一

1 Nginx 特点

Nginx 做为 HTTP 服务器，有以下几项基本特性：

* 处理静态文件，索引文件以及自动索引；打开文件描述符缓冲．
* 无缓存的反向代理加速，简单的负载均衡和容错．
* FastCGI，简单的负载均衡和容错．
* 模块化的结构。包括 gzipping, byte ranges, chunked responses,以及 SSI-filter 等 filter。如果由 FastCGI 或其它代理服务器处理单页中存在的多个 SSI，则这项处理可以并行运行，而不需要相互等待。
* 支持 SSL 和 TLSSNI．

二 Nginx平台初探

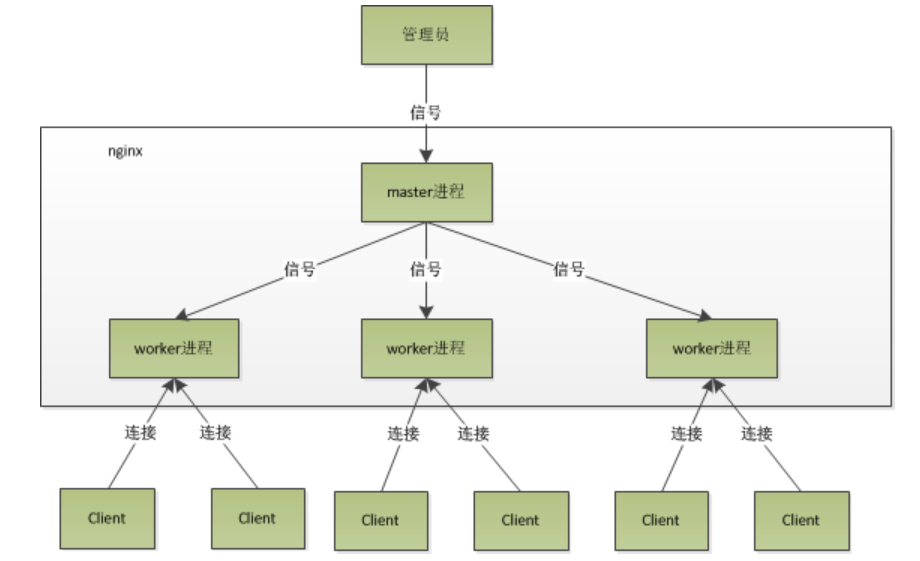
1 初探Nginx架构

1）进程模型

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。

**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。



如果我们要操作 Nginx，只需要与 master 进程通信就行了。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。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。

新版本中 通过reload stop等命令向master发送信号 而不是直接操作master进程。

worker 进程又是如何处理请求的呢？

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。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。

1. 处理事件

高并发：Nginx 采用了异步非阻塞的方式来处理请求，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。与多线程相比，这种事件处理方式是有很大的优势的，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4用更多的内存而已。

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，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。

事件通常有三种类型，网络事件、信号、定时器

网络事件：异步非阻塞解决

信号：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。

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。

1. 连接connection

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。

处理连接：

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worker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。

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。

连接数上限：

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和程序所能支持的最大并发数。当 fd 用完后，再创建 socket 时会失败。

Nginx 通过设置 worker\_connectons 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。

worker\_connections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= 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。

让进程公平地竞争从客户端来的连接：

一个客户端连接过来后，多个空闲的进程，会竞争这个连接，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。）

1. 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-响应类型的的网络协议，而 http 是文本协议，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。**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。 区别：Nginx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。解析到的参数，会保存到 ngx\_http\_request\_t 结构当中。

Nginx 是如何处理一个完整的请求的：

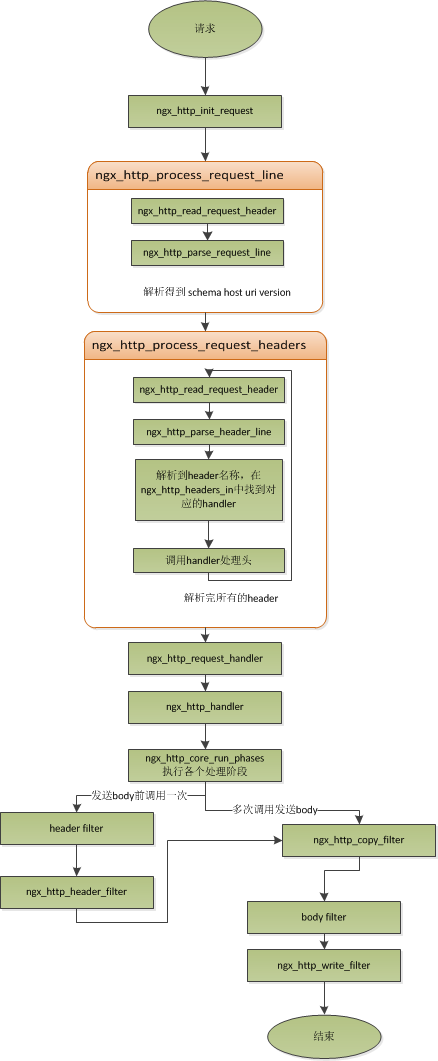
一个请求是ngx\_http\_init\_request开始的，在这个函数中，会设置读事件为 ngx\_http\_process\_request\_line，也就是说，接下来的网络事件，会由 \*\_line 来执行。在解析完请求行后，Nginx 会设置读事件的 handler 为 ngx\_http\_process\_request\_headers，然后后续的请求就在 ngx\_http\_process\_request\_headers 中进行读取与解析。

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。

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。虽然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。 此时请求头已经读取完成了，Nginx 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。

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。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。



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。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。

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。

1. Keepalive

长连接：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。

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。