Nginx入门指南

极客学院出版

前言

Nginx 是一款轻量级的 Web 服务器/反向代理服务器及电子邮件（IMAP/POP3）代理服务器，其特点是占有内存少，并发能力强。

本教程根据淘宝核心系统服务器平台组的成员的日常工作总结而成，主要介绍了 Nginx 平台的特点及模块开发，帮助读者更好的构建和维护 Nginx 服务器。

适用人群

高性能 Web 服务器维护人员，对互联网服务器感兴趣的程序开发者。

学习前提

学习本教程前，我们假定您已经能够搭 Nginx 服务器，并能够进行简单常规的操作。

鸣谢：淘宝核心系统服务器平台组成员 (http://tengine.taobao.org/book/index.html)

目录

1 背景介绍

什么是 Nginx

Nginx 是俄罗斯人编写的十分轻量级的 HTTP 服务器,Nginx，它的发音为“engine X”，是一个高性能的HTTP和反向代理服务器，同时也是一个 IMAP/POP3/SMTP 代理服务器。Nginx 是由俄罗斯人 Igor Sysoev 为俄罗斯访问量第二的 Rambler.ru 站点开发的，它已经在该站点运行超过两年半了。Igor Sysoev 在建立的项目时,使用基于 BSD 许可。

英文主页：http://nginx.net 。

到 2013 年，目前有很多国内网站采用 Nginx 作为 Web 服务器，如国内知名的新浪、163、腾讯、Discuz、豆瓣等。据 netcraft 统计，Nginx 排名第 3，约占 15% 的份额(参见：http://news.netcraft.com/archives/category/web-server-survey/)

Nginx 以事件驱动的方式编写，所以有非常好的性能，同时也是一个非常高效的反向代理、负载平衡。其拥有匹配 Lighttpd 的性能，同时还没有 Lighttpd 的内存泄漏问题，而且 Lighttpd 的 mod_proxy 也有一些问题并且很久没有更新。

现在，Igor 将源代码以类 BSD 许可证的形式发布。Nginx 因为它的稳定性、丰富的模块库、灵活的配置和低系统资源的消耗而闻名. 业界一致认为它是 Apache2.2 + mod_proxy_balancer 的轻量级代替者，不仅是因为响应静态页面的速度非常快，而且它的模块数量达到 Apache 的近 2/3。对 proxy 和 rewrite 模块的支持很彻底，还支持 mod_fcgi、ssl、vhosts ，适合用来做 mongrel clusters 的前端 HTTP 响应。

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 Fast CGI 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.

Nginx 专为性能优化而开发，性能是其最重要的考量,实现上非常注重效率 。它支持内核 Poll 模型，能经受高负载的考验,有报告表明能支持高达 50,000 个并发连接数。

Nginx 具有很高的稳定性。其它 HTTP 服务器，当遇到访问的峰值，或者有人恶意发起慢速连接时，也很可能会导致服务器物理内存耗尽频繁交换，失去响应，只能重启服务器。例如当前 apache 一旦上到 200 个以上进程，web响应速度就明显非常缓慢了。而 Nginx 采取了分阶段资源分配技术，使得它的 CPU 与内存占用率非常低。Nginx 官方表示保持 10,000 个没有活动的连接，它只占 2.5M 内存，所以类似 DOS 这样的攻击对 Nginx 来说基本上是毫无用处的。就稳定性而言,Nginx 比 lighthttpd 更胜一筹。

Nginx 支持热部署。它的启动特别容易, 并且几乎可以做到 7*24 不间断运行，即使运行数个月也不需要重新启动。你还能够在不间断服务的情况下，对软件版本进行进行升级。

Nginx 采用 master-slave 模型,能够充分利用 SMP 的优势，且能够减少工作进程在磁盘 I/O 的阻塞延迟。当采用 select()/poll() 调用时，还可以限制每个进程的连接数。

Nginx 代码质量非常高，代码很规范，手法成熟，模块扩展也很容易。特别值得一提的是强大的 Upstream 与 Filter 链。Upstream 为诸如 reverse proxy,与其他服务器通信模块的编写奠定了很好的基础。而 Filter 链最酷的部分就是各个 filter 不必等待前一个 filter 执行完毕。它可以把前一个 filter 的输出做为当前 filter 的输入，这有点像 Unix 的管线。这意味着，一个模块可以开始压缩从后端服务器发送过来的请求，且可以在模块接收完后端服务器的整个请求之前把压缩流转向客户端。

