 r->content_handler 如果并未完成整个请求的处理，而只是需要等待某个事件发生而退出处理流程的话，必须返回一个合适的值传给 ngx_http_finalize_request，一般而言是返回 NGX_DONE，而且需要将请求的引用计数（r->count）加 1，确保 ngx_http_finalize_request 函数不会将该请求释放掉。

函数的其他部分处理走 handler 链的情况，特殊的地方是 CONTENT 阶段是 ngx_http_core_run_phases 函数跑的最后一个阶段，如果最后一个 handler 返回 NGX_DECLINED，此时 Nginx 会给客户端返回 NGX_HTTP_FORBIDDEN（403）或 NGX_HTTP_NOT_FOUND（404）。

LOG 阶段

LOG 阶段主要的目的就是记录访问日志，进入该阶段表明该请求的响应已经发送到系统发送缓冲区。另外这个阶段的 handler 链实际上并不是在 ngx_http_core_run_phases 函数中执行，而是在释放请求资源的 ngx_http_free_request 函数中运行，这样做的原因实际是为了简化流程，因为 ngx_http_core_run_phases 可能会执行多次，而 LOG 阶段只需要再请求所有逻辑都结束时运行一次，所以在 ngx_http_free_request 函数中运行 LOG 阶段的 handler 链是非常好的选择。具体的执行的函数为 ngx_http_log_request：

```
static void
ngx_http_log_request(ngx_http_request_t *r)
{
    ngx_uint_t          i, n;
    ngx_http_handler_pt *log_handler;
    ngx_http_core_main_conf_t *cmcf;

    cmcf = ngx_http_get_module_main_conf(r, ngx_http_core_module);

    log_handler = cmcf->phases[NGX_HTTP_LOG_PHASE].handlers.elts;
    n = cmcf->phases[NGX_HTTP_LOG_PHASE].handlers.nelts;

    for (i = 0; i < n; i++) {
        log_handler[i](r);
    }
}
```

函数非常简单，仅仅是遍历 LOG 阶段的 handler 链，逐一执行，而且不会检查返回值。LOG 阶段和其他阶段的不同点有两个，一是执行点是在 ngx_http_free_request 中，二是这个阶段的所有 handler 都会被执行。

至此，Nginx 请求处理的多阶段执行链的各个阶段都已经介绍完毕，弄清楚每个阶段的执行时机以及每个阶段的不同特点对写模块非常重要。

Nginx filter

在 CONTENT 阶段产生的数据被发往客户端（系统发送缓存区）之前，会先经过过滤。Nginx 的 filter 的工作方式和做鱼有些类似。比如一条鱼，可以把它切成鱼片（也可以切块，切泥），然后通过不同的烹饪方法就得到水煮鱼或者日式生鱼片或者废了等等。同样是一条鱼，加工得到的结果却截然不同，就是因为中间不同的工序赋予了这条鱼各种属性。Nginx 的 filter 也是一个道理，前面的 Handler 好比这条鱼，filter 负责加工，最后得到的 HTTP 响应就会各种各样，格式可以是 JSON 或者 YAML，内容可能多一些或者少一些，HTTP 属性可各异，可以选择压缩，甚至内容可以被丢弃。

对应 HTTP 请求的响应头和响应体，Nginx 分别设置了 header filter 和 body filter。两种机制都是采用链表的方式，不同过滤模块对应链表的一个节点，一般而言一个模块会同时注册 header filter 和 body filter。一个典型的 filter 模块，比如 gzip 模块使用类似如下的代码来注册：

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

```

...

static ngx_int_t
ngx_http_gzip_filter_init(ngx_conf_t *cf)
{
    ngx_http_next_header_filter = ngx_http_top_header_filter;
    ngx_http_top_header_filter = ngx_http_gzip_header_filter;

    ngx_http_next_body_filter = ngx_http_top_body_filter;
    ngx_http_top_body_filter = ngx_http_gzip_body_filter;

    return NGX_OK;
}

```

上面的代码中，gzip 模块首先在模块的开头声明了两个 static 类型的全局变量

ngx_http_next_header_filter 和 ngx_http_next_body_filter，在 ngx_http_gzip_filter_init 函数中，这二个变量分别被赋值为 ngx_http_top_header_filter 及 ngx_http_top_body_filter。而后二者定义在 ngx_http.c，并在 ngx_http.h 头文件中被导出。ngx_http_top_header_filter 和 ngx_http_top_body_filter 实际上是 filter 链表的头结点，每次注册一个新的 filter 模块时，它们的值先被保存在新模块的内部全局变量 ngx_http_next_header_filter 及 ngx_http_next_body_filter，然后被赋值为新模块注册的 filter 函数，而且 Nginx filter 是先从头节点开始执行，所以越晚注册的模块越早执行。

采用默认编译选项，Nginx 默认编译的模块如下：

```

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_regex_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_conn_module,
    &ngx_http_limit_req_module,
    &ngx_http_geo_module,
    &ngx_http_map_module,
    &ngx_http_split_clients_module,
    &ngx_http_referer_module,
    &ngx_http_rewrite_module,
    &ngx_http_proxy_module,
    &ngx_http_fastcgi_module,
    &ngx_http_uwsgi_module,
    &ngx_http_scgi_module,
    &ngx_http_memcached_module,
    &ngx_http_empty_gif_module,
    &ngx_http_browser_module,
    &ngx_http_upstream_ip_hash_module,
    &ngx_http_upstream_keepalive_module,
    &ngx_http_write_filter_module,
据 */
    &ngx_http_header_filter_module,
拼接出完整的 http 响应头,

    &ngx_http_chunked_filter_module,
求, 强制短连接 (低于 http 1.1)

    &ngx_http_range_header_filter_module,
    &ngx_http_gzip_filter_module,
    &ngx_http_postpone_filter_module,
据的输出顺序 */
    &ngx_http_ssi_filter_module,
式, 去获取 include 进来的文件 */
    &ngx_http_charset_filter_module,
符集转换到另外一种字符集 */
    &ngx_http_userid_filter_module,
    &ngx_http_headers_filter_module,
支持添加任意名称的头 */
    &ngx_http_copy_filter_module,

新的节点), 并交给后续 filter

    &ngx_http_range_body_filter_module,
求包含 range 请求,

    &ngx_http_not_modified_filter_module,
复的 last-modified 值,

回 304 */
    NULL

```

/* 最后一个 body filter, 负责往外发送数

/* 最后一个 header filter, 负责在内存中

并调用 ngx_http_write_filter 发送 */

/* 对响应头中没有 content_length 头的请

或采用 chunked 编码 (http 1.1) */

/* header filter, 负责处理 range 头 */

/* 支持流式的数据压缩 */

/* body filter, 负责处理子请求和主请求数

/* 支持过滤 SSI 请求, 采用发起子请求的方

/* 支持添加 charset, 也支持将内容从一种字

/* 支持添加统计用的识别用户的 cookie */

/* 支持设置 expire 和 Cache-control 头,

/* 根据需求重新复制输出链表中的某些节点

(比如将 in_file 的节点从文件读出并复制到

进行处理 */

/* body filter, 支持 range 功能, 如果请

那就只发送 range 请求的一段内容 */

/* 如果请求的 if-modified-since 等于回

说明回复没有变化, 清空所有回复的内容, 返

```
};
```

从模块的命名可以很容易看出哪些模块是 filter 模块，一般而言 Nginx 的 filter 模块名以 filter_module 结尾，普通的模块名以 module 结尾。上面的列表从下往上看，ngx_http_not_modified_filter_module 实际上 filter 链的第一个节点，而 ngx_http_write_filter_module 是最后一个节点。filter 模块的执行顺序特别重要，比如数据经过 gzip 模块后就变成了压缩之后的数据，如果在 gzip 模块后面运行的 filter 模块需要再查看数据的原始内容就不可能了（除非再做解压），第三方模块会被 Nginx 注册在 ngx_http_copy_filter_module 之后，ngx_http_headers_filter_module 之前。这样设定的原因是为了确保一些模块比如 gzip filter，chunked filter，copy filter 运行在 filter 链的开头或尾部。

header filter 分析

通常 Nginx 调用 ngx_http_send_header 函数来发送响应头，看下它的实现：

```
ngx_int_t
ngx_http_send_header(ngx_http_request_t *r)
{
    if (r->err_status) {
        r->headers_out.status = r->err_status;
        r->headers_out.status_line.len = 0;
    }

    return ngx_http_top_header_filter(r);
}
```

上面的代码中调用了 ngx_http_top_header_filter，也就是 header filter 的头节点，按照上一节介绍的顺序，ngx_http_not_modified_filter_module 是最后一个注册的 filter 模块，而最后定义的会最先执行，初始化之后，它实际上是 ngx_http_not_modified_header_filter 函数：

```
static ngx_int_t
ngx_http_not_modified_header_filter(ngx_http_request_t *r)
{
    if (r->headers_out.status != NGX_HTTP_OK
        || r != r->main
        || r->headers_out.last_modified_time == -1)
    {
        return ngx_http_next_header_filter(r);
    }
}
```

```

    }

    if (r->headers_in.if_unmodified_since) {
        return ngx_http_test_precondition(r);
    }

    if (r->headers_in.if_modified_since) {
        return ngx_http_test_not_modified(r);
    }

    return ngx_http_next_header_filter(r);
}

```

而在 ngx_http_not_modified_header_filter 函数中，它会调用模块内部定义的函数指针变量 ngx_http_next_header_filter，而该变量保存的是上一模块注册的 header filter 函数，同样的下一个 header filter 函数内部也会调用其模块内部的 ngx_http_next_header_filter，直到调用到最后一个 header filter - ngx_http_header_filter。

ngx_http_header_filter，这个 filter 负责计算响应头的总大小，并分配内存，组装响应头，并调用 ngx_http_write_filter 发送。Nginx 中，header filter 只会被调用一次，ngx_http_header_filter 函数中首先会检查 r->header_sent 标识是否已经被设置，如果是的话，则直接返回；否则设置该标识，并发送响应头。另外如果是子请求的话，也会直接退出函数。

body filter 分析

Nginx 中通常调用 ngx_http_output_filter 函数来发送响应体，它的实现如下：

```

ngx_int_t
ngx_http_output_filter(ngx_http_request_t *r, ngx_chain_t *in)
{
    ngx_int_t      rc;
    ngx_connection_t *c;

    c = r->connection;

    ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http output filter \"%V?%V\"", &r->uri, &r->args);

    rc = ngx_http_top_body_filter(r, in);

    if (rc == NGX_ERROR) {

```

```
    /* NGX_ERROR may be returned by any filter */
    c->error = 1;
}

return rc;
}
```

body filter 链调用的原理和 header filter 一样，和 ngx_http_send_header 函数不同的是，上面的函数多了一个类型为 ngx_chain_t * 的参数，因为 Nginx 实现的是流式的输出，并不用等到整个响应体都生成了才往客户端发送数据，而是产生一部分内容之后将其组织成链表，调用 ngx_http_output_filter 发送，并且待发送的内容可以在文件中，也可以是在内存中，Nginx 会负责将数据流式的，高效的传输出去。而且当发送缓存区满了时，Nginx 还会负责保存未发送完的数据，调用者只需要对新数据调用一次 ngx_http_output_filter 即可。

ngx_http_copy_filter_module 分析

ngx_http_copy_filter_module 是响应体过滤链（body filter）中非常重要的一个模块，这个 filter 模块主要是来将一些需要复制的 buf（可能在文件中，也可能在内存中）重新复制一份交给后面的 filter 模块处理。先来看它的初始化函数：

```
static ngx_int_t
ngx_http_copy_filter_init(ngx_conf_t *cf)
{
    ngx_http_next_body_filter = ngx_http_top_body_filter;
    ngx_http_top_body_filter = ngx_http_copy_filter;

    return NGX_OK;
}
```

可以看到，它只注册了 body filter，而没有注册 header filter，也就是说只有 body filter 链中才有这个模块。

该模块有一个命令，命令名为 output_buffers，用来配置可用的 buffer 数和 buffer 大小，它的值保存在 copy filter 的 loc conf 的 bufs 字段，默认数量为 1，大小为 32768 字节。这个参数具体的作用后面会做介绍。

Nginx 中，一般 filter 模块可以 header filter 函数中根据请求响应头设置一个模块上下文（context），用来保存相关的信息，在 body filter 函数中使用这个上下文。而 copy filter 没有 header filter，因此它的 context 的初始化也是放在 body filter 中的，而它的 ctx 就是 ngx_output_chain_ctx_t，为什么名字是 output_chain 呢，这是因为 copy filter 的主要逻辑的处理都放在 ngx_output_chain 模块中，另外这个模块在 core 目录下，而不是属于 http 目录。

