# Nginx

## ngx\_cycle\_t结构体

void \*\*\*\*conf\_ctx为什么是四个指针？

总的来说conf\_ctx是一个指向指针数组的指针，这就是两个\*。而这个数组的每个成员又是指向指针数组的指针。所以一共四个\*。

具体看下这个字段的赋值：

conf\_ctx是在ngx\_init\_cycle中赋值。在后续的create\_conf过程中，定义了create\_conf的NGX\_CORE\_MODULE模块，比如ngx\_thread\_pool会将执行create\_conf的结果保存在conf\_ctx的数组中。

对于没有定义create\_conf的模块，比如ngx\_http\_module，则是在模块的配置解析函数里完成对conf\_ctx数组成员的赋值。比如当解析到配置http，就会触发ngx\_http\_block函数，创建了一个用于保存http模块配置解析内容的数据结构，赋值给conf\_ctx作为其数组成员。

不管是create\_conf还是NGX\_CORE\_MODULE的配置解析函数，都是创建了一块内存空间，用来保存该模块配置解析的内容。

需要说明的是，并不是所有的模块在conf\_ctx里面存储的都是相同的数据结构，比如http模块和event模块存储的数据结构就不一样。

## 二、Nginx启动流程

1、Nginx配置可以通过配置文件的方式，也可以通过命令行的方式。因此首先要确定配置文件的路径。

2、启动过程中的核心函数之一ngx\_init\_cycle。该函数的作用是解析各个模块的配置，默认配置初始化等工作。

3、ngx框架只会调用所有NGX\_CORE\_MODULE模块的create\_conf方法。核心模块的子模块，比如http模块下的各个子模块，都是由ngx\_http\_module来调用create\_conf。这样的设计可以降低框架的复杂度。

4、并不是所有的NGX\_CORE\_MODULE模块都有create\_conf方法，create\_conf的作用在于创建后续配置解析需要用到的动态内存，比如动态数组。因此只有需要用到动态内存的模块才会有create\_conf方法。

5、接下来ngx\_conf\_parse则是解析具体的配置项。完成配置解析之后，需要打开解析配置得到的目录、文件。同时初始化ngx\_cycle\_t结构体中的shared\_memory链表用于进程间通信的共享内存。

6、然后是对监听端口的处理。配置解析已经解析出各个模块需要监听的端口，并加入到listening数组中，这一步就是要对数组中的每个成员创建socket并监听对应端口。

7、最后则是启动子进程（work进程）。如果是配置了单进程模式，则不会启动子进程。调用各个模块init\_process的方法，之后开始正常的事件循环。

## 三、Nginx事件驱动

ngx\_event\_module是nginx的core模块之一，ngx\_epoll\_module、ngx\_poll\_module、ngx\_select\_module等都是ngx\_event\_module的子模块。这些不同的事件驱动模块适配了多种不同操作系统的事件驱动模型。

ngx\_event\_module抽象了十种事件驱动需要用到操作：add、del、enable、disable、add\_conn、del\_conn、process\_changes、process\_events、init、done。

nginx的事件并不是在使用的时候才去创建，而是在ngx启动过程中就已经创建完成了并保存在ngx\_cycle\_t的read\_events和write\_events两个事件数组中。在上面描述的ngx启动流程最后一步，会调用各个模块的init\_process方法，就包括ngx\_event\_process\_init，这个函数就实现了上述功能。

ngx\_cycle\_t的结构体中定义了connections数组，用来存储nginx接收的客户端连接，这种连接由叫做被动连接。区别于nginx连接上游服务器的连接ngx\_peer\_connenction\_t（主动连接）。connections同样也是在ngx\_event\_process\_init函数中赋值的。上述三个数组的大小是一样的。

connections和free\_connections两个数据结构构成了一个循环链表。connections指向表头，而free\_connections则指向第一个空闲连接。当有连接接入的时候，从free\_connections取出一个空闲连接，然后将free\_connections指向next。归还连接的时候只要直接将连接加入表头即可。ngx\_connection\_t中有read和write两个指针，分别指向了上述read\_events和write\_events中的成员。

nginx可以通过ngx\_event\_core\_module里面的use配置来决定使用哪一种事件驱动。但代码中use配置是没有默认值的，那么如果没有配置use的话，会选择哪个事件驱动呢？在ngx\_event\_core\_init\_conf函数里面，是根据宏定义来判断的，这个是在编译阶段决定的。