Nginx 采用了一些 os 提供的最新特性如对 sendfile (Linux2.2+)，accept-filter (FreeBSD4.1+)，TCP_DEFER_ACCEPT (Linux 2.4+)的支持，从而大大提高了性能。

当然，Nginx 还很年轻，多多少少存在一些问题，比如：Nginx 是俄罗斯人创建，虽然前几年文档比较少，但是目前文档方面比较全面，英文资料居多，中文的资料也比较多，而且有专门的书籍和资料可供查找。

Nginx 的作者和社区都在不断的努力完善，我们有理由相信 Nginx 将继续以高速的增长率来分享轻量级 HTTP 服务器市场，会有一个更美好的未来。

2

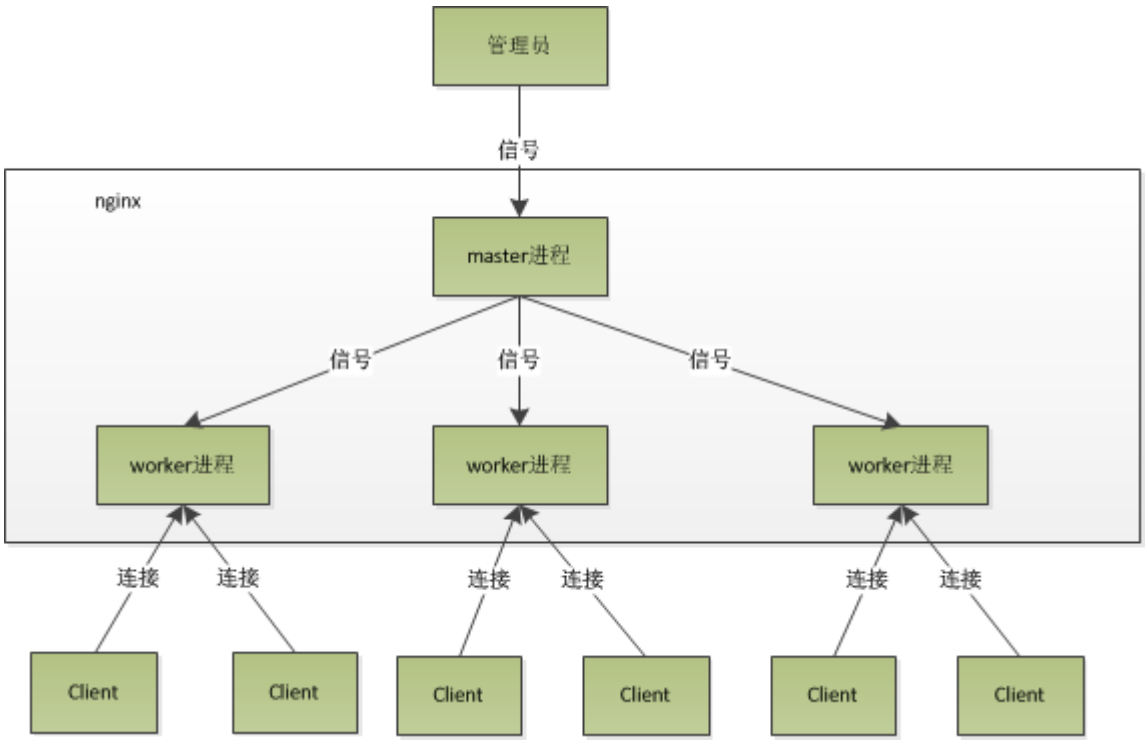
Nginx 平台初探

初探 Nginx 架构

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，才会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

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_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。

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。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。