接下来看一下上面说到的 context 结构：

```
struct ngx_output_chain_ctx_s {
    ngx_buf_t          *buf;           /* 保存临时的 buf */
    ngx_chain_t        *in;           /* 保存了将要发送的 chain */
    ngx_chain_t        *free;         /* 保存了已经发送完毕的 chain，以
    便于重复利用 */
    ngx_chain_t        *busy;         /* 保存了还未发送的 chain */

    unsigned            sendfile:1;    /* sendfile 标记 */
    unsigned            directio:1;    /* directio 标记 */
    #if (NGX_HAVE_ALIGNED_DIRECTIO)
        unsigned        unaligned:1;
    #endif
    unsigned            need_in_memory:1; /* 是否需要在内存中保存一份(使用
    sendfile 的话，                                内存中没有文件的拷贝的，而我们有
    时需要处理文件，                                此时就需要设置这个标记) */
    unsigned            need_in_temp:1; /* 是否需要在内存中重新复制一份，
    不管 buf 是在内存还是文件，                    这样的话，后续模块可以直接修改这
    块内存 */
    #if (NGX_HAVE_FILE_AIO)
        unsigned        aio:1;
        ngx_output_chain_aio_pt aio_handler;
    #endif

    off_t              alignment;

    ngx_pool_t         *pool;
    ngx_int_t          allocated;      /* 已经分配的 buf 个数 */
    ngx_bufs_t         bufs;          /* 对应 loc conf 中设置的 bufs */
    ngx_buf_tag_t      tag;           /* 模块标记，主要用于 buf 回收 */
}
```



```

    ngx_output_chain_filter_pt    output_filter;    /* 一般是 ngx_http_next_filter,
也就是继续调用 filter 链 */
    void                          *filter_ctx;      /* 当前 filter 的上下文,
                                                    这里是由于 upstream 也会调用
output_chain */
};

```

为了更好的理解 context 结构每个域的具体含义，接下来分析 filter 的具体实现：

```

static ngx_int_t
ngx_http_copy_filter(ngx_http_request_t *r, ngx_chain_t *in)
{
    ngx_int_t                rc;
    ngx_connection_t         *c;
    ngx_output_chain_ctx_t    *ctx;
    ngx_http_core_loc_conf_t  *clcf;
    ngx_http_copy_filter_conf_t *conf;

    c = r->connection;

    ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
        "http copy filter: \"%V?%V\"", &r->uri, &r->args);

    /* 获取 ctx */
    ctx = ngx_http_get_module_ctx(r, ngx_http_copy_filter_module);

    /* 如果为空，则说明需要初始化 ctx */
    if (ctx == NULL) {
        ctx = ngx_palloc(r->pool, sizeof(ngx_output_chain_ctx_t));
        if (ctx == NULL) {
            return NGX_ERROR;
        }

        ngx_http_set_ctx(r, ctx, ngx_http_copy_filter_module);

        conf = ngx_http_get_module_loc_conf(r, ngx_http_copy_filter_module);
        clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

        /* 设置 sendfile */
        ctx->sendfile = c->sendfile;
        /* 如果 request 设置了 filter_need_in_memory 的话，ctx 的这个域就会被设置 */
        ctx->need_in_memory = r->main_filter_need_in_memory
            || r->filter_need_in_memory;
        /* 和上面类似 */
        ctx->need_in_temp = r->filter_need_temporary;

        ctx->alignment = clcf->directio_alignment;
    }
}

```

```

    ctx->pool = r->pool;
    ctx->bufs = conf->bufs;
    ctx->tag = (ngx_buf_tag_t) &ngx_http_copy_filter_module;
    /* 可以看到 output_filter 就是下一个 body filter 节点 */
    ctx->output_filter = (ngx_output_chain_filter_pt)
                        ngx_http_next_body_filter;
    /* 此时 filter ctx 为当前的请求 */
    ctx->filter_ctx = r;

    ...

    if (in && in->buf && ngx_buf_size(in->buf)) {
        r->request_output = 1;
    }
}

...

for ( ;; ) {
    /* 最关键的函数，下面会详细分析 */
    rc = ngx_output_chain(ctx, in);

    if (ctx->in == NULL) {
        r->buffered &= ~NGX_HTTP_COPY_BUFFERED;

    } else {
        r->buffered |= NGX_HTTP_COPY_BUFFERED;
    }

    ...

    return rc;
}
}

```

上面的代码去掉了 AIO 相关的部分，函数首先设置并初始化 context，接着调用 ngx_output_chain 函数，这个函数实际上包含了 copy filter 模块的主要逻辑，它的原型为：

```
ngx_int_t ngx_output_chain(ngx_output_chain_ctx_t *ctx, ngx_chain_t *in)
```

分段来看它的代码，下面这段代码是一个快捷路径(short path)，也就是说当能直接确定所有的 in chain 都不需要复制的时，可以直接调用 output_filter 来交给剩下的 filter 去处理：

```
if (ctx->in == NULL && ctx->busy == NULL) {
```

```

/*
 * the short path for the case when the ctx->in and ctx->busy chains
 * are empty, the incoming chain is empty too or has the single buf
 * that does not require the copy
 */

if (in == NULL) {
    return ctx->output_filter(ctx->filter_ctx, in);
}

if (in->next == NULL
#ifdef NGX_SENDFILE_LIMIT)
    && !(in->buf->in_file && in->buf->file_last > NGX_SENDFILE_LIMIT)
#endif
    && ngx_output_chain_as_is(ctx, in->buf))
{
    return ctx->output_filter(ctx->filter_ctx, in);
}
}

```

上面可以看到了一个函数 `ngx_output_chain_as_is`，这个函数很关键，下面还会再次被调用，这个函数主要用来判断是否需要复制 `buf`。返回 1,表示不需要拷贝，否则为需要拷贝：

```

static ngx_inline ngx_int_t
ngx_output_chain_as_is(ngx_output_chain_ctx_t *ctx, ngx_buf_t *buf)
{
    ngx_uint_t  sendfile;

    /* 是否为特殊 buf (special buf)，是的话返回 1，也就是不用拷贝 */
    if (ngx_buf_special(buf)) {
        return 1;
    }

    /* 如果 buf 在文件中，并且使用了 directio 的话，需要拷贝 buf */
    if (buf->in_file && buf->file->directio) {
        return 0;
    }

    /* sendfile 标记 */
    sendfile = ctx->sendfile;

#ifdef NGX_SENDFILE_LIMIT)
    /* 如果 pos 大于 sendfile 的限制，设置标记为 0 */
    if (buf->in_file && buf->file_pos >= NGX_SENDFILE_LIMIT) {
        sendfile = 0;
    }

```

```

#endif

    if (!sendfile) {
        /* 如果不走 sendfile, 而且 buf 不在内存中, 则我们就需要复制到内存一份 */
        if (!ngx_buf_in_memory(buf)) {
            return 0;
        }

        buf->in_file = 0;
    }

    /* 如果需要内存中有一份拷贝, 而并不在内存中, 此时返回 0, 表示需要拷贝 */
    if (ctx->need_in_memory && !ngx_buf_in_memory(buf)) {
        return 0;
    }

    /* 如果需要内存中有可修改的拷贝, 并且 buf 存在于只读的内存中或者 mmap 中, 则返回 0 */
    if (ctx->need_in_temp && (buf->memory || buf->mmap)) {
        return 0;
    }

    return 1;
}

```

上面有两个标记要注意, 一个是 `need_in_memory`, 这个主要是用于当使用 `sendfile` 的时候, Nginx 并不会将请求文件拷贝到内存中, 而有时需要操作文件的内容, 此时就需要设置这个标记。然后后面的 `body filter` 就能操作内容了。

第二个是 `need_in_temp`, 这个主要是用于把本来就存在于内存中的 `buf` 复制一份可修改的拷贝出来, 这里有用到的模块有 `charset`, 也就是编解码 `filter`。

然后接下来这段是复制 `in chain` 到 `ctx->in` 的结尾, 它是通过调用 `ngx_output_chain_add_copy` 来进行 `add copy` 的, 这个函数比较简单, 这里就不分析了, 不过只有一个要注意的地方, 那就是如果 `buf` 是存在于文件中, 并且 `file_pos` 超过了 `sendfile limit`, 此时就会切割 `buf` 为两个 `buf`, 然后保存在两个 `chain` 中, 最终连接起来:

```

/* add the incoming buf to the chain ctx->in */

if (in) {
    if (ngx_output_chain_add_copy(ctx->pool, &ctx->in, in) == NGX_ERROR) {
        return NGX_ERROR;
    }
}

```

```
}  
}
```

然后就是主要的逻辑处理阶段。这里 nginx 做的非常巧妙也非常复杂，首先是 chain 的重用，然后是 buf 的重用。

先来看 chain 的重用。关键的几个结构以及域：ctx 的 free，busy 以及 ctx->pool 的 chain 域。

其中每次发送没有发完的 chain 就放到 busy 中，而已经发送完毕的就放到 free 中，而最后会调用 ngx_free_chain 来将 free 的 chain 放入到 pool->chain 中，而在 ngx_alloc_chain_link 中，如果 pool->chain 中存在 chain 的话，就不用 malloc 了，而是直接返回 pool->chain，相关的代码如下：

```
/* 链接 cl 到 pool->chain 中 */  
#define ngx_free_chain(pool, cl)                                \\\br/>    cl->next = pool->chain;                                       \\\br/>    pool->chain = cl  
  
/* 从 pool 中分配 chain */  
ngx_chain_t *  
ngx_alloc_chain_link(ngx_pool_t *pool)  
{  
    ngx_chain_t  *cl;  
  
    cl = pool->chain;  
    /* 如果 cl 存在，则直接返回 cl */  
    if (cl) {  
        pool->chain = cl->next;  
        return cl;  
    }  
    /* 否则才会 malloc chain */  
    cl = ngx_palloc(pool, sizeof(ngx_chain_t));  
    if (cl == NULL) {  
        return NULL;  
    }  
  
    return cl;  
}
```

然后是 buf 的重用，严格意义上来说 buf 的重用是从 free 中的 chain 中取得的，当 free 中的 buf 被重用，则这个 buf 对应的 chain 就会被链接到 ctx->pool 中，从而这个 chain 就会被重用。也就是说首先

考虑的是 buf 的重用，只有当这个 chain 的 buf 确定不需要被重用(或者说已经被重用)的时候，chain 才会被链接到 ctx->pool 中被重用。

还有一个就是 ctx 的 allocated 域，这个域表示了当前的上下文中已经分配了多少个 buf，output_buffer 命令用来设置 output 的 buf 大小以及 buf 的个数。而 allocated 如果比 output_buffer 大的话，则需要先发送完已经存在的 buf，然后才能再次重新分配 buf。

来看代码，上面所说的重用以及 buf 的控制，代码里面都可以看的比较清晰。下面这段主要是拷贝 buf 前所做的一些工作，比如判断是否拷贝，以及给 buf 分配内存等：

```
/* out 为最终需要传输的 chain，也就是交给剩下的 filter 处理的 chain */
out = NULL;
/* last_out 为 out 的最后一个 chain */
last_out = &out;
last = NGX_NONE;

for ( ;; ) {

    /* 开始遍历 chain */
    while (ctx->in) {

        /* 取得当前 chain 的 buf 大小 */
        bsize = ngx_buf_size(ctx->in->buf);

        /* 跳过 bsize 为 0 的 buf */
        if (bsize == 0 && !ngx_buf_special(ctx->in->buf)) {
            ngx_debug_point();

            ctx->in = ctx->in->next;

            continue;
        }

        /* 判断是否需要复制 buf */
        if (ngx_output_chain_as_is(ctx, ctx->in->buf)) {

            /* move the chain link to the output chain */
            /* 如果不需要复制，则直接链接 chain 到 out，然后继续循环 */
            cl = ctx->in;
            ctx->in = cl->next;

            *last_out = cl;
            last_out = &cl->next;
        }
    }
}
```

```

    cl->next = NULL;

    continue;
}

/* 到达这里，说明我们需要拷贝 buf，这里 buf 最终都会被拷贝进 ctx->buf 中，
因此这里先判断 ctx->buf 是否为空 */
if (ctx->buf == NULL) {

    /* 如果为空，则取得 buf，这里要注意，一般来说如果没有开启 directio 的话，
    这个函数都会返回 NGX_DECLINED */
    rc = ngx_output_chain_align_file_buf(ctx, bsize);

    if (rc == NGX_ERROR) {
        return NGX_ERROR;
    }

    /* 大部分情况下，都会落入这个分支 */
    if (rc != NGX_OK) {

        /* 准备分配 buf，首先在 free 中寻找可以重用的 buf */
        if (ctx->free) {

            /* get the free buf */
            /* 得到 free buf */
            cl = ctx->free;
            ctx->buf = cl->buf;
            ctx->free = cl->next;
            /* 将要重用的 chain 链接到 ctx->pool 中，以便于 chain 的重用 */
            ngx_free_chain(ctx->pool, cl);

        } else if (out || ctx->allocated == ctx->bufs.num) {
            /* 如果已经等于 buf 的个数限制，则跳出循环，发送已经存在的 buf。
            这里可以看到如果 out 存在的话，nginx 会跳出循环，然后发送 out，
            等发送完会再次处理，这里很好的体现了 nginx 的流式处理 */
            break;
        } else if (ngx_output_chain_get_buf(ctx, bsize) != NGX_OK) {
            /* 上面这个函数也比较关键，它用来取得 buf。接下来会详细看这个函数 */
            return NGX_ERROR;
        }
    }
}

/* 从原来的 buf 中拷贝内容或者从文件中读取内容 */
rc = ngx_output_chain_copy_buf(ctx);

if (rc == NGX_ERROR) {
    return rc;
}

