
元始天尊

目錄

前言

本书为自学快速搭建Nginx环境，并提供基于Nginx服务解决方案等。

鉴于新人快速入门。

> 作者: 刘丹冰
>
> 传智播客C++学院

1 Nginx简介

Nginx由来

2004 年10月4号 俄罗斯人创作

当时的web server满足不了需求，阿帕奇解决不了，单枪匹马开发nginx。

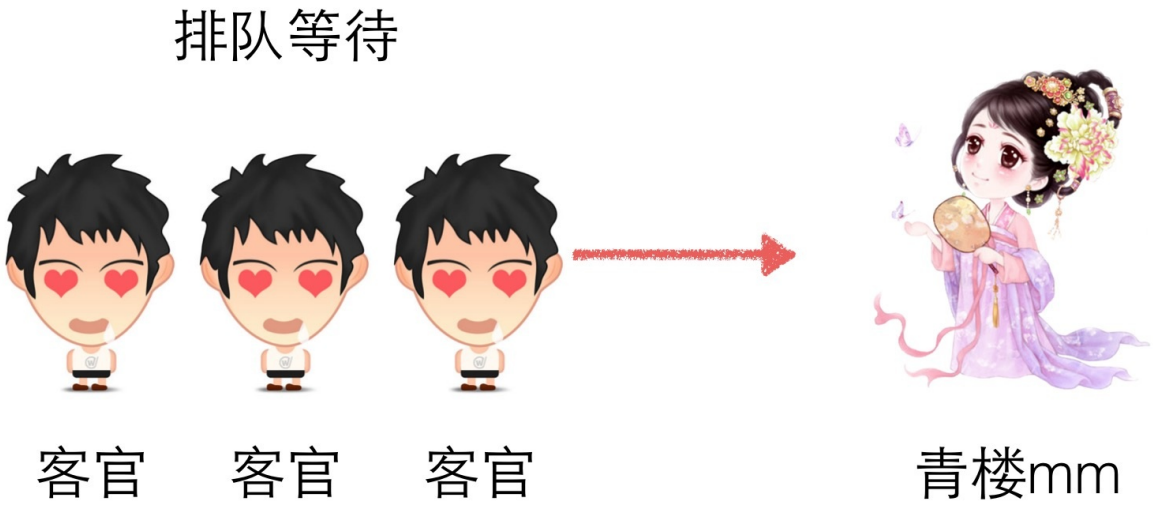
Nginx三