# 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。