```

```

    }

    if (rc == NGX_AGAIN) {
        if (out) {
            break;
        }

        return rc;
    }

    /* delete the completed buf from the ctx->in chain */

    if (ngx_buf_size(ctx->in->buf) == 0) {
        ctx->in = ctx->in->next;
    }
    /* 分配新的 chain 节点 */
    cl = ngx_alloc_chain_link(ctx->pool);
    if (cl == NULL) {
        return NGX_ERROR;
    }

    cl->buf = ctx->buf;
    cl->next = NULL;
    *last_out = cl;
    last_out = &cl->next;
    ctx->buf = NULL;
}
...
}

```

上面的代码分析的时候有个很关键的函数，那就是 `ngx_output_chain_get_buf`，这个函数当没有可重用的 buf 时用来分配 buf。

如果当前的 buf 位于最后一个 chain，则需要特殊处理，一是 buf 的 `recycled` 域，另外是将要分配的 buf 的大小。

先来说 `recycled` 域，这个域表示当前的 buf 需要被回收。而一般情况下 Nginx(比如在非 last buf)会缓存一部分 buf(默认是 1460 字节)，然后再发送，而设置了 `recycled` 的话，就不会让它缓存 buf，也就是尽量发送出去，然后以供回收使用。因此如果是最后一个 buf，则不需要设置 `recycled` 域的，否则的话，需要设置 `recycled` 域。

然后就是 buf 的大小。这里会有两个大小，一个是需要复制的 buf 的大小，一个是配置文件中设置的 buf 的大小。如果不是最后一个 buf，则只需要分配配置中设置的 buf 的大小就行了。如果是最后一个 buf，则就处理不太一样，下面的代码会看到：

```
static ngx_int_t
ngx_output_chain_get_buf(ngx_output_chain_ctx_t *ctx, off_t bsize)
{
    size_t      size;
    ngx_buf_t   *b, *in;
    ngx_uint_t   recycled;

    in = ctx->in->buf;
    /* 可以看到这里分配的 buf，每个 buf 的大小是配置文件中设置的 size */
    size = ctx->bufs.size;
    /* 默认有设置 recycled 域 */
    recycled = 1;
    /* 如果当前的 buf 是属于最后一个 chain 的时候，需要特殊处理 */
    if (in->last_in_chain) {
        /* 如果 buf 大小小于配置指定的大小，则直接按实际大小分配，不设置回收标记 */
        if (bsize < (off_t) size) {

            /*
             * allocate a small temp buf for a small last buf
             * or its small last part
             */
            size = (size_t) bsize;
            recycled = 0;

        } else if (!ctx->directio
                   && ctx->bufs.num == 1
                   && (bsize < (off_t) (size + size / 4)))
        {
            /*
             * allocate a temp buf that equals to a last buf,
             * if there is no directio, the last buf size is lesser
             * than 1.25 of bufs.size and the temp buf is single
             */

            size = (size_t) bsize;
            recycled = 0;
        }
    }
    /* 开始分配 buf 内存 */
    b = ngx_calloc_buf(ctx->pool);
    if (b == NULL) {
        return NGX_ERROR;
    }
}
```

```

if (ctx->directio) {
    /* directio 需要对齐 */

    b->start = ngx_pmemalign(ctx->pool, size, (size_t) ctx->alignment);
    if (b->start == NULL) {
        return NGX_ERROR;
    }

} else {
    /* 大部分情况会走到这里 */
    b->start = ngx_palloc(ctx->pool, size);
    if (b->start == NULL) {
        return NGX_ERROR;
    }
}

b->pos = b->start;
b->last = b->start;
b->end = b->last + size;
/* 设置 temporary */
b->temporary = 1;
b->tag = ctx->tag;
b->recycled = recycled;

ctx->buf = b;
/* 更新 allocated, 可以看到每分配一个就加 1 */
ctx->allocated++;

return NGX_OK;
}

```

分配新的 buf 和 chain，并调用 ngx_output_chain_copy_buf 拷贝完数据之后，Nginx 就将新的 chain 链表交给下一个 body filter 继续处理：

```

if (out == NULL && last != NGX_NONE) {

    if (ctx->in) {
        return NGX_AGAIN;
    }

    return last;
}

last = ctx->output_filter(ctx->filter_ctx, out);

if (last == NGX_ERROR || last == NGX_DONE) {

```

```

    return last;
}

ngx_chain_update_chains(ctx->pool, &ctx->free, &ctx->busy, &out,
                        ctx->tag);
last_out = &out;

```

在其他 body filter 处理完之后，ngx_output_chain 函数还需要更新 chain 链表，以便回收利用，ngx_chain_update_chains 函数主要是将处理完毕的 chain 节点放入到 free 链表，没有处理完毕的放到 busy 链表中，另外这个函数用到了 tag，它只回收 copy filter 产生的 chain 节点。

ngx_http_write_filter_module 分析

ngx_http_write_filter_module 是最后一个 body filter，可以看到它的注册函数的特殊性：

```

static ngx_int_t
ngx_http_write_filter_init(ngx_conf_t *cf)
{
    ngx_http_top_body_filter = ngx_http_write_filter;

    return NGX_OK;
}

```

ngx_http_write_filter_module 是第一个注册 body filter 的模块，于是它也是最后一个执行的 body filter 模块。

直接来看 ngx_http_write_filter，下面的代码中去掉了一些调试代码：

```

ngx_int_t
ngx_http_write_filter(ngx_http_request_t *r, ngx_chain_t *in)
{
    off_t                size, sent, nsent, limit;
    ngx_uint_t           last, flush;
    ngx_msec_t           delay;
    ngx_chain_t          *cl, *ln, **ll, *chain;
    ngx_connection_t     *c;
    ngx_http_core_loc_conf_t *clcf;

    c = r->connection;

    if (c->error) {
        return NGX_ERROR;
    }
}

```

```

}

size = 0;
flush = 0;
last = 0;
ll = &r->out;

/* find the size, the flush point and the last link of the saved chain */

for (cl = r->out; cl; cl = cl->next) {
    ll = &cl->next;

#if 1
    if (ngx_buf_size(cl->buf) == 0 && !ngx_buf_special(cl->buf)) {
        return NGX_ERROR;
    }
#endif

    size += ngx_buf_size(cl->buf);

    if (cl->buf->flush || cl->buf->recycled) {
        flush = 1;
    }

    if (cl->buf->last_buf) {
        last = 1;
    }
}

/* add the new chain to the existent one */

for (ln = in; ln; ln = ln->next) {
    cl = ngx_alloc_chain_link(r->pool);
    if (cl == NULL) {
        return NGX_ERROR;
    }

    cl->buf = ln->buf;
    *ll = cl;
    ll = &cl->next;

#if 1
    if (ngx_buf_size(cl->buf) == 0 && !ngx_buf_special(cl->buf)) {
        return NGX_ERROR;
    }
#endif

    size += ngx_buf_size(cl->buf);

```

```

    if (cl->buf->flush || cl->buf->recycled) {
        flush = 1;
    }

    if (cl->buf->last_buf) {
        last = 1;
    }
}

*ll = NULL;

clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

/*
 * avoid the output if there are no last buf, no flush point,
 * there are the incoming bufs and the size of all bufs
 * is smaller than "postpone_output" directive
 */

if (!last && !flush && in && size < (off_t) clcf->postpone_output) {
    return NGX_OK;
}

/* 如果请求由于被限速而必须延迟发送时，设置一个标识后退出 */
if (c->write->delayed) {
    c->buffered |= NGX_HTTP_WRITE_BUFFERED;
    return NGX_AGAIN;
}

/* 如果 buffer 总大小为 0，而且当前连接之前没有由于底层发送接口的原因延迟，
   则检查是否有特殊标记 */
if (size == 0 && !(c->buffered & NGX_LOWLEVEL_BUFFERED)) {
    /* last_buf 标记，表示请求体已经发送结束 */
    if (last) {
        r->out = NULL;
        c->buffered &= ~NGX_HTTP_WRITE_BUFFERED;

        return NGX_OK;
    }

    /* flush 生效，而且又没有实际数据，则清空当前的未发送队列 */
    if (flush) {
        do {
            r->out = r->out->next;
        } while (r->out);

        c->buffered &= ~NGX_HTTP_WRITE_BUFFERED;

        return NGX_OK;
    }
}

```

```

    return NGX_ERROR;
}
/* 请求有速率限制，则计算当前可以发送的大小 */
if (r->limit_rate) {
    limit = r->limit_rate * (ngx_time() - r->start_sec + 1)
           - (c->sent - clcf->limit_rate_after);

    if (limit <= 0) {
        c->write->delayed = 1;
        ngx_add_timer(c->write,
                      (ngx_msec_t) (- limit * 1000 / r->limit_rate + 1));

        c->buffered |= NGX_HTTP_WRITE_BUFFERED;

        return NGX_AGAIN;
    }

    if (clcf->sendfile_max_chunk
        && (off_t) clcf->sendfile_max_chunk < limit)
    {
        limit = clcf->sendfile_max_chunk;
    }
} else {
    limit = clcf->sendfile_max_chunk;
}

sent = c->sent;
/* 发送数据 */
chain = c->send_chain(c, r->out, limit);

if (chain == NGX_CHAIN_ERROR) {
    c->error = 1;
    return NGX_ERROR;
}
/* 更新限速相关的信息 */
if (r->limit_rate) {

    nsent = c->sent;

    if (clcf->limit_rate_after) {

        sent -= clcf->limit_rate_after;
        if (sent < 0) {
            sent = 0;
        }
    }
}

```

```

        nsent -= clcf->limit_rate_after;
        if (nsent < 0) {
            nsent = 0;
        }
    }

    delay = (ngx_msec_t) ((nsent - sent) * 1000 / r->limit_rate);

    if (delay > 0) {
        limit = 0;
        c->write->delayed = 1;
        ngx_add_timer(c->write, delay);
    }
}

if (limit
    && c->write->ready
    && c->sent - sent >= limit - (off_t) (2 * ngx_pagesize))
{
    c->write->delayed = 1;
    ngx_add_timer(c->write, 1);
}
/* 更新输出链，释放已经发送的节点 */
for (cl = r->out; cl && cl != chain; /* void */) {
    ln = cl;
    cl = cl->next;
    ngx_free_chain(r->pool, ln);
}

r->out = chain;
/* 如果数据未发送完毕，则设置一个标记 */
if (chain) {
    c->buffered |= NGX_HTTP_WRITE_BUFFERED;
    return NGX_AGAIN;
}

c->buffered &= ~NGX_HTTP_WRITE_BUFFERED;
/* 如果由于底层发送接口导致数据未发送完全，且当前请求没有其他数据需要发送，
   此时要返回 NGX_AGAIN，表示还有数据未发送 */
if ((c->buffered & NGX_LOWLEVEL_BUFFERED) && r->postponed == NULL) {
    return NGX_AGAIN;
}

return NGX_OK;
}

```

Nginx 将待发送的 chain 链表保存在 r->out，上面的函数先检查之前未发送完的链表中是否有 flush，recycled 以及 last_buf 标识，并计算所有 buffer 的大小，接着对新输入的 chain 链表做同样的事情，并将新链表加到 r->out 的队尾。

如果没有输出链表中没有被标识为最后一块 buffer 的节点，而且没有需要 flush 或者急着回收的 buffer，并且当前队列中 buffer 总大小不够 postpone_output 指令设置的大小（默认为 1460 字节）时，函数会直接返回。

ngx_http_write_filter 会调用 c->send_chain 往客户端发送数据，c->send_chain 的取值在不同操作系统，编译选项以及协议下(https 下用的是 ngx_ssl_send_chain)会取不同的函数，典型的 linux 操作系统下，它的取值为 ngx_linux_sendfile_chain，也就是最终会调用这个函数来发送数据。它的函数原型为：

```
ngx_chain_t *  
ngx_linux_sendfile_chain(ngx_connection_t *c, ngx_chain_t *in, off_t limit)
```

第一个参数是当前的连接，第二个参数是所需要发送的 chain，第三个参数是所能发送的最大值。

首先看一下这个函数定义的一些重要局部变量：

send 表示将要发送的 buf 已经已经发送的大小；

sent 表示已经发送的 buf 的大小；

prev_send 表示上一次发送的大小，也就是已经发送的 buf 的大小；

fprev 和 prev-send 类似，只不过是 file 类型的；

complete 表示是否 buf 被完全发送了，也就是 sent 是否等于 send - prev_send；

header 表示需要是用 writev 来发送的 buf，也就是 only in memory 的 buf；

struct iovec *iov, headers[NGX_HEADERS] 这个主要是用于 sendfile 和 writev 的参数，这里注意上面 header 数组保存的就是 iovec。

下面看函数开头的一些初始化代码：


```

wev = c->write;

if (!wev->ready) {
    return in;
}

/* the maximum limit size is 2G-1 - the page size */

if (limit == 0 || limit > (off_t) (NGX_SENDFILE_LIMIT - ngx_pagesize)) {
    limit = NGX_SENDFILE_LIMIT - ngx_pagesize;
}

send = 0;

/* 设置 header, 也就是 in memory 的数组 */
header.elts = headers;
header.size = sizeof(struct iovec);
header.nalloc = NGX_HEADERS;
header.pool = c->pool;