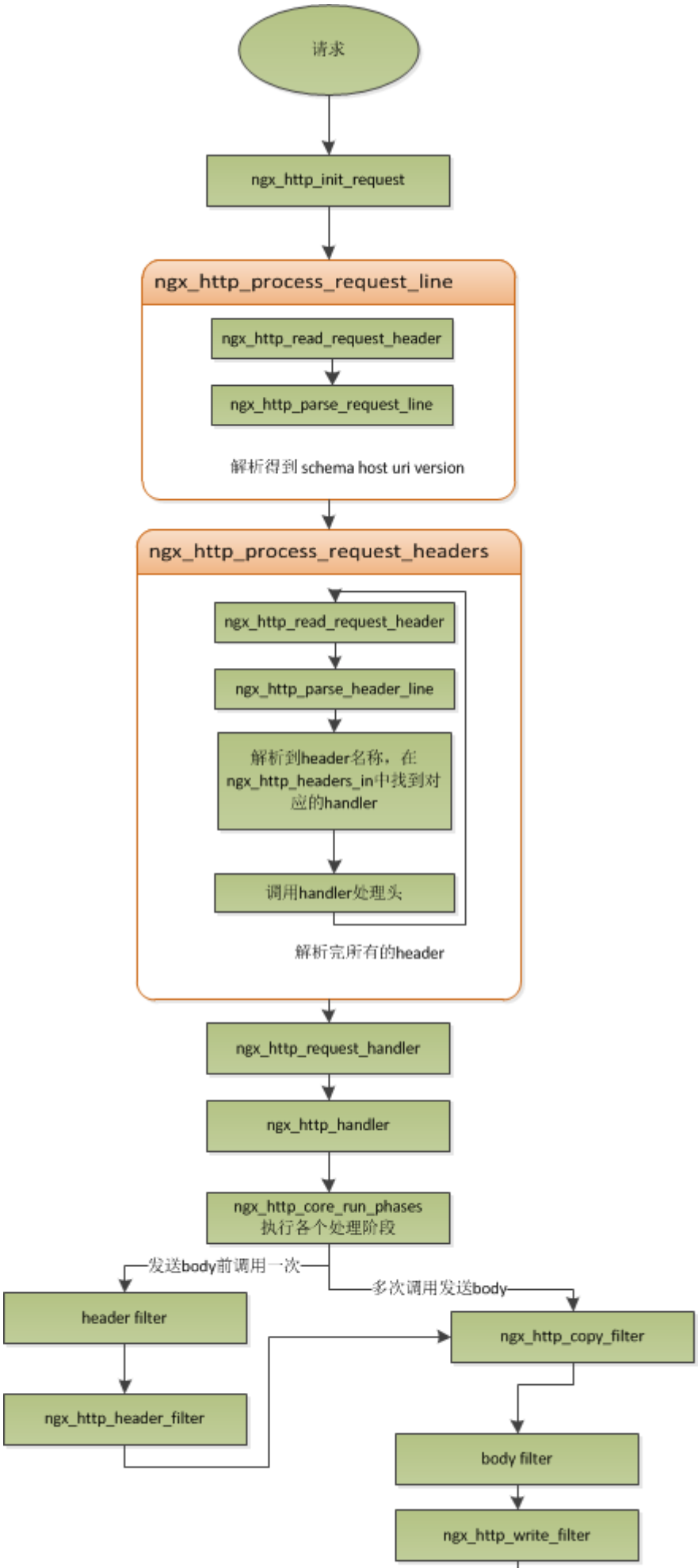
在解析完请求行后，Nginx 会设置读事件的 handler 为 ngx_http_process_request_headers，然后后续的请求就在 ngx_http_process_request_headers 中进行读取与解析。ngx_http_process_request_headers 函数用来读取请求头，跟请求行一样，还是调用 ngx_http_read_request_header 来读取请求头，调用 ngx_http_parse_header_line 来解析一行请求头，解析到的请求头会保存到 ngx_http_request_t 的域 headers_in 中，headers_in 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 48k，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:


请求
ngx_http_init_request
ngx_http_process_request_line
ngx_http_read_request_header
ngx_http_parse_request_line
解析得到 schema host uri version
ngx_http_process_request_headers
ngx_http_read_request_header
ngx_http_parse_header_line
解析到header名称，在ngx_http_headers_in中找到对应的handler
调用handler处理头
解析完所有的header
ngx_http_request_handler
ngx_http_handler
ngx_http_core_run_phases
执行各个处理阶段
发送body前调用一次
多次调用发送body
header filter
ngx_http_header_filter
ngx_http_copy_filter
body filter
ngx_http_write_filter

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 http 1.0，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，大分部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

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

在 Nginx 源码目录的 src/core 下面的 `ngx_string.h|c` 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，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");
u_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

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/ngx_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。