通过ngx\_event\_t中的instance字段来判断事件是否过期。利用了内存地址最后一位一定是0的特性，在ngx\_epoll\_add\_event事件中，instance字段会或上ngx\_connection\_t结构体的地址，并将其赋值给event.data.ptr传入内核。当事件触发，由内核传递给应用程序处理的时候，ngx\_epoll\_process\_event先从ptr中取出instance然后将ptr还原成ngx\_connection\_t的数据结构。比较ngx\_connection\_t中读写事件的instance字段和ptr中取出的instance读写字段是否一致，就能判断事件是否过期。

读事件的instance字段在ngx\_event\_process\_init时候会初始化为1，写事件的instance字段则初始化为0。每次ngx\_get\_connection时，读写事件的instance字段都会取反，加上上面的流程就能判断事件是否过期。

## 四、Nginx定时事件

ngx每个子进程都会独立管理时间。时间更新是在ngx\_epoll\_process\_events中。当flag中设置了NGX\_UPDATE\_TIME或ngx\_timer\_alarm == 1时，调用ngx\_time\_update更新时间。

ngx配置项time\_resolution是设置时间最长更新间隔。通过系统调用setitimer，每隔time\_resolustion毫秒调用一次ngx\_timer\_singal\_handler，将ngx\_timer\_alarm设置为1。

设置了time\_resolution之后，epoll\_wait的timer参数将被设置为-1。这样如果没有时间发生，epoll\_wait会立即返回，不会一直等待。这样可以避免ngx\_epoll\_process\_events长时间得不到执行。但是如果某个handler模块回调执行时间过长，时间精度还是会受到影响。

保存定时事件的数据结构是红黑树。超时时间最短的时间在树的最左边。每次只需要将最左节点与当前时间比较，就可以知道是否有超时事件。调用ngx\_expire\_timer会执行所有已超时的事件。

## 五、惊群”问题

所谓惊群问题，就是多个子进程都监听了某个端口。在没有请求接入的时候，子进程都在等待。当有一个请求接入的时候，所有的子进程都会被唤醒，但最终只有一个子进程能accept成功，其他子进程又重新进入等待。而系统唤醒子进程的代价是很大的。

ngx通过配置accept\_mutex锁来解决这个问题，默认是开启。通过ngx\_accept\_disabled变量控制子进程是否竞争accept\_mutex锁。

当前空闲连接数小于总连接的1/8的时候，ngx\_accpet\_disabled大于0，当前子进程不竞争accept\_mutex锁。accept\_mutex是一个共享内存锁，加锁之后其值为进程号。获得accept\_mutex锁的进程才能处理accept事件和已有的连接事件。而没有获得锁的进程只能处理已有的连接事件。

accept\_mutex锁何时释放？accept获得新连接的进程并不会马上处理新连接上的事件，而是将新连接加入ngx\_posted\_accept\_event队列中。已有连接的事件则加入ngx\_posted\_event队列中。然后先处理ngx\_posted\_accept\_event队列中的事件，处理完成之后立刻释放accept\_mutex锁，再去处理ngx\_posted\_event事件。

ngx\_process\_events\_and\_timers是ngx处理所有事件（post事件和定时事件）。

## work进程

|  |  |  |
| --- | --- | --- |
| 信号 | 全局变量 | 意义 |
| QUIT | ngx\_quit | 正常退出 |
| TERM或INT | ngx\_terminate | 强制退出 |
| USR1 | ngx\_reopen | 重新打开所有文件 |
| WINCH | ngx\_debug\_quit | 无实际意义 |