```

下面这段代码就是处理 in memory 的部分, 然后将 buf 放入对应的 iovec 数组, 处理核心思想就是合并内存连续并相邻的 buf(不管是 in memory 还是 in file):

```

for (cl = in; cl && send < limit; cl = cl->next) {

    if (ngx_buf_special(cl->buf)) {
        continue;
    }

    /* 如果既不在内存中, 又不在文件中, 则返回错误 */
    if (!ngx_buf_in_memory(cl->buf) && !cl->buf->in_file) {
        return NGX_CHAIN_ERROR;
    }

    /* 如果不只是在 buf 中, 这是因为有时 in file 的 buf 可能需要内存中也有拷贝,
       如果一个 buf 同时 in memoey 和 in file 的话, Nginx 会把它当做 in file 来处理 */
    if (!ngx_buf_in_memory_only(cl->buf)) {
        break;
    }

    /* 得到 buf 的大小 */
    size = cl->buf->last - cl->buf->pos;

    /* 大于 limit 的话修改为 size */
    if (send + size > limit) {
        size = limit - send;
    }
}

```

```

}

/* 如果 prev 等于 pos，则说明当前的 buf 的数据和前一个 buf 的数据是连续的 */
if (prev == cl->buf->pos) {
    iov->iov_len += (size_t) size;

} else {
    if (header.nelts >= IOV_MAX) {
        break;
    }
    /* 否则说明是不同的 buf，因此增加一个 iovc */
    iov = ngx_array_push(&header);
    if (iov == NULL) {
        return NGX_CHAIN_ERROR;
    }

    iov->iov_base = (void *) cl->buf->pos;
    iov->iov_len = (size_t) size;
}

/* 这里可以看到 prev 保存了当前 buf 的结尾 */
prev = cl->buf->pos + (size_t) size;
/* 更新发送的大小 */
send += size;
}

```

然后是 in file 的处理，这里比较核心的一个判断就是 `fprev == cl->buf->file_pos`，和上面的 in memory 类似，`fprev` 保存的就是上一次处理的 buf 的尾部。这里如果这两个相等，那就说明当前的两个 buf 是连续的(文件连续)：

```

/* 如果 header 的大小不为 0 则说明前面有需要发送的 buf，
   并且数据大小已经超过限制则跳过 in file 处理 */
if (header.nelts == 0 && cl && cl->buf->in_file && send < limit) {
    /* 得到 file
       file = cl->buf;

    /* 开始合并 */
    do {
        /* 得到大小 */
        size = cl->buf->file_last - cl->buf->file_pos;

        /* 如果太大则进行对齐处理 */
        if (send + size > limit) {
            size = limit - send;

            aligned = (cl->buf->file_pos + size + ngx_pagesize - 1)

```

```

        & ~((off_t) ngx_pagesize - 1);

    if (aligned <= cl->buf->file_last) {
        size = aligned - cl->buf->file_pos;
    }
}

/* 设置 file_size */
file_size += (size_t) size;
/* 设置需要发送的大小 */
send += size;
/* 和上面的 in memory 处理一样就是保存这次的 last */
fprev = cl->buf->file_pos + size;
cl = cl->next;

} while (cl
        && cl->buf->in_file
        && send < limit
        && file->file->fd == cl->buf->file->fd
        && fprev == cl->buf->file_pos);
}

```

然后就是发送部分，这里 in file 使用 sendfile，in memory 使用 writev。处理逻辑比较简单，就是发送后判断发送成功的大小

```

if (file) {
#ifdef 1
    if (file_size == 0) {
        ngx_debug_point();
        return NGX_CHAIN_ERROR;
    }
#endif
#ifdef (NGX_HAVE_SENDFILE64)
    offset = file->file_pos;
#else
    offset = (int32_t) file->file_pos;
#endif

    /* 数据在文件中则调用 sendfile 发送数据 */
    rc = sendfile(c->fd, file->file->fd, &offset, file_size);

    ...

    /* 得到发送成功的字节数 */
    sent = rc > 0 ? rc : 0;

} else {
    /* 数据在内存中则调用 writev 发送数据 */

```

```

    rc = writev(c->fd, header.elts, header.nelts);

    ...
    /* 得到发送成功的字节数 */
    sent = rc > 0 ? rc : 0;
}

```

接下来就是需要根据发送成功的字节数来更新 chain:

```

/* 如果 send - prev_send == sent 则说明该发送的都发完了 */
if (send - prev_send == sent) {
    complete = 1;
}
/* 更新 conngect 的 sent 域 */
c->sent += sent;

/* 开始重新遍历 chain, 这里是为了防止没有发送完全的情况,
   此时我们就需要切割 buf 了 */
for (cl = in; cl; cl = cl->next) {

    if (ngx_buf_special(cl->buf)) {
        continue;
    }

    if (sent == 0) {
        break;
    }
    /* 得到 buf size */
    size = ngx_buf_size(cl->buf);

    /* 如果大于当前的 size, 则说明这个 buf 的数据已经被完全发送完毕了,
       因此更新它的域 */
    if (sent >= size){
        /* 更新 sent 域 */
        sent -= size;
        /* 如果在内存则更新 pos */
        if (ngx_buf_in_memory(cl->buf)) {
            cl->buf->pos = cl->buf->last;
        }
        /* 如果在 file 中则更 file_pos */
        if (cl->buf->in_file) {
            cl->buf->file_pos = cl->buf->file_last;
        }

        continue;
    }

    /* 到这里说明当前的 buf 只有一部分被发送出去了, 因此只需要修改指针。

```

```

    以便于下次发送 */
    if (ngx_buf_in_memory(cl->buf)) {
        cl->buf->pos += (size_t) sent;
    }
    /* 同上 */
    if (cl->buf->in_file) {
        cl->buf->file_pos += sent;
    }

    break;
}

```

最后一部分是一些是否退出循环的判断。这里要注意，Nginx 中如果发送未完全的话，将会直接返回，返回的就是没有发送完毕的 chain，它的 buf 也已经被更新。然后 Nginx 返回去处理其他的事情，等待可写之后再次发送未发送完的数据：

```

if (eintr) {
    continue;
}
/* 如果未完成，则设置 wev->ready 为 0 后返回 */
if (!complete) {
    wev->ready = 0;
    return cl;
}
/* 发送数据超过限制，或没有数据了 */
if (send >= limit || cl == NULL) {
    return cl;
}
/* 更新 in，也就是开始处理下一个 chain */
in = cl;

```

subrequest 原理解析 (99%)

子请求并不是 http 标准里面的概念，它是在当前请求中发起的一个新的请求，它拥有自己的

ngx_http_request_t 结构，uri 和 args。一般来说使用 subrequest 的效率可能会有些影响，因为它需要重新从 server rewrite 开始走一遍 request 处理的 PHASE，但是它在某些情况下使用能带来方便，比较常用的是用 subrequest 来访问一个 upstream 的后端，并给它一个 ngx_http_post_subrequest_t 的回调 handler，这样有点类似于一个异步的函数调用。对于从 upstream 返回的数据，subrequest 允许根据创建时指定的 flag，来决定由用户自己处理(回调 handler 中)还是由 upstream 模块直接发送到 out put

filter。简单的说一下 subrequest 的行为，nginx 使用 subrequest 访问某个 location，产生相应的数据，并插入到 nginx 输出链的相应位置（创建 subrequest 时的位置），下面用 nginx 代码内的 addition 模块(默认未编译进 nginx 核心，请使用--with-http_addition_module 选项包含此模块)来举例说明一下：

```
location /main.htm {
    # content of main.htm: main
    add_before_body /hello.htm;
    add_after_body /world.htm;
}
location /hello.htm {
    #content of hello.htm: hello
}
location /world.htm {
    #content of world.htm: world
}
```

访问/main.htm，将得到如下响应：

```
hello
main
world
```

上面的 add_before_body 指令发起一个 subrequest 来访问/hello.htm，并将产生的内容(hello)插入主请求响应体的开头，add_after_body 指令发起一个 subrequest 访问/world.htm，并将产生的内容(world)附加在主请求响应体的结尾。addition 模块是一个 filter 模块，但是 subrequest 既可以在 phase 模块中使用，也可以在 filter 模块中使用。

在进行源码解析之前，先来想想如果是我们自己要实现 subrequest 的上述行为，该如何来做？

subrequest 还可能有自己的 subrequest，而且每个 subrequest 都不一定按照其创建的顺序来输出数据，所以简单的采用链表不好实现，于是进一步联想到可以采用树的结构来做，主请求即为根节点，每个节点可以有自己的子节点，遍历某节点表示处理某请求，自然的可以想到这里可能是用后根(序)遍历的方法，没错，实际上 Igor 采用树和链表结合的方式实现了 subrequest 的功能，但是由于节点（请求）产生数据的顺序不是固定按节点创建顺序(左->右)，而且可能分多次产生数据，不能简单的用后根(序)遍历。Igor 使用了 2 个链表的结构来实现，第一个是每个请求都有的 postponed 链表，一般情况下每个链表节点保存了该请求的一个子请求，该链表节点定义如下：

```

struct ngx_http_postponed_request_s {
    ngx_http_request_t      *request;
    ngx_chain_t             *out;
    ngx_http_postponed_request_t *next;
};

```

可以看到它有一个 request 字段，可以用来保存子请求，另外还有一个 ngx_chain_t 类型的 out 字段，实际上一个请求的 postponed 链表里面除了保存子请求的节点，还有保存该请求自己产生的数据的节点，数据保存在 out 字段；第二个是 posted_requests 链表，它挂载了当前需要遍历的请求（节点），该链表保存在主请求（根节点）的 posted_requests 字段，链表节点定义如下：

```

struct ngx_http_posted_request_s {
    ngx_http_request_t      *request;
    ngx_http_posted_request_t *next;
};

```

在 ngx_http_run_posted_requests 函数中会顺序的遍历主请求的 posted_requests 链表：

```

void
ngx_http_run_posted_requests(ngx_connection_t *c)
{
    ...
    for ( ;; ) {
        /* 连接已经断开，直接返回 */
        if (c->destroyed) {
            return;
        }

        r = c->data;
        /* 从 posted_requests 链表的队头开始遍历 */
        pr = r->main->posted_requests;

        if (pr == NULL) {
            return;
        }

        /* 从链表中移除即将要遍历的节点 */
        r->main->posted_requests = pr->next;
        /* 得到该节点中保存的请求 */
        r = pr->request;

        ctx = c->log->data;
        ctx->current_request = r;
    }
}

```

```

    ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
                  "http posted request: \"%V?%V\"", &r->uri, &r->args);
    /* 遍历该节点（请求） */
    r->write_event_handler(r);
}
}

```

ngx_http_run_posted_requests 函数的调用点后面会做说明。

了解了一些实现的原理，来看代码就简单多了，现在正式进行 subrequest 的源码解析，首先来看一下创建 subrequest 的函数定义：

```

ngx_int_t
ngx_http_subrequest(ngx_http_request_t *r,
    ngx_str_t *uri, ngx_str_t *args, ngx_http_request_t **psr,
    ngx_http_post_subrequest_t *ps, ngx_uint_t flags)

```

参数 r 为当前的请求，uri 和 args 为新的要发起的 uri 和 args，当然 args 可以为 NULL，psr 为指向一个 ngx_http_request_t 指针的指针，它的作用就是获得创建的子请求，ps 的类型为 ngx_http_post_subrequest_t，它的定义如下：

```

typedef struct {
    ngx_http_post_subrequest_pt    handler;
    void                            *data;
} ngx_http_post_subrequest_t;

typedef ngx_int_t (*ngx_http_post_subrequest_pt)(ngx_http_request_t *r,
    void *data, ngx_int_t rc);

```

它就是之前说到的回调 handler，结构里面的 handler 类型为 ngx_http_post_subrequest_pt，它是函数指针，data 为传递给 handler 的额外参数。再来看一下 ngx_http_subrequest 函数的最后一个参数是 flags，现在的源码中实际上只有 2 种类型的 flag，分别为 NGX_HTTP_SUBREQUEST_IN_MEMORY 和 NGX_HTTP_SUBREQUEST_WAITED，第一个就是指定文章开头说到的子请求的 upstream 处理数据的方式，第二个参数表示如果该子请求提前完成(按后序遍历的顺序)，是否设置将它的状态设为 done，当