.. code:: c

```
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 hander 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

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。
- nalloc: 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

ngx_hash_t 是 Nginx 自己的 hash 表的实现。定义和实现位于 src/core/ngx_hash.h|c 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吧。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

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 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。
- names: 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。**
- 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

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

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/ngx_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，并将这个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。
- type: 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 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/ngx_http.c` 中的 ngx_http_server_names 函数。

ngx_chain_t

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/ngx_buf.h|c 。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

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。
- 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去，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

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（ngx_list_part_t）固定长度的数组的长度，即最多可以存放的元素个数。
- size: 每个元素所占用的内存大小。
- 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

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|c 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

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 footbar 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：

- 操作系统提供的机制（例如 epoll, kqueue 等）产生相关的事件。
- 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。
- 处理 request 的 header 和 body。
- 产生响应，并发送回客户端。
- 完成 request 的处理。
- 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。

- 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）。
- 处理请求头。
- 处理请求体。
- 如果有的话，调用与此请求（URL 或者 Location）关联的 handler。
- 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：

- 获取 location 配置。
- 产生适当的响应。
- 发送 response header。
- 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:

- 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。
- 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。
- 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。
- 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）。
- 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 fiter 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：

- server-side includes。
- XSLT filtering。
- 图像缩放之类的。
- 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

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/ngx_conf_file.h` 中。

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

name: 配置指令的名称。

type: 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。

set: 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_MOUDLE，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。
- merge_srv_conf: 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。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，这个函数负责对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. 另外一个是，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。

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_pcalloc(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_pcalloc(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/ngx_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 源代码的情况下来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/ngx_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/ngx_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:
#if (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_pcalloc(r->pool, sizeof(ngx_buf_t));
if (b == NULL) {
    return NGX_HTTP_INTERNAL_SERVER_ERROR;
}

b->file = ngx_pcalloc(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 index 文件，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/ngx_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/ngx_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

过滤模块简介

执行时间和内容

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

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。一般情况下，第三方过滤模块的 config 文件会将模块名追加到变量 HTTP_AUX_FILTER_MODULES 中，此时该模块只能加入到 copy_filter 和 headers_filter 模块之间执行。

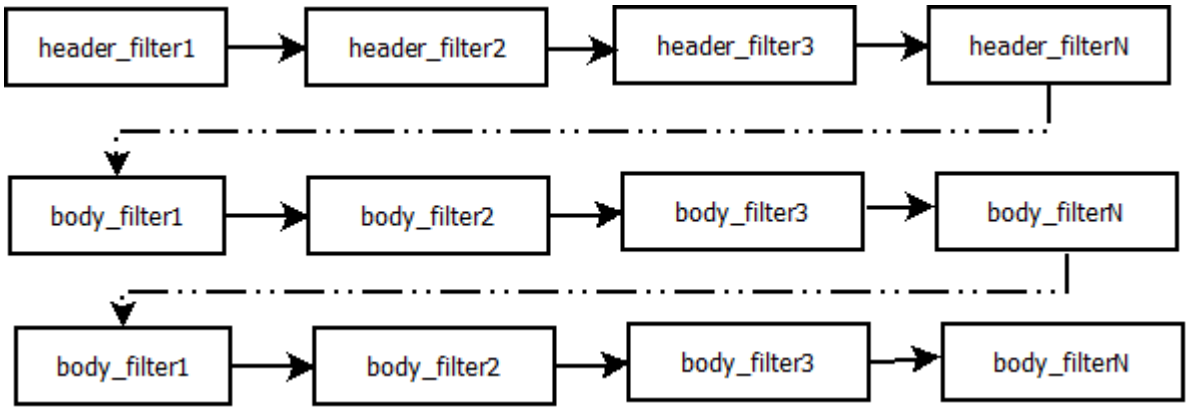
Nginx 执行的时候是怎么按照次序依次来执行各个过滤模块呢？它采用了一种很隐晦的方法，即通过局部的全局变量。比如，在每个 filter 模块，很可能看到如下代码：

```
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 fitler 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

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

相关结构体

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;
};
```

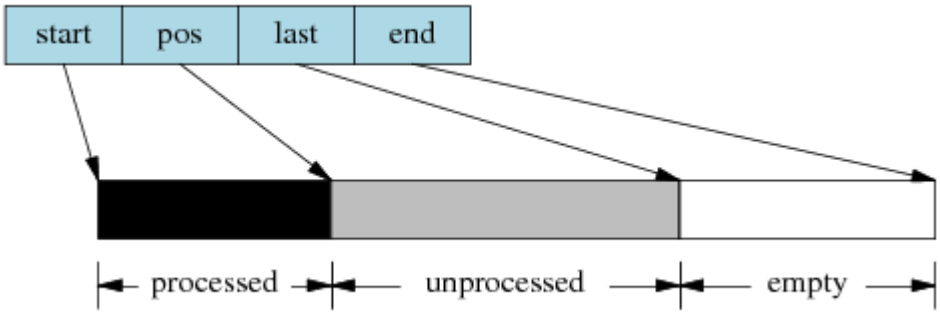
在过滤模块中，所有输出的内容都是通过一条单向链表所组成。这种单向链表的设计，正好应和了 Nginx 流式的输出模式。每次 Nginx 都是读到一部分的内容，就放到链表，然后输出出去。这种设计的好处是简单，非阻塞，但是相应的问题就是跨链表的内容操作非常麻烦，如果需要跨链表，很多时候都只能缓存链表的内容。

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

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

filter module	description
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

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

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_fitler 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 writev 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

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

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

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。

5

upstream 模块

upstream 模块简介

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：

SN	描述
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 upstreamserver 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。

SN	描述
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;
    }
```