ngx\_work\_process\_cycle实际上就是一个死循环，循环里做的事情就是处理各种信号、网络事件和定时事件。

正常退出会调用所有活动连接的处理方法去关闭连接，同时保证所有定时事件处理完成之后才会调用所有模块的exit\_process方法。

强制退出则是直接调用所有模块exit\_process方法，退出work进程。

## master进程

|  |  |  |
| --- | --- | --- |
| 信号 | 全局变量 | 意义 |
| QUIT | ngx\_quit | 正常退出 |
| TERM或INT | ngx\_terminate | 强制退出 |
| USR1 | ngx\_reopen | 重新打开所有文件 |
| WINCH | ngx\_noaccept | 所有子进程不接受新连接，相当于对所有子进程发送QUIT信号 |
| USR2 | ngx\_change\_binary | 平滑升级Nginx版本 |
| HUP | ngx\_reconfigure | 重配置 |
| CHLD | ngx\_reap | 有子进程以外结束 |

ngx\_processes数组来保存所有的子进程。通过ngx\_spawn\_process函数调用fork来创建子进程。子进程最多1024个。

如果有子进程意外退出，发出CHLD信号，会触发ngx\_signal\_handler，调用ngx\_process\_get\_status。通过waitpid系统调用得到意外退出的子进程id，在ngx\_processes数组中找到对应的子进程结构体，将其exited标志位置为1。master进程的循环里会调用ngx\_reap\_children来将异常退出的子进程拉起。

## Nginx内存池

ngx内存池分为大块内存和小块内存。在ngx\_create\_pool函数中，一个内存池，小块内存最大不能超过4095个字节。如果调用ngx\_create\_pool传入的size小于4095的时候，那么对于该内存池，小于等于size大小的内存属于小块内存。

小块内存池在数据结构上是通过链表的方式组织起来的，当一个小块内存池剩余空间不足够分配内存的时候，会重新创建一个新的ngx\_pool\_t组成链表。后续再分配内存的时候，遍历链表的每一个pool，判断其剩余内存是否满足分配需求。如果不满足，则到下一个pool分配。需要注意的是，如果一个pool超过4次不满足分配需求，那么会将链表的current指针指向下一个pool。这样下次再分配内存将直接略过这个pool，以提高内存分配的效率。

大内存则不是通过内存池的形式来管理的，大内存是直接通过malloc系统调用来管理的， 然后通过链表将大内存块组织起来。如果申请了超过3个大内存块，则会将新申请的大内存放在链表的头部，提升效率。

对于小内存来说，使用者一般是不需要关心什么时候释放它，只要直接释放整个内存池就可以了。小内存的生命周期和内存池的生命周期是一样的。而对于大内存，如果其本身生命周期远小于内存池的生命周期，那么就有必要提前释放大内存。ngx\_pfree就提供了删除大内存块的功能。

需要留意的是，删除大内存块，不会删除ngx\_pool\_large\_t这个数据结构，只是删除其指向的内存空间，数据结构本身是可以被复用的。下次如果再有申请大内存卡，直接挂在空ngx\_pool\_large\_t数据结构上即可，这样也可以减少链表操作。