设置该参数时，提前完成就会设置 done，不设时，会让该子请求等待它之前的子请求处理完毕才会将状态设置为 done。

进入 ngx_http_subrequest 函数内部看看：

```
{
    ...
    /* 解析 flags， subrequest_in_memory 在 upstream 模块解析完头部，
       发送 body 给 downstream 时用到 */
    sr->subrequest_in_memory = (flags & NGX_HTTP_SUBREQUEST_IN_MEMORY) != 0;
    sr->waited = (flags & NGX_HTTP_SUBREQUEST_WAITED) != 0;

    sr->unparsed_uri = r->unparsed_uri;
    sr->method_name = ngx_http_core_get_method;
    sr->http_protocol = r->http_protocol;

    ngx_http_set_exten(sr);
    /* 主请求保存在 main 字段中 */
    sr->main = r->main;
    /* 父请求为当前请求 */
    sr->parent = r;
    /* 保存回调 handler 及数据，在子请求执行完，将会调用 */
    sr->post_subrequest = ps;
    /* 读事件 handler 赋值为不做任何事的函数，因为子请求不用再读数据或者检查连接状态；
       写事件 handler 为 ngx_http_handler，它会重走 phase */
    sr->read_event_handler = ngx_http_request_empty_handler;
    sr->write_event_handler = ngx_http_handler;

    /* ngx_connection_s 的 data 字段比较关键，它保存了当前可以向 out chain 输出数据的请
       求，
       具体意义后面会做详细介绍 */
    if (c->data == r && r->postponed == NULL) {
        c->data = sr;
    }
    /* 默认共享父请求的变量，当然你也可以根据需求在创建完子请求后，再创建子请求独立的变
       量集 */
    sr->variables = r->variables;

    sr->log_handler = r->log_handler;

    pr = ngx_palloc(r->pool, sizeof(ngx_http_postponed_request_t));
    if (pr == NULL) {
        return NGX_ERROR;
    }

    pr->request = sr;
```

```

pr->out = NULL;
pr->next = NULL;
/* 把该子请求挂载在其父请求的 postponed 链表的队尾 */
if (r->postponed) {
    for (p = r->postponed; p->next; p = p->next) { /* void */ }
    p->next = pr;
} else {
    r->postponed = pr;
}
/* 子请求为内部请求，它可以访问 internal 类型的 location */
sr->internal = 1;
/* 继承父请求的一些状态 */
sr->discard_body = r->discard_body;
sr->expect_tested = 1;
sr->main_filter_need_in_memory = r->main_filter_need_in_memory;

sr->uri_changes = NGX_HTTP_MAX_URI_CHANGES + 1;

tp = ngx_timeofday();
r->start_sec = tp->sec;
r->start_msec = tp->msec;

r->main->subrequests++;
/* 增加主请求的引用数，这个字段主要是在 ngx_http_finalize_request 调用的一些结束请求和
连接的函数中使用 */
r->main->count++;

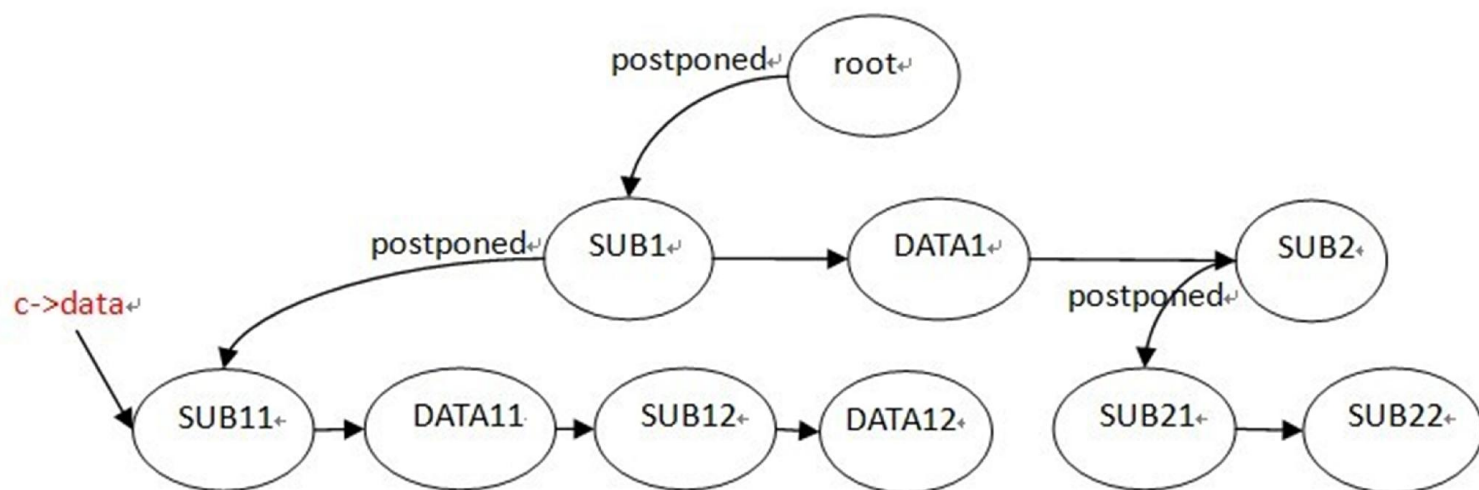
*psr = sr;
/* 将该子请求挂载在主请求的 posted_requests 链表队尾 */
return ngx_http_post_request(sr, NULL);
}

```

到这时，子请求创建完毕，一般来说子请求的创建都发生在某个请求的 content handler 或者某个 filter 内，从上面的函数可以看到子请求并没有马上被执行，只是被挂载在了主请求的 posted_requests 链表中，那它什么时候可以执行呢？之前说到 posted_requests 链表是在 ngx_http_run_posted_requests 函数中遍历，那么 ngx_http_run_posted_requests 函数又是在什么时候调用？它实际上是在某个请求的读（写）事件的 handler 中，执行完该请求相关的处理后被调用，比如主请求在走完一遍 PHASE 的时候会调用 ngx_http_run_posted_requests，这时子请求得以运行。

这时实际还有 1 个问题需要解决，由于 nginx 是多进程，是不能够随意阻塞的（如果一个请求阻塞了当前进程，就相当于阻塞了这个进程 accept 到的所有其他请求，同时该进程也不能 accept 新请求），一个请求可能由于某些原因需要阻塞（比如访问 io），nginx 的做法是设置该请求的一些状态并在 epoll 中添加相应的事件，然后转去处理其他请求，等到该事件到来时再继续处理该请求，这样的行为就意味着一个请求可能需要多次执行机会才能完成，对于一个请求的多个子请求来说，意味着它们完成的先后顺序可能和它们创建的顺序是不一样的，所以必须有一种机制让提前完成的子请求保存它产生的数据，而不是直接输出到 out chain，同时也能够让当前能够往 out chain 输出数据的请求及时的输出产生的数据。作者 Igor 采用 ngx_connection_t 中的 data 字段，以及一个 body filter，即 ngx_http_postpone_filter，还有 ngx_http_finalize_request 函数中的一些逻辑来解决这个问题。

下面用一个图来做说明，下图是某时刻某个主请求和它的所有子孙请求的树结构：



图中的 root 节点即为主请求，它的 postponed 链表从左至右挂载了 3 个节点，SUB1 是它的第一个子请求，DATA1 是它产生的一段数据，SUB2 是它的第 2 个子请求，而且这 2 个子请求分别有它们自己的子请求及数据。ngx_connection_t 中的 data 字段保存的是当前可以往 out chain 发送数据的请求，文章开头说到发到客户端的数据必须按照子请求创建的顺序发送，这里即是按后序遍历的方法（SUB11->DATA11->SUB12->DATA12->(SUB1)->DATA1->SUB21->SUB22->(SUB2)->(ROOT)），上图中当前能够往客户端（out chain）发送数据的请求显然就是 SUB11，如果 SUB12 提前执行完成，并产生数据 DATA121，只要前面它还有节点未发送完毕，DATA121 只能先挂载在 SUB12 的 postponed 链表下。这里还要注意一下的是 c->data 的设置，当 SUB11 执行完并且发送完数据之后，下一个将要

发送的节点应该是 **DATA11**，但是该节点实际上保存的是数据，而不是子请求，所以 **c->data** 这时应该指向的是拥有改数据节点的 **SUB1** 请求。

下面看下源码具体是怎样实现的，首先是 ngx_http_postpone_filter 函数：

[illegible]

```

        &pr->request->uri, &pr->request->args);

    r->postponed = pr->next;

    /* 按照后序遍历产生的序列，因为当前请求（节点）有未处理的子请求（节点），
       必须先处理完改子请求，才能继续处理后面的子节点。
       这里将该子请求设置为可以往 out chain 发送数据的请求。 */
    c->data = pr->request;
    /* 将该子请求加入主请求的 posted_requests 链表 */
    return ngx_http_post_request(pr->request, NULL);
}
/* 如果该节点保存的是数据，可以直接处理该节点，将它发送到 out chain */
if (pr->out == NULL) {
    ngx_log_error(NGX_LOG_ALERT, c->log, 0,
                  "http postpone filter NULL output",
                  &r->uri, &r->args);

} else {
    ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
                  "http postpone filter output \"%V?%V\"",
                  &r->uri, &r->args);

    if (ngx_http_next_filter(r->main, pr->out) == NGX_ERROR) {
        return NGX_ERROR;
    }
}

r->postponed = pr->next;

} while (r->postponed);

return NGX_OK;
}

```

再来看 ngx_http_finalize_request 函数：

```

void
ngx_http_finalize_request(ngx_http_request_t *r, ngx_int_t rc)
{
    ...
    /* 如果当前请求是一个子请求，检查它是否有回调 handler，有的话执行之 */
    if (r != r->main && r->post_subrequest) {
        rc = r->post_subrequest->handler(r, r->post_subrequest->data, rc);
    }

    ...

    /* 子请求 */
}

```

```

if (r != r->main) {
    /* 该子请求还有未处理完的数据或者子请求 */
    if (r->buffered || r->postponed) {
        /* 添加一个该子请求的写事件，并设置合适的 write event handler，
           以便下次写事件来的时候继续处理，这里实际上下次执行时会调用
           ngx_http_output_filter 函数，
           最终还是会进入 ngx_http_postpone_filter 进行处理 */
        if (ngx_http_set_write_handler(r) != NGX_OK) {
            ngx_http_terminate_request(r, 0);
        }

        return;
    }
    ...

    pr = r->parent;

    /* 该子请求已经处理完毕，如果它拥有发送数据的权利，则将权利移交给父请求， */
    if (r == c->data) {

        r->main->count--;

        if (!r->logged) {

            clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

            if (clcf->log_subrequest) {
                ngx_http_log_request(r);
            }

            r->logged = 1;

        } else {
            ngx_log_error(NGX_LOG_ALERT, c->log, 0,
                "subrequest: \"%V?%V\" logged again",
                &r->uri, &r->args);
        }

        r->done = 1;
        /* 如果该子请求不是提前完成，则从父请求的 postponed 链表中删除 */
        if (pr->postponed && pr->postponed->request == r) {
            pr->postponed = pr->postponed->next;
        }
        /* 将发送权利移交给父请求，父请求下次执行的时候会发送它的 postponed 链表中可
           以
           发送的数据节点，或者将发送权利移交给它的下一个子请求 */
        c->data = pr;

```

```

    } else {
        /* 到这里其实表明该子请求提前执行完成，而且它没有产生任何数据，则它下次再次
获得
        执行机会时，将会执行 ngx_http_request_finalzier 函数，它实际上是执行
        ngx_http_finalize_request (r,0)，也就是什么都不干，直到轮到它发送数据
        时，
        ngx_http_finalize_request 函数会将它从父请求的 postponed 链表中删除 */
        r->write_event_handler = ngx_http_request_finalizer;

        if (r->waited) {
            r->done = 1;
        }
    }
    /* 将父请求加入 posted_request 队尾，获得一次运行机会 */
    if (ngx_http_post_request(pr, NULL) != NGX_OK) {
        r->main->count++;
        ngx_http_terminate_request(r, 0);
        return;
    }

    return;
}
/* 这里是处理主请求结束的逻辑，如果主请求有未发送的数据或者未处理的子请求，
则给主请求添加写事件，并设置合适的 write event handler，
以便下次写事件来的时候继续处理 */
if (r->buffered || c->buffered || r->postponed || r->blocked) {

    if (ngx_http_set_write_handler(r) != NGX_OK) {
        ngx_http_terminate_request(r, 0);
    }

    return;
}

...
}

```

https 请求处理解析

nginx 支持 ssl 简介

nginx-1.2.0 编译时默认是不支持 ssl 协议的，需要通过编译指令来开启对其支持：

```
./configure --with-http_ssl_module
```

在 nginx 源码中，ssl 相关代码用宏定义变量 `NGX_HTTP_SSL` 来控制是否开启。这给我们查找和阅读 ssl 相关代码带来了方便，如下：

ssl 协议工作在 tcp 协议与 http 协议之间。nginx 在支持 ssl 协议时，需要注意三点，其他时候只要正常处理 http 协议即可：

1. tcp 连接建立时，在 tcp 连接上建立 ssl 连接
2. tcp 数据接收后，将收到的数据解密并将解密后的数据交由正常 http 协议处理流程
3. tcp 数据发送前，对(http)数据进行加密，然后再发送

以下章节将分别介绍这三点。

ssl 连接建立(ssl 握手)

对 ssl 连接建立的准备

更具 ssl 协议规定，在正式发起数据收发前，需要建立 ssl 连接，连接建立过程既 ssl 握手。nginx 在创建和初始化 http 请求阶段的同时为 tcp 连接建立做准备，主要流程在 `ngx_http_init_request` 函数中实现：

```
static void
ngx_http_init_request(ngx_event_t *rev)
{
    ...

    #if (NGX_HTTP_SSL)

        {
            ngx_http_ssl_srv_conf_t *sscf;

            sscf = ngx_http_get_module_srv_conf(r, ngx_http_ssl_module);
            if (sscf->enable || addr_conf->ssl) {

                /* c->ssl 不为空时，表示请求复用长连接(已经建立过 ssl 连接) */
                if (c->ssl == NULL) {

                    c->log->action = "SSL handshaking";

                    /*
                     * nginx.conf 中开启 ssl 协议(listen 443 ssl);,