1. 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;
```

1. 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;
```

1. 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;
```

1. 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;
```

1. 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。

- 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。
- ngx_http_memcached_reinit_request：无需初始化。
- ngx_http_memcached_abort_request：无需额外操作。

- ngx_http_memcached_finalize_request：无需额外操作。
- 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。

- 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。
- 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

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。

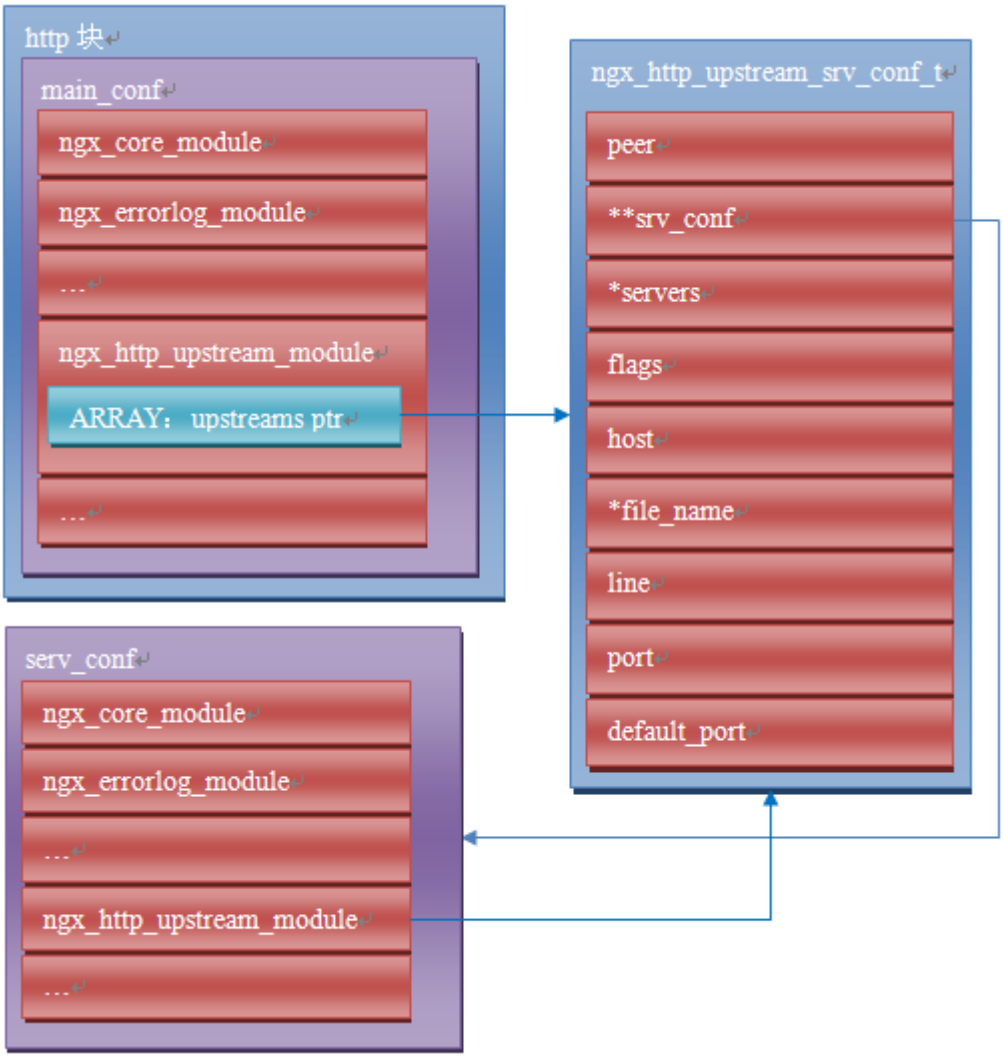
设置 uscf->flags

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；
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 = &iphp->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?