ngx内存管理还提供了文件资源的管理，这样用户就不需要释放文件等资源，通过ngx\_pool\_cleanup\_t数据结构来实现。

## 九、Nginx监听端口接收请求的过程

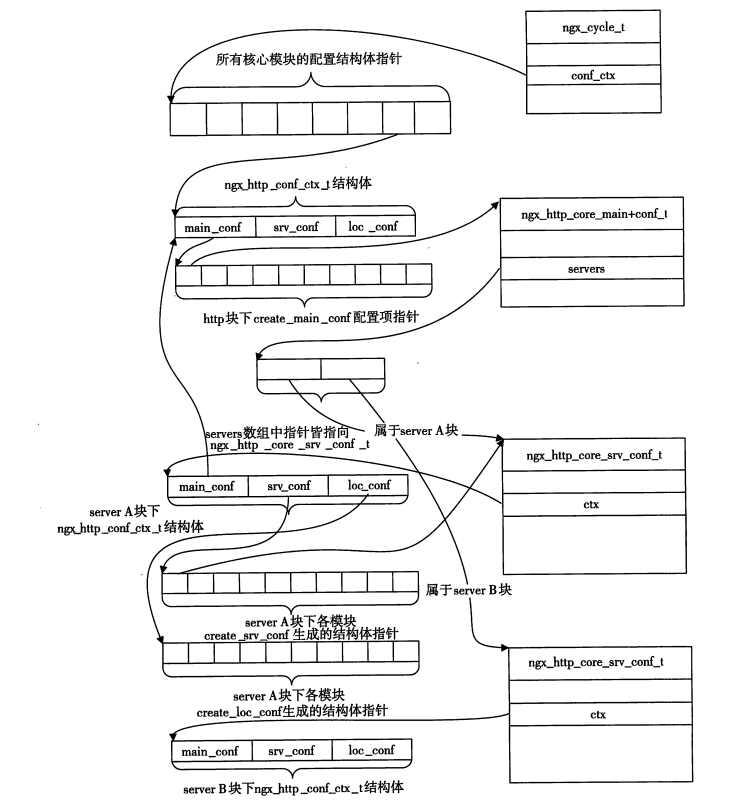
配置解析完成之后，调用ngx\_open\_listening\_sockets函数来创建socket、bind、listen等操作。

ngx\_init\_cycle会调用各个模块的init函数，其中包括event模块的ngx\_event\_process\_init函数，其中给每个监听端口都添加了读事件，当有请求接入的时候就会回调ngx\_event\_accept函数处理。ngx\_event\_accept函数中会为调用ngx\_get\_connection为新连接分配connection结构，同时设置ngx\_recv和ngx\_send函数。调用ngx\_epoll\_add\_connection为新建立的连接设置epoll事件，最后调用的hanlder实际上就是ngx\_http\_init\_connection。

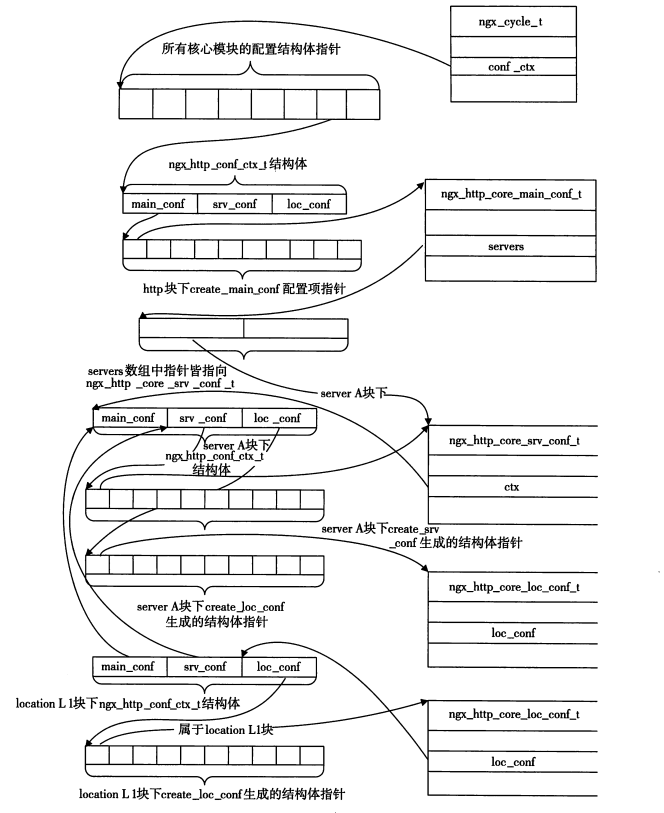
## 十、HTTP模块配置管理

ngx\_http\_conf\_ctx\_t有三个指针数组void \*\*main\_conf; void \*\*svr\_conf; void \*\*loc\_conf。组织关系如下：