```



```

        * 却没用设置服务器证书(ssl_certificate <certificate_path>;)
        */
    if (addr_conf->ssl && sscf->ssl.ctx == NULL) {
        ngx_log_error(NGX_LOG_ERR, c->log, 0,
                      "no \"ssl_certificate\" is defined "
                      "in server listening on SSL port");
        ngx_http_close_connection(c);
        return;
    }

    /*
     * 创建 ngx_ssl_connection_t 并初始化
     * openssl 库中关于 ssl 连接的初始化
     */
    if (ngx_ssl_create_connection(&sscf->ssl, c, NGX_SSL_BUFFER)
        != NGX_OK)
    {
        ngx_http_close_connection(c);
        return;
    }

    rev->handler = ngx_http_ssl_handshake;
}

/* ssl 加密的数据必须读到内存中 */
r->main_filter_need_in_memory = 1;
}
}

#endif

...

}

```

ngx_http_init_request 大部分流程已经在前面章节分析过了，这个函数主要负责初始化 http 请求，此时并没有实际解析 http 请求。若发来的请求是经由 ssl 协议加密的，直接解析 http 请求就会出错。

ngx_http_init_request 中 ssl 协议相关处理流程:

- 1, 首先判断 c->ssl 是否为空。若不为空：说明这里是 http 长连接的情况，ssl 连接已经在第一个请求进入时建立了。这里只要复用这个 ssl 连接即可，跳过 ssl 握手阶段。

2.(1), 若 `c->ssl` 为空: 需要进行 ssl 握手来建立连接。此时调用 `ngx_ssl_create_connection` 为 ssl 连接建立做准备。

`ngx_ssl_create_connection` 简化代码如下:

```
ngx_int_t
ngx_ssl_create_connection(ngx_ssl_t *ssl, ngx_connection_t *c, ngx_uint_t flags)
{
    ngx_ssl_connection_t *sc;

    /* ngx_ssl_connection_t 是 nginx 对 ssl 连接的描述结构, 记录了 ssl 连接的信息和状态 */
    sc = ngx_palloc(c->pool, sizeof(ngx_ssl_connection_t));

    sc->buffer = ((flags & NGX_SSL_BUFFER) != 0);

    /* 创建 openssl 库中对 ssl 连接的描述结构 */
    sc->connection = SSL_new(ssl->ctx);

    /* 关联(openssl 库)ssl 连接到 tcp 连接对应的 socket */
    SSL_set_fd(sc->connection, c->fd);

    if (flags & NGX_SSL_CLIENT) {
        /* upstream 中发起对后端的 ssl 连接, 指明 nginx ssl 连接是客户端 */
        SSL_set_connect_state(sc->connection);
    } else {
        /* 指明 nginx ssl 连接是服务端 */
        SSL_set_accept_state(sc->connection);
    }

    /* 关联(openssl 库)ssl 连接到用户数据(当前连接 c) */
    SSL_set_ex_data(sc->connection, ngx_ssl_connection_index, c);

    c->ssl = sc;

    return NGX_OK;
}
```

2.(2), 设置连接读事件处理函数为 `ngx_http_ssl_handshake`, 这将改变后续处理 http 请求的正常流程为: 先进行 ssl 握手, 再正常处理 http 请求。

3, 标明当前待发送的数据须在内存中, 以此可以让 ssl 对数据进行加密。由于开启了 ssl 协议, 对发送出去的数据要进行加密, 这就要求待发送的数据必须在内存中。标识 `r->main_filter_need_in_memory`

为1,可以让后续数据发送前,将数据读取到内存中(防止在文件中的数据通过 sendfile 直接发送出去,而没有加密)。

实际 ssl 握手阶段

由于在 ngx_http_init_request 中将连接读事件处理函数设置成 ngx_http_ssl_handshake, 当连接中有可读数据时, 将会进入 ngx_http_ssl_handshake 来处理(若未开启 ssl, 将进入 ngx_http_process_request_line 直接解析 http 请求)

在 ngx_http_ssl_handshake 中, 来进行 ssl 握手:

1, 首先判断连接是否超时, 如果超时则关闭连接

```
static void
ngx_http_process_request(ngx_http_request_t *r)
{
    if (rev->timedout) {
        ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT, "client timed out");
        c->timedout = 1;
        ngx_http_close_request(r, NGX_HTTP_REQUEST_TIME_OUT);
        return;
    }
}
```

2, 首字节预读: 从 tcp 连接中查看一个字节(通过 MSG_PEEK 查看 tcp 连接中数据, 但不会实际读取该数据), 若 tcp 连接中没有准备好的数据, 则重新添加读事件退出等待新数据到来。

```
n = recv(c->fd, (char *) buf, 1, MSG_PEEK);

if (n == -1 && ngx_socket_errno == NGX_EAGAIN) {

    if (!rev->timer_set) {
        ngx_add_timer(rev, c->listening->post_accept_timeout);
    }

    if (ngx_handle_read_event(rev, 0) != NGX_OK) {
        ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
    }

    return;
}
```

3, 首字节探测: 若成功查看 1 个字节数据, 通过该首字节来探测接受到的数据是 ssl 握手包还是 http 数据。根据 ssl 协议规定, ssl 握手包的首字节中包含有 ssl 协议的版本信息。nginx 根据此来判断是进行 ssl 握手还是返回正常处理 http 请求(实际返回应答 400 BAD REQUEST)。

```
if (n == 1) {
    if (buf[0] & 0x80 /* SSLv2 */ || buf[0] == 0x16 /* SSLv3/TLSv1 */) {
        ngx_log_debug1(NGX_LOG_DEBUG_HTTP, rev->log, 0,
            "https ssl handshake: 0x%02Xd", buf[0]);

        /*
         * 调用 ngx_ssl_handshake 函数进行 ssl 握手, 连接双方会在 ssl 握手时交换相
         * 关数据(ssl 版本, ssl 加密算法, server 端的公钥等) 并正式建立起 ssl 连接。
         * ngx_ssl_handshake 函数内部对 openssl 库进行了封装。
         * 调用 SSL_do_handshake()来进行握手, 并根据其返回值判断 ssl 握手是否完成
         * 或者出错。
         */
        rc = ngx_ssl_handshake(c);

        /*
         * ssl 握手可能需要多次数据交互才能完成。
         * 如果 ssl 握手没有完成, ngx_ssl_handshake 会根据具体情况(如需要读取更
         * 多的握手数据包, 或者需要发送握手数据包) 来重新添加读写事件
         */
        if (rc == NGX_AGAIN) {

            if (!rev->timer_set) {
                ngx_add_timer(rev, c->listening->post_accept_timeout);
            }

            c->ssl->handler = ngx_http_ssl_handshake_handler;
            return;
        }

        /*
         * 若 ssl 握手完成或者出错, ngx_ssl_handshake 会返回 NGX_OK 或者 NGX_ERROR, 然后
         * ngx_http_ssl_handshake 调用
         * ngx_http_ssl_handshake_handler 以继续处理
         */

        ngx_http_ssl_handshake_handler(c);

        return;
    } else {
        ngx_log_debug0(NGX_LOG_DEBUG_HTTP, rev->log, 0,
            "plain http");
    }
}
```

```

        r->plain_http = 1;
    }
}

```

需要特别注意，如果 ssl 握手完成，ngx_ssl_handshake 会替换连接的读写接口。这样，后续需要读写数据时，替换的接口会对数据进行加密解密。详细代码见下：

```

ngx_int_t
ngx_ssl_handshake(ngx_connection_t *c)
{
    n = SSL_do_handshake(c->ssl->connection);
    /* 返回 1 表示 ssl 握手成功 */
    if (n == 1) {
        ...
        c->ssl->handshaked = 1;

        c->recv = ngx_ssl_recv;
        c->send = ngx_ssl_write;
        c->recv_chain = ngx_ssl_recv_chain;
        c->send_chain = ngx_ssl_send_chain;

        return NGX_OK;
    }
    ...
}

```

4, 探测为 http 协议: 正常的 http 协议包处理直接调用 ngx_http_process_request_line 处理 http 请求，并将读事件处理函数设置成 ngx_http_process_request_line。(实际处理结果是向客户端返回 400 BAD REQUEST，在 ngx_http_process_request 中又对 r->plain_http 标志的单独处理。)

```

    c->log->action = "reading client request line";

    rev->handler = ngx_http_process_request_line;
    ngx_http_process_request_line(rev);

} /* end of ngx_http_process_request() */

```

5, 当 ssl 握手成功或者出错时，调用 ngx_http_ssl_handshake_handler 函数。

5.(1), 若 ssl 握手完成 (c->ssl->handshaked 由 ngx_ssl_handshake() 确定握手完成后设为 1), 设置读事件处理函数为 ngx_http_process_request_line, 并调用此函数正常处理 http 请求。

5.(2), 若 ssl 握手没完成 (则说明 ssl 握手出错), 则返回 400 BAD REQUEST 给客户端。

至此, ssl 连接已经建立, 此后在 ngx_http_process_request 中会读取数据并解密然后正常处理 http 请求。

```
static void
ngx_http_ssl_handshake_handler(ngx_connection_t *c)
{
    ngx_http_request_t  *r;

    if (c->ssl->handshaked) {

        /*
         * The majority of browsers do not send the "close notify" alert.
         * Among them are MSIE, old Mozilla, Netscape 4, Konqueror,
         * and Links.  And what is more, MSIE ignores the server's alert.
         *
         * Opera and recent Mozilla send the alert.
         */

        c->ssl->no_wait_shutdown = 1;

        c->log->action = "reading client request line";

        c->read->handler = ngx_http_process_request_line;
        /* STUB: epoll edge */ c->write->handler = ngx_http_empty_handler;

        ngx_http_process_request_line(c->read);

        return;
    }

    r = c->data;

    ngx_http_close_request(r, NGX_HTTP_BAD_REQUEST);

    return;
}
```

ssl 协议接受数据

ngx_http_process_request 中处理 http 请求，需要读取和解析 http 协议。而实际数据读取是通过 c->recv() 函数来读取的，此函数已经在 ngx_ssl_handshake 中被替换成 ngx_ssl_recv 了。

ngx_ssl_recv 函数中调用 openssl 库函数 SSL_read() 来读取并解密数据，简化后如下：

```
ssize_t ngx_ssl_recv(ngx_connection_t *c, u_char *buf, size_t size)
{
    ...
    n = SSL_read(c->ssl->connection, buf, size);
    ...
    return n;
}
```

ssl 协议发送数据

当 nginx 发送数据时，如使用 ngx_output_chain 函数发送缓存的 http 数据缓存链时，通过调用 c->send_chain() 来发送数据。这个函数已经在 ngx_ssl_handshake 中被设置成 ngx_ssl_send_chain 了。ngx_ssl_send_chain 会进一步调用 ngx_ssl_write。而 ngx_ssl_write 调用 openssl 库 SSL_write 函数来加密并发送数据。

```
/* ngx_output_chain
 * -> ..
 * -> ngx_chain_writer
 * -> c->send_chain (ngx_ssl_send_chain)
 * -> ngx_ssl_write
 */
ssize_t ngx_ssl_write(ngx_connection_t *c, u_char *data, size_t size)
{
    ...
    n = SSL_write(c->ssl->connection, data, size);
    ...
    return n;
}
```

nginx 的 upstream 原理解析

无缓冲的处理流程

带缓冲的处理流程

subrequest 访问 **upstream** 的处理流程

负载均衡算法

round-robin 算法

基于 **ip** 的 **hash** 算法

upstream 缓存机制

常用 **upstream** 模块分析

proxy 模块

fastcgi 模块

nginx 的 **script** 处理

正则表达式

变量

复杂变量

if 的处理

附录 A 编码风格 (100%)

Nginx 代码风格图示 (100%)

一、基本原则

K&R 编码风格（偏 BSD 子类）。

每行不能超过 80 列。

不用 TAB 对齐，用空格。

默认对齐单元是 4 个空格。

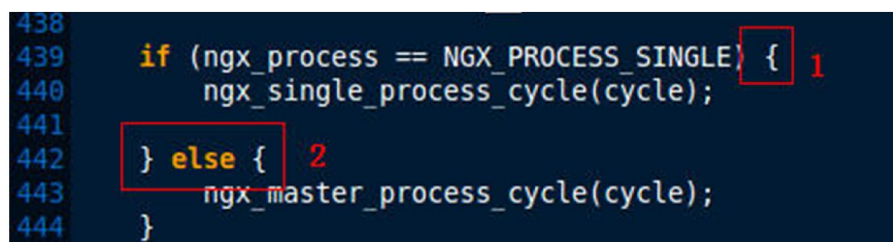
除宏定义外，字母均为小写，单词间用下划线_间隔。

使用 C 方式的注释，不得使用//形式注释。

中缀运算符的前后须空一格，如 $3 + 2$ 以及 $a > 3$ 。

逗号后须空一格，如 `foo(a, b, c);`

二、风格图示



```
438
439     if (ngx_process == NGX_PROCESS_SINGLE) { 1
440         ngx_single_process_cycle(cycle);
441     } else { 2
442         ngx_master_process_cycle(cycle);
443     }
```

The image shows a code snippet from a dark-themed editor. It displays an `if/else` statement. Red boxes highlight the opening curly brace of the `if` block and the `else` keyword. Red numbers '1' and '2' are placed next to these elements, corresponding to the numbered list below. The code is as follows:

1、 `if/while/for/switch` 语句的左花括号和关键字在同一行上，和括号之间空一个空格。

2、 `else` 关键字和两个花括号在同一行上。

```

1  3
2  /*
3   * Copyright (C) Igor Sysoev
4   */
5
6  4
7  #include <ngx_config.h>
8  #include <ngx_core.h>
9  #include <ngx.h>
10
11
12 static ngx_int_t ngx_add_inherited_sockets(ngx_cycle_t *cycle);
13 static ngx_int_t ngx_get_options(int argc, char *const *argv);
14 static ngx_int_t ngx_process_options(ngx_cycle_t *cycle);
15 static ngx_int_t ngx_save_argv(ngx_cycle_t *cycle, int argc, char *const *argv);
16 static void *ngx_core_module_create_conf(ngx_cycle_t *cycle);
17 static char *ngx_core_module_init_conf(ngx_cycle_t *cycle, void *conf);
18 static char *ngx_set_user(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
19 static char *ngx_set_env(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
20 static char *ngx_set_priority(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
21 static char *ngx_set_worker_processes(ngx_conf_t *cf, ngx_command_t *cmd,
22     void *conf);
23 static char *ngx_set_cpu_affinity(ngx_conf_t *cf, ngx_command_t *cmd,
24     void *conf);
25 5
26
27 static ngx_conf_enum_t ngx_debug_points[] = {
28 6 { ngx_string("stop"), NGX_DEBUG_POINTS_STOP },
29   { ngx_string("abort"), NGX_DEBUG_POINTS_ABORT },
30   { ngx_null_string, 0 }
31 };
32
33

```

- 3、 文件开始的注释空一行。
- 4、 较为完整的代码块间的距离为空两行。如函数声明、函数定义之间等。
- 5、 函数声明或定义若一行显示不下，则函数原型空 4 个空格。
- 6、 结构体数组的花括号和内容之间空一个空格。

```

128 static ngx_command_t  ngx_epoll_commands[] = {
129     8
130     { ngx_string("epoll_events"),
131       NGX_EVENT_CONF|NGX_CONF_TAKE1,
132       ngx_conf_set_num_slot,
133       0,
134       offsetof(ngx_epoll_conf_t, events),
135       NULL },
136
137     { ngx_string("worker_aio_requests"),
138       NGX_EVENT_CONF|NGX_CONF_TAKE1,
139       ngx_conf_set_num_slot,
140       0,
141       offsetof(ngx_epoll_conf_t, aio_requests),
142       NULL },
143     9
144     ngx_null_command
145 };

```

7、 结构体数组的左花括号放在同一行上。

8、 较大的结构体数组元素最开始空一行。

9、 元素内容上下对齐。

```

167 ngx_module_t  ngx_epoll_module = {
168     NGX_MODULE_V1,
169     &ngx_epoll_module_ctx,
170     ngx_epoll_commands,
171     NGX_EVENT_MODULE,
172     NULL,
173     NULL,
174     NULL,
175     NULL,
176     NULL,
177     NULL,
178     NULL,
179     NGX_MODULE_V1_PADDING
180 };

```

10

```

/* module context */
/* module directives */
/* module type */
/* init master */
/* init module */
/* init process */
/* init thread */
/* exit thread */
/* exit process */
/* exit master */

```

10、 注释上下对齐。

```

417     ngx_log_debug3(NGX_LOG_DEBUG_EVENT, ev->log, 0,
418         11 "epoll add event: fd:%d op:%d ev:%08XD",
419         c->fd, op, ee.events);
420

```

11、 函数调用折行时，参数上下对齐。

```

436 static ngx_int_t 12
437 ngx_epoll_del_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
438 {
439     int
440     uint32_t
441 13 ngx_event_t *e; 14
442     ngx_connection_t *c;
443     struct epoll_event ee;
444

```

12、函数定义时，类型单独一行。

13、变量声明的类型上下排列按照从短到长的顺序。注意，最下面的变量的类型和名称间的空格为 2-3 个。一般情况下为 2 个，这是 Nginx 中最小的变量声明中类型和名称的距离。

14、变量名称上下对齐——字母对齐，不包括指针的*号。

```

49 typedef struct {
50     ngx_msec_t
51     ngx_uint_t
52
53     ngx_uint_t
54     time_t
55     ngx_uint_t
56     off_t
57
58     ngx_str_t
59 } ngx_http_upstream_state_t;

```

15

```

    bl_time;
    bl_state;

    status;
    response_sec;
    response_msec;
    response_length;

    *peer;

```

15、结构体内变量上下对齐（字母，不包括指针的*号）。

```

768 default:
769
770 16 /* cached NGX_HTTP_BAD_GATEWAY, NGX_HTTP_GATEWAY_TIME_OUT, etc. */
771
772     u->cache_status = NGX_HTTP_CACHE_HIT;
773
774     return rc;
775 }

```

16、单行注释格式为/* something */

```

1163 {
1164
1165 17 /*
1166     * the r->request_body->buf can be reused for one request only,
1167     * the subrequests should allocate their own temporary bufs
1168     */
1169
1170     u->output.free = ngx_alloc_chain_link(r->pool);

```


17、多行注释的格式为：

```
/*
 * something
 */
```

```
249 size_t
250 ngx_inet_ntop(int family, void *addr, u_char *text, size_t len)
251 {
252     u_char *p;
253
254     switch (family) {
255 19 case AF_INET:
256
257         p = addr;
258
259         return ngx_snprintf(text, len, "%ud.%ud.%ud.%ud",
260                               p[0], p[1], p[2], p[3])
261             - text;
262     }
```

18、函数定义的左花括号独占一行。

19、switch 语句中，switch 和 case 关键字上下对齐。

```
1176 if (lock_file.len) {
1177     lock_file.len--;
1178
1179     if (ccf->lock_file.len != lock_file.len
1180         || ngx_strncmp(ccf->lock_file.data, lock_file.data, lock_file.len)
1181             != 0)
1182 20 {
1183         ngx_log_error(NGX_LOG_EMERG, cycle->log, 0,
1184                       "\\lock_file\\ could not be changed, ignored");
1185     }
1186 }
```

20、当条件表达式过长需要折行时，关系运算符须位于下一行的行首，并与上一行的条件表达式的第一个字符对齐，同时右花括号须位于单独的一行，并与 if/while 等关键字对齐。

```
if (shift) {
    cidr->u.in.mask = htonl((ngx_uint_t) (0 - (1 << (32 - shift))));
    21
} else {
    /* x86 compilers use a shl instruction that shifts by modulo 32 */
    cidr->u.in.mask = 0;
}
```

21、 else 语句之前须空出一行。

```
613 static ngx_int_t
614 ngx_parse_inet_url(ngx_pool_t *pool, ngx_url_t *u)
615 {
616     u_char          *p, *host, *port, *last, *uri, *args;
617     size_t          len;
618     ngx_int_t        n;
619     struct hostent   *h;
620     struct sockaddr_in *sin;
```

22、在函数中，相同类型的变量声明放在一行上。

附录 B 常用 API

B.1 字符串操作

B.2 读取请求体

B.3 操作请求头

B.4 取参数

B.5 URI 处理

B.6 buffer 操作

B.7 变量操作

B.8 日志输出

B.9 配置解析接口

B.10 文件操作

B.11 锁操作

附录 C 模块编译，调试与测试

C.1 编译与安装

环境要求

操作系统：目前 Nginx 各版本在以下操作系统和平台测试通过：

FreeBSD 3 — 10 / i386; FreeBSD 5 — 10 / amd64;

Linux 2.2 — 3 / i386; Linux 2.6 — 3 / amd64;

Solaris 9 / i386, sun4u; Solaris 10 / i386, amd64, sun4v;

AIX 7.1 / powerpc;

HP-UX 11.31 / ia64;

MacOS X / ppc, i386;

Windows XP, Windows Server 2003

磁盘空间：必须保证至少 10M 以上的磁盘工具，并且随着编译设置及第三方模块的安装而有所不同；

编译器及相关工具：必须确保操作系统安装有 GCC 编译器；make 工具；用户可通过 yum 命令安装编译器及相关工具：yum -y install gcc gcc-c++ make;

模块依赖性：Nginx 的一些模块需要第三方库的支持，如 rewrite 模块需要 pcre 库，gzip 模块需要 zlib 模块，ssl 功能你需要 openssl 库等。用户可通过 yum 命令安装这些依赖库：yum -y install pcre pcre-devel zlib zlib-devel openssl openssl-devel;

下载

Nginx 是开源软件,用户可以访问 <http://nginx.org/> 网站获取源码包或 Windows 二进制文件下载。

其中 1.3.x 版本为开发版本,1.2.x 版本为稳定版本。开发版本分支会较快的获得新功能和缺陷修复,但同时也可能会遇到新的缺陷。一旦更新稳定下来,就会被加入稳定版本分支。

作为生产环境,通常建议用户使用稳定版本。

Nginx 在 Windows 环境下安装

nginx 的 windows 版本使用原生 win32 API (非 Cygwin 模拟层)。当前存在的已知问题: 1.采用 select 作为通知方法,所以不具备很高的性能和扩展性; 2.虽然可以启动若干工作进程运行,实际上只有一个进程在处理请求所有请求; 3.一个工作进程只能处理不超过 1024 个并发连接; 4.缓存和其他需要共享内存支持的模块在 windows vista 及后续版本的操作系统中无法工作,因为在这些操作系统中,地址空间的布局是随机的; 5.除了 XSLT 过滤器、图像过滤器、GeoIP 模块和嵌入 Perl 语言支持以外, Nginx 的 Windows 版本与 Unix 版本相比,功能几乎齐全。

安装 Nginx 的 Windows 版本,建议下载最新的 1.3.13 开发版本,因为开发分支上包含了所有已知的问题修复,尤其是针对 Windows 版本的问题修复。解压下载得到的 zip 文件,进入 nginx-1.3.13 目录,运行 nginx。

C 盘根目录下安装例子

```
cd c:\
unzip nginx-1.3.13.zip
cd nginx-1.3.13
start nginx
```

Nginx 的 Windows 版本的控制命令包含如下:

```
nginx -s stop 快速退出
nginx -s quit 优雅退出
nginx -s reload 更换配置,启动新的工作进程,优雅的关闭以往的工作进程
nginx -s reopen 重新打开日志文件
```

Nginx 在 Linux 环境下安装

Nginx 在 Linux 环境下可以通过编译源码的方式安装,最简单的安装命令如下:


```
wget http://nginx.org/download/nginx-1.2.0.tar.gz
tar zxvf nginx-1.2.0.tar.gz
cd nginx-1.2.0
./configure
make
sudo make install
```

按照以上命令，Nginx 将被默认安装到/usr/local/nginx 目录下。用户可以通过./configure --help 命令查看 Nginx 可选择的编译选项进行自定义安装配置。

Nginx 的 configure 脚本支持以下选项:

```
--prefix=<PATH> #Nginx 安装路径。如果没有指定，默认为 /usr/local/nginx

--sbin-path=<PATH> #Nginx 可执行文件安装路径。只能安装时指定，如果没有指定，默认为
<prefix>/sbin/nginx

--conf-path=<PATH> #在没有给定 -c 选项下默认的 nginx.conf 的路径。如果没有指定，默认为
<prefix>/conf/nginx.conf

--pid-path=<PATH> #在 nginx.conf 中没有指定 pid 指令的情况下，默认的 nginx.pid 的路径。
如果没有指定，默认为 <prefix>/logs/nginx.pid

--lock-path=<PATH> #nginx.lock 文件的路径

--error-log-path=<PATH> #在 nginx.conf 中没有指定 error_log 指令的情况下，默认的错误日
志的路径。如果没有指定，默认为 <prefix>/logs/error.log

--http-log-path=<PATH> #在 nginx.conf 中没有指定 access_log 指令的情况下，默认的访问日
志的路径。如果没有指定，默认为 <prefix>/logs/access.log。

--user=<USER> #在 nginx.conf 中没有指定 user 指令的情况下，默认的 nginx 使用的用户。如果
没有指定，默认为 nobody

--group=<GROUP> #在 nginx.conf 中没有指定 user 指令的情况下，默认的 nginx 使用的组。如果
没有指定，默认为 nobody

--builddir=DIR #指定编译的目录

--with-rtsig_module #启用 rtSIG 模块

--with-select_module(--without-select_module) #允许或不允许开启 SELECT 模式，如果
configure 没有找到合适的模式，比如，kqueue(sun os)、epoll(linux kernel 2.6+)、rtSIG(实
时信号)或/dev/poll（一种类似 select 的模式，底层实现与 SELECT 基本相同，都是采用轮询的方
法），SELECT 模式将是默认安装模式
```

--with-poll_module(--without-poll_module) #允许或不允许开启 POLL 模式，如果没有合适的模式，比如：kqueue (sun os)、epoll (linux kernel 2.6+)，则开启该模式

--with-http_ssl_module #开启 HTTP SSL 模块，使 NGINX 可以支持 HTTPS 请求。这个模块需要已经安装了 OPENSSL，在 DEBIAN 上是 libssl