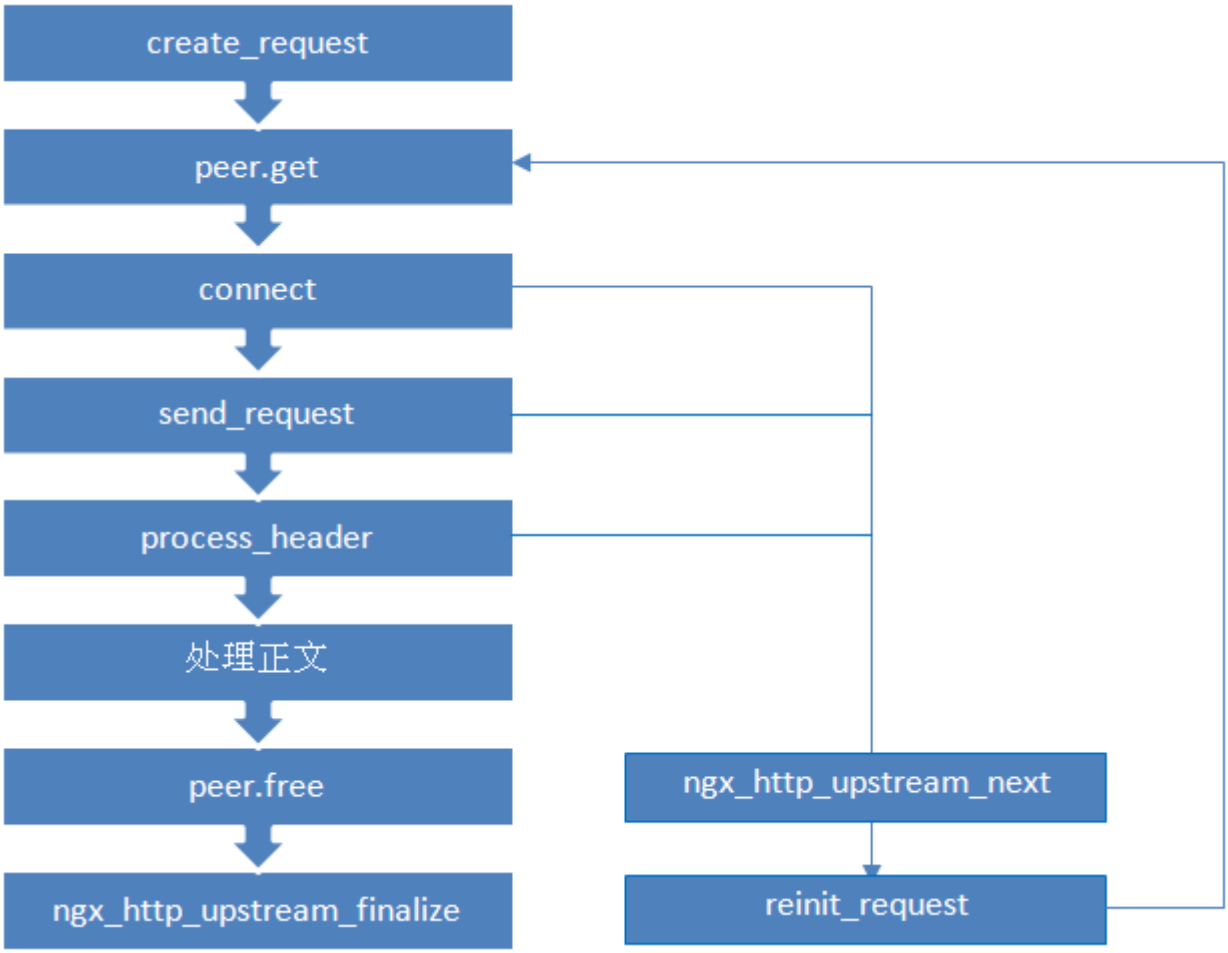
A: 使用后端 keepalive 连接的时候，连接在使用完以后并不关闭，而是存放在一个队列中，新的请求只需要从队列中取出连接，这些连接都是已经准备好的。

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。

6

其他模块

core 模块

Nginx 的启动模块

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:

- 时间、正则、错误日志、ssl 等初始化
- 读入命令行参数
- OS 相关初始化
- 读入并解析配置
- 核心模块初始化
- 创建各种暂时文件和目录
- 创建共享内存
- 打开 listen 的端口
- 所有模块初始化
- 启动 worker 进程

event 模块

event 的类型和功能

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 EPOLLLT（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

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

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(logn)，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。