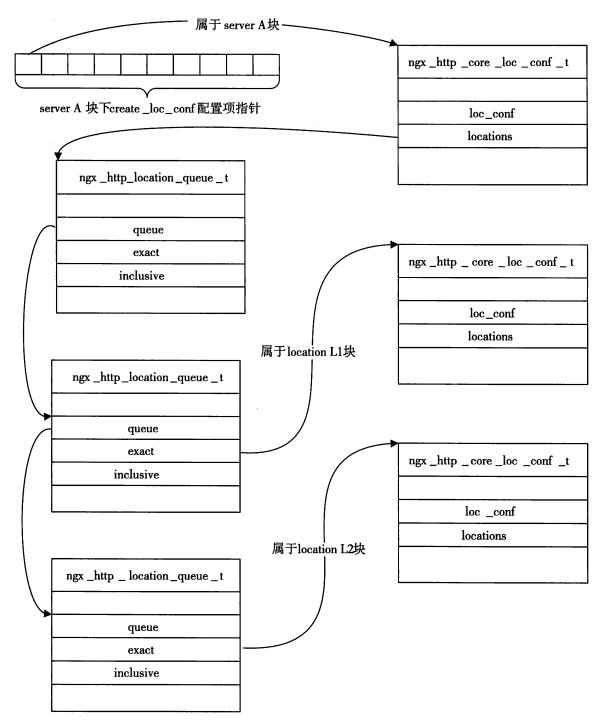
main\_conf成员是通过ngx\_module\_t中的create\_main\_conf函数来创建的，大部分的http模块没有实现这个函数。以ngx\_http\_core\_module\_t为例，create\_main\_conf创建了ngx\_http\_core\_mian\_conf\_t数据结构作为main\_conf数组的成员。main\_conf的成员是不同的数据结构，比如ngx\_http\_upstream\_module\_t的create\_main\_conf创建的数据结构是ngx\_http\_upstream\_main\_conf\_t，这一点从main\_conf是void类型的指针也可以得到印证。



main\_conf下面有个server的数组，里面保存了这个main下面所有的srv\_conf。而某个srv\_conf则是通过ctx的main\_conf指针关联到它所属的main\_conf。

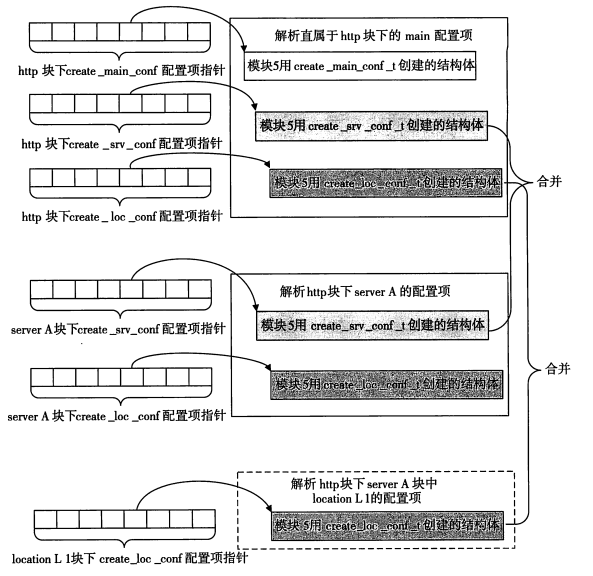


loc\_conf和srv\_conf与main\_conf之间的关系。同一个server下的location配置是通过双向链表串联起来的，ngx\_http\_core\_loc\_conf\_t里面的location就是双向链表。



配置解析完成之后会进行配置合并，配置合并可以解决几个问题

1. 没有的配置的配置项设置默认值。
2. 将mian配置的配置项同步到下属的srv配置中。
3. 将srv配置的配置项同步到下属的loc配置中。



server对应http请求的host。location对应http请求的url。

## 十一、监听端口管理

ngx\_http\_core\_main\_conf\_t结构中ngx\_array\_t \*ports存放的就是所有的监听端口ngx\_http\_conf\_port\_t。这个数据结构里也有一个动态数组ngx\_array\_t addrs存放着ngx\_http\_conf\_addr\_t，即某个端口对应的ip。ngx\_http\_conf\_addr\_t的动态数组则保存着ngx\_http\_core\_srv\_conf\_t。