--with-http_realip_module #启用 ngx_http_realip_module

--with-http_addition_module #启用 ngx_http_addition_module

--with-http_sub_module #启用 ngx_http_sub_module

--with-http_dav_module #启用 ngx_http_dav_module

--with-http_flv_module #启用 ngx_http_flv_module

--with-http_stub_status_module #启用 "server status" 页

--without-http_charset_module #禁用 ngx_http_charset_module

--without-http_gzip_module #禁用 ngx_http_gzip_module。如果启用，需要 zlib。

--without-http_ssi_module #禁用 ngx_http_ssi_module

--without-http_userid_module #禁用 ngx_http_userid_module

--without-http_access_module #禁用 ngx_http_access_module

--without-http_auth_basic_module #禁用 ngx_http_auth_basic_module

--without-http_autoindex_module #禁用 ngx_http_autoindex_module

--without-http_geo_module #禁用 ngx_http_geo_module

--without-http_map_module #禁用 ngx_http_map_module

--without-http_referer_module #禁用 ngx_http_referer_module

--without-http_rewrite_module #禁用 ngx_http_rewrite_module。如果启用需要 PCRE。

--without-http_proxy_module #禁用 ngx_http_proxy_module

--without-http_fastcgi_module #禁用 ngx_http_fastcgi_module

--without-http_memcached_module #禁用 ngx_http_memcached_module

--without-http_limit_zone_module #禁用 ngx_http_limit_zone_module

```
--without-http_empty_gif_module #禁用 ngx_http_empty_gif_module

--without-http_browser_module #禁用 ngx_http_browser_module

--without-http_upstream_ip_hash_module #禁用 ngx_http_upstream_ip_hash_module

--with-http_perl_module #启用 ngx_http_perl_module

--with-perl_modules_path=PATH #指定 perl 模块的路径

--with-perl=PATH #指定 perl 执行文件的路径

--http-log-path=PATH #指定 http 默认访问日志的路径

--http-client-body-temp-path=PATH #指定 http 客户端请求缓存文件存放目录的路径

--http-proxy-temp-path=PATH #指定 http 反向代理缓存文件存放目录的路径

--http-fastcgi-temp-path=PATH #指定 http FastCGI 缓存文件存放目录的路径

--without-http #禁用 HTTP server

--with-mail #启用 IMAP4/POP3/SMTP 代理模块

--with-mail_ssl_module #启用 ngx_mail_ssl_module

--with-cc=PATH #指定 C 编译器的路径

--with-cpp=PATH #指定 C 预处理器的路径

--with-cc-opt=OPTIONS #设置 C 编译器的额外选项

--with-ld-opt=OPTIONS #设置链接的额外选项

--with-cpu-opt=CPU #为特定的 CPU 编译，有效的值包括：pentium, pentiumpro, pentium3,
pentium4, athlon, opteron, amd64, sparc32, sparc64, ppc64

--without-pcre #禁止 PCRE 库的使用。同时也会禁止 HTTP rewrite 模块。在 "location" 配置指令中的正则表达式也需要 PCRE

--with-pcre=DIR #指定 PCRE 库的源代码的路径

--with-pcre-opt=OPTIONS #设置 PCRE 的额外编译选项

--with-md5=DIR #使用 MD5 汇编源码

--with-md5-opt=OPTIONS #设置 MD5 库的额外编译选项
```

```
--with-md5-asm #使用 MD5 汇编源码

--with-sha1=DIR #设置 sha1 库的源代码路径

--with-sha1-opt=OPTIONS #设置 sha1 库的额外编译选项

--with-sha1-asm #使用 sha1 汇编源码

--with-zlib=DIR #设置 zlib 库的源代码路径

--with-zlib-opt=OPTIONS #设置 zlib 库的额外编译选项

--with-zlib-asm=CPU #zlib 针对 CPU 的优化，合法的值是：pentium，pentiumpro

--with-openssl=DIR #设置 OpenSSL 库的源代码路径

--with-openssl-opt=OPTIONS #设置 OpenSSL 库的额外编译选项

--with-debug #启用调试日志

--add-module=PATH #添加一个在指定路径中能够找到的第三方模块
```

在不同版本间，选项可能会有些许变化，请总是使用 `./configure --help` 命令来检查当前的选项列表。

测试

将 Nginx conf 文件的 server block 部分的配置如下：

```
server {
    listen 80;
    server_name localhost;

    location / {
        root html;
        index index.html index.htm;
    }

    # redirect server error pages to the static page /50x.html
    error_page 500 502 503 504 /50x.html;
    location = /50x.html {
        root html;
    }
}
```

用户可以通过访问“<http://localhost:80/index.html>”页面来查看 Nginx 的欢迎页面。

Nginx 在 Windows 环境下查看 nginx 进程

用户还可以通过命令行运行 tasklist 命令来查看 nginx 进程:

```
C:\>tasklist /fi "imagename eq nginx.exe"
```

映像名称 PID 会话名 会话# 内存使用

```
=====
nginx.exe 463024 Console 1 5,036 K
nginx.exe 462960 Console 1 5,280 K
```

如果 nginx 没有启动或没有得到预期展示页面, 可查看 error.log 文件以查看失败原因。如果日志文件不存在, 可在 Windows 事件日志中查看。

Nginx 在 Linux 环境下查看 nginx 进程

用户可以通过执行 ps/top 命令来查看 nginx 进程:

```
ps aux|grep nginx
```

```
admin 24913 0.0 0.0 58596 1048 ? Ss Feb27 0:00 nginx: master process ./nginx
admin 24914 0.0 0.0 72772 5420 ? S Feb27 0:03 nginx: worker process
```

同上, 如果 nginx 没有启动或者没有得到预期展示页面, 可以查看 error.log 文件或调试来查看失败原因。

C.2 调试日志

用户在使用 Nginx 的过程中, 可能会遇到所请求的资源不正确, Nginx Core Dump, 段错误等异常情况, 这时需要有相应的机制来进行调试及问题定位, 特别是面对大量的日志信息, 合理的调试处理机制对用户来说是一件非常重要的事情。以下将着重为大家介绍调试日志。

一, 开启调试日志:

要开启调试日志, 首先需要在配置 Nginx 时打开调试功能, 然后编译:

```
./configure --with-debug ...
```

然后在配置文件中设置 error_log 的级别为:

```
error_log /path/to/log debug;
```

Nginx 的 Windows 二进制版本总是将调试日志开启的，因此只需要设置 debug 的日志级别即可。

二，日志级别分析：

在此，我们通过分析 Nginx 源码了解下 Nginx 将日志分为几个等级及不同日志等级之间的相互关系：

Ngx_log.h 代码

```
#define NGX_LOG_STDERR 0
#define NGX_LOG_EMERG 1
#define NGX_LOG_ALERT 2
#define NGX_LOG_CRIT 3
#define NGX_LOG_ERR 4
#define NGX_LOG_WARN 5
#define NGX_LOG_NOTICE 6
#define NGX_LOG_INFO 7
#define NGX_LOG_DEBUG 8

#define NGX_LOG_DEBUG_CORE 0x010
#define NGX_LOG_DEBUG_ALLOC 0x020
#define NGX_LOG_DEBUG_MUTEX 0x040
#define NGX_LOG_DEBUG_EVENT 0x080
#define NGX_LOG_DEBUG_HTTP 0x100
#define NGX_LOG_DEBUG_MAIL 0x200
#define NGX_LOG_DEBUG_MYSQL 0x400

#define NGX_LOG_DEBUG_FIRST NGX_LOG_DEBUG_CORE
#define NGX_LOG_DEBUG_LAST NGX_LOG_DEBUG_MYSQL
#define NGX_LOG_DEBUG_CONNECTION 0x80000000
#define NGX_LOG_DEBUG_ALL 0x7fffffff0
```

其中默认有效的第一级别日志

是“stderr”，“emerg”，“alert”，“crit”，“error”，“warn”，“notice”，“info”，“debug”。而 Ngx_log.h 内

列出的其他 debug 第二级别日

志：“debug_core”，“debug_alloc”，“debug_mutex”，“debug_event”，“debug_http”，“debug_mail”

，“debug_mysql”等则需要在配置 Nginx 时启动调试日志功能才能使用，并且用户可以通过修改

Ngx_log.h 及 Ngx_log.c 源码来更新 debug 第二级别。

我们再通过 Ngx_log.c 的部分代码分析下可以如何使用这些日志级别：

Ngx_log.c 代码

```
char *
ngx_log_set_levels(ngx_conf_t *cf, ngx_log_t *log)
{
    ...

    for (n = 1; n <= NGX_LOG_DEBUG; n++) {
        if (ngx_strcmp(value[i].data, err_levels[n].data) == 0) {

            if (log->log_level != 0) {
                ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
                                    "duplicate log level \"%V\"",
                                    &value[i]);
                return NGX_CONF_ERROR;
            }

            log->log_level = n;
            found = 1;
            break;
        }
    }

    for (n = 0, d = NGX_LOG_DEBUG_FIRST; d <= NGX_LOG_DEBUG_LAST; d <= 1) {
        if (ngx_strcmp(value[i].data, debug_levels[n++]) == 0) {
            if (log->log_level & ~NGX_LOG_DEBUG_ALL) {
                ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
                                    "invalid log level \"%V\"",
                                    &value[i]);
                return NGX_CONF_ERROR;
            }

            log->log_level |= d;
            found = 1;
            break;
        }
    }

    ...
    if (log->log_level == NGX_LOG_DEBUG) {
        log->log_level = NGX_LOG_DEBUG_ALL;
    }
    ...
}
```

按照以上代码逻辑，我们可以得出以下结论：

1. 第一级别日志之间是互斥的，如果配置文件内加入如下配置项：

```
error_log path/logs/error.log warn;
error_log path/logs/error.log info;
```

那么启动 Nginx 将报错如下:

```
[emerg]: duplicate log level "info" in /path/conf/nginx.conf:XX
```

但是需要注意的是,在配置文件不同 block 中是允许重新定义错误日志的。但是当用户在重新定义错误日志时,如果没有指定相应的日志级别,那么调试日志将会被屏蔽。下面的例子里,在 server 层中重新定义的日志就屏蔽了这个虚拟主机的调试日志:

```
error_log /path/to/log debug;

http {
    server {
        error_log /path/to/log;
        ...
    }
}
```

为了避免这个问题,可以注释这行重新定义日志的配置,或者也给日志指定 debug 级别:

```
error_log /path/to/log debug;

http {
    server {
        error_log /path/to/log debug;
        ...
    }
}
```

2. 第二级别日志是多选的,用户可以根据项目需要配置多个第二级别日志:

```
error_log logs/error.log debug_mysql;
error_log logs/error.log debug_core;
```

3. 在第一级别日志与第二级别日志组合配置时,仅有在第一级别日志为“debug”时才可以有第二级别的配置,其他第一级别日志的情况下指定第二级别日志将无法启动 Nginx,如:

```
error_log logs/error.log error;
error_log logs/error.log debug_core;
```

启动 Nginx 将获得如下错误信息:

```
[emerg]: invalid log level "debug_http" in /path/conf/nginx.conf:XX
```


当用户开启 debug 级别日志时，会输出所有 debug_开头的调试信息，因此可以通过上面组合 debug_core|debug_http 的形式来获取用户所需要的调试信息。

三，日志格式设置：

用户在使用 Nginx 提供 web 服务的时候，可能会有很多场景需要记录日志，如打点日志，访问日志，数据统计日志，性能分析日志等。为了更加方便的对日志进行分析，我们可以通过设置日志格式的方式来要求 Nginx 按照用户要求进行日志的展现。

控制 nginx 日志输出的指令如下：

```
log_format customLog
"$remote_addr^A$remote_user^A$time_local^A$request_method^A$uri^A$args^A$server_
protocol"
    ^A$status^A$body_bytes_sent^A$http_referer"
    ^A$http_user_agent";
access_log /path/logs/access.log customLog;
```

上面例子中通过使用特殊字符（^A）来作为日志字段的分隔符，用户后续可以使用 sort 和 grep 之类的工具对特定 url 做分析，如统计各 url 请求量倒排取前 50 个：

```
awk -F^A '{print $5}' /path/logs/access.log | sort | uniq -c | sort -nr | head -50
```

类似上面的日志定制化设置，可以让用户在调试日志的过程中随心所欲，如鱼得水。 详细的 log_format 指令和 access_log 指令，用户可以访问 Nginx 官网的 HttpLog 模块 <http://wiki.nginx.org/HttpLogModule> 。

四，调试日志的几个注意点：

1. 勘误：在 Nginx Wiki 里面 error log 相关部分

（http://wiki.nginx.org/NginxHttpMainModule#error_log）的介绍中提到

```
Default values for the error level:
in the main section - error
in the HTTP section - crit
in the server section - crit
```

但是，我们从源码上看：

```
static char *
ngx_error_log(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
{
    ...

    if (cf->args->nelts == 2) {
        cf->cycle->new_log.log_level = NGX_LOG_ERR;
        return NGX_CONF_OK;
    }
    ...
}
```

当 `error_log` 的日志级别选项为配置时,默认日志级别为 `error`,无上面提及的三个 `section` 的区别。故特在此勘误。

2. 配置 `error_log off` 并不能关闭日志记录——日志信息会被写入到文件名为 `off` 的文件当中。如果要关闭日志记录,用户可以做如下配置:

```
error_log /dev/null crit;
```

3. 如果 `nginx` 进程没有权限将日志信息写入指定的 `log` 地址,那么 `nginx` 会在启动是报错:

```
[alert]: could not open error log file: open() "/path/log/nginx/error.log" failed
(13: Permission denied)
```

4. 通过 `debug_connection` 配置项,用户可以针对某些地址开启调试日志:

```
error_log /path/to/log;

events {
    debug_connection 10.232.10.1;
    debug_connection 10.232.10.0/24;
}
```

C.3 使用 GDB 调试

C.4 功能测试

C.5 性能/压力测试

C.6 常见缺陷分析