收到http请求之后会开始解析http请求头部，在ngx\_http\_process\_request\_line，找到完整的host信息且判断host信息有效后会调用ngx\_http\_set\_virtual\_server，根据host信息找到对应的srv和loc配置。

## 十二、HTTP处理过程

十一个阶段：

NGX\_HTTP\_POST\_READ\_HEAD

NGX\_HTTP\_SERVER\_REWRITE\_PHASE

NGX\_HTTP\_FIND\_CONFIG\_PHASE

NGX\_HTTP\_REWRITE\_PHASE

NGX\_HTTP\_POST\_REWRITE\_PHASE

NGX\_HTTP\_PREACCESS\_PHASE

NGX\_HTTP\_ACCESS\_PHASE

NGX\_HTTP\_POST\_ACCESS\_PHASE

NGX\_HTTP\_TRY\_FILES\_PHASE

NGX\_HTTP\_CONTENT\_PHASE

NGX\_HTTP\_LOG\_PHASE

其中NGX\_HTTP\_FIND\_CONFIG\_PHASE、NGX\_HTTP\_POST\_REWRITE\_PHASE、NGX\_HTTP\_POST\_ACCESS\_PHASE、NGX\_HTTP\_TRY\_FILE\_PHASE这四个阶段是不能够挂载http自定义的handler方法，只能使用框架定义的。

每个处理阶段都有checker方法和handler方法，checker方法决定是否执行handler方法，而且checker方法无法自定义。

NGX\_HTTP\_SERVER\_REWRITE\_PHASE

NGX\_HTTP\_FIND\_CONFIG\_PHASE

NGX\_HTTP\_REWRITE\_PHASE

NGX\_HTTP\_POST\_REWRITE\_PHASE

NGX\_HTTP\_ACCESS\_PHASE

NGX\_HTTP\_TRY\_FILES\_PHASE

NGX\_HTTP\_CONTENT\_PHASE

是有定义checker方法的，剩余的阶段都是调用ngx\_http\_core\_generic\_phase作为checker方法。这个函数的实现很简单就是直接调用handler，然后通过handler方法的返回值来判断接下来的策略。

ngx\_http\_core\_content\_phase里面会判断ngx\_http\_request\_t里面的content\_handler，如果http模块有定义这个handler则不会执行ngx\_http\_phase\_hander里面的hander方法。这里就涉及到一个hander方法挂载问题。

如果返回NGX\_OK，那么不管是不是还有handler方法没有执行完成，都不会再继续执行，而是进入下一个phase。如果返回NGX\_DECLINED则继续执行下一个handler方法。如果返回NGX\_AGAIN或者NGX\_DONE则停留在当前的处理函数，下次继续调用。

ngx\_http\_process\_request\_line函数开始接收数据，因为nginx的epoll是边缘触发机制，因此函数通过一个循环不断的调用recv来接受缓冲区数据。接收到的数据保存在ngx\_http\_request\_t的header\_in这个buf中。nginx每次接收到一部分数据之后就会进行解析。解析发现不完整则继续接收数据，然后再解析。如此反复，直到http头部解析完成。

请求行解析完成之后就开始解析请求头，也是边读取边解析。

处理完请求行和请求头就进入上面说的十一个处理阶段ngx\_http\_core\_run\_phases。

## 十三、handler方法挂载

nginx模块分两种挂载方式：按处理阶段挂载和按需挂载。

按处理阶段挂载：对于十一个nginx处理阶段都适用，做法就是将handler函数挂载到ngx\_http\_core\_main\_conf\_t中各个phase的handler数组中。

ngx\_http\_core\_main\_conf\_t中的phase是一个数组对应了是一个处理阶段。数组成员handlers是一个动态数组，这个数组中则保存了各个http模块自定义的在某一个或者多个处理阶段需要调用的handler函数。handlers动态数组的赋值一般在各个http模块的init函数中进行。赋值的先后顺序也就决定了调用的先后顺序。

按需挂载：这种挂载方式只有在NGX\_HTTP\_CONTENT\_PHASE阶段才能被执行。这种挂载方式的另外一个特点就是如果使用了这种方式，则不会再去执行其他模块通过上面一种方式在NGX\_HTTP\_CONTENT\_PHASE阶段挂载的函数。做法就是将handler赋值给ngx\_http\_core\_loc\_conf\_t下的handler函数指针。

## 十四、handler模块的开发步骤

1、编写模块的基本结构：

模块的定义，即ngx\_module\_t结构，每个模块都会定义这样一个结构。

模块的上下文，即ngx\_module\_t的ctx字段。比如http模块就是定义ngx\_http\_module\_t，event模块就是定义ngx\_event\_module\_t。主要是定义了配置相关的操作，比如create\_conf、init\_conf等。

模块的配置，即ngx\_command\_t结构，对应ngx\_module\_t的command字段。

2、实现handler挂载函数。

3、实现handler处理函数。

4、将新增模块加入编译脚本。在新增模块的目录下创建一个config文件，config文件定义需要编译的模块名称、源文件名称。然后通过configure --add-module参数来指定需要编译的模块。

## 十五、filter模块

filter模块是对http响应头和响应体处理的模块。filter模块调用顺序是在编译的时候决定的。在ngx\_modules.c文件中，可以看到filter模块的定义，实际执行的时候顺序和文件中定义的先后顺序相反。自定义的filter模块只能加入到copy\_filter和headers\_filter之间执行。

不论是header\_filter还是body\_filter都会被nginx框架串联成链表的形式。以header\_filter为例，两个指针全局变量http\_top\_header\_filter和局部变量http\_next\_header\_filter，top\_header指向当前的filter模块，next\_header则指向上一个filter模块。top\_header最终就是所有filter模块的入口，通过next\_header找到下一个filter模块。执行完了所有的header\_filter之后再执行body\_filter。

第一个执行的是ngx\_http\_not\_modified\_filter\_module模块，这个模块只处理响应头部的，没有响应体的filter函数。作用就是如果请求的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。

倒数第二个执行的是ngx\_http\_header\_filter\_module，将所有的headers组成一个完整的http头部。

最后一个执行的是ngx\_http\_write\_filter\_module，将所有需要输出的内容存入ngx\_http\_request\_t的out链表中。

## 十六、upstream模块的使用方法

需要使用upstream机制，先调用ngx\_http\_upstream\_create创建upstream结构。upstream结构提供了如下的回调函数由使用者自行定义。

|  |  |
| --- | --- |
| create\_request | 生成发送到后端服务器的请求缓冲 |
| 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。选定新服务之后会调用该函数，重新初始化upstream的状态 |
| process\_header | 处理后端服务器返回的信息头部 |
| abort\_request | 客户端放弃请求时被调用，但不需要实现关闭与后端服务器连接的操作 |
| finalize\_request | 正常完成与后端服务器的请求后调用该函数 |
| input\_filter | 处理后端服务器返回的响应正文 |
| input\_filter\_init | 初始化input\_filter的上下文 |

abort\_request和finalize\_request一般不会有什么具体的操作，在nginx的memcache模块中，upstream这两个回调方法都只是输出了日志打印而已。而且memcache的reinit\_request函数也是什么都没做，但是这个接口不同模块的实现差异比较大，有的模块是需要做处理的。

完成上述回调函数之后调用ngx\_http\_upstream\_init完成对整个upstream结构的初始化过程。

ngx\_http\_upstream\_init->ngx\_http\_upstream\_init\_request->ngx\_http\_upstream\_connect->ngx\_http\_upstream\_send\_request

ngx\_http\_upstream\_process\_header是处理响应头部的入口函数，先处理异常情况，如果是写超时或者连接失败，那么就换一个后端服务器。如果是正常情况，就for循环不断的recv数据，交给init阶段定义的process\_header回调函数处理。

upstream有三种处理上游服务器包体的方式：

1. 不转发包体
2. 以下游网速优先来转发响应

开辟固定大小的缓冲区，如果缓冲区满了就停止接收上游服务器的响应数据。缓冲区数据发送给客户端之后会清空，就可以继续接收上游服务器的响应数据。

1. 以上游网速优先来转发响应

开辟更大的缓冲区接收上游服务器的响应数据，如果缓冲区存满了就会将数据保存到临时文件中。

## 十七、upstream负载均衡

1. 轮询-默认负载均衡策略

轮询算法支持按照权重进行轮询。轮询中如果发现后端服务器不可用，则自动删除。

1. ip\_hash

根据客户端ip进行哈希，保证同一个客户端始终连接相同的后端服务器。如果后端服务器不可用，只能手动删除。

1. 最少连接

这种方式就是解决连接时间长短不一导致后端服务器负载不均衡的情况。

第三负载均衡插件：

fair：服务器响应时间短的优先分配。

url\_hash：根据url来hash后端服务器，保证缓存有更高的命中率。

## 十八、Nginx进程间通信

nginx使用三种进程间通信方式：共享内存、socket、信号。

使用三种数据同步方式：原子操作、信号量、文件锁。

**共享内存**：

通过mmap或shmget系统调用，nginx中ngx\_stat\_accepted、ngx\_stat\_requests等统计作用的全局变量就是采用共享内存的方式存储的。

**频道socketpair**：

int socketpair(int d, int type, int protocol, int sv[2]);

d在linux系统下通常取值AF\_UNIX

type取值SOCK\_STREAM或者SOCK\_DGRAM

protocol必须传递0。

sv是存放创建好的socket的fd的数组。

函数返回0的时候表示socket创建成功，返回-1的时候表示失败。

nginx先调用socketpair创建套接字，然后fork子进程。然后在父进程中关闭sv[1]的套接字，在子进程中关闭sv[0]的套接字。父进程通过sv[0]跟子进程通信，子进程通过sv[1]跟父进程通信。

work进程也是通过epoll事件驱动来调度从socket频道中接收父进程发来的消息。

**信号**：

nginx的signals数组了定义了会用到的信号以及对应的处理函数。父进程收到信号会调用ngx\_signal\_handler函数进行处理。

**原子操作**：

nginx提供两个原子操作的函数ngx\_atomic\_cmp\_set和ngx\_atomic\_fetch\_add。

**信号量**：

nginx使用信号量作为互斥锁来使用。

通过ngx\_shmtx\_create创建信号量。

ngx\_shmtx\_lock加锁。

ngx\_shmtx\_unlock解锁。

ngx\_shmtx\_destroy销毁信号量。

信号量最开始是0，sem\_post信号量加1，sem\_wait信号量减1。如果信号量小于等于0，那么sem\_wait会阻塞当前进程。

所以sem\_wait是解锁，sem\_post是加锁。

**文件锁**：

nginx的文件锁cmd参数有两个值F\_SETLK和F\_SETLKW，第一个如果获取锁失败会直接返回，不会阻塞。第二个获取锁失败会一直等待。

**互斥锁**：

nginx的互斥锁有两个实现，一个是上面讲的文件锁。另一个是基于原子变量。

如果当前操作系统支持原子变量，则会优先使用原子变量锁。原子变量实现的互斥锁会同时判断是否使用信号量。如果使用信号量的话，则会使进程进入睡眠状态。不使用信号量则互斥锁和自旋锁是一样的。

锁为0或者正数的时候表示没有进程获得了锁，为负数的时候表示有进程获得了锁。和信号量类似。

**自旋锁**：

自旋锁就是获取不到的锁的时候，暂时不要让出正在使用的CPU，而是等待一段时间，检查是否有其他进程释放锁，这样可以减少进程间切换的次数。

如果等待了一段时间也无法获取到锁，则调用ngx\_sched\_yield，当前进程不休眠，而是暂时让出CPU。这样可以减少进程休眠再被唤醒的消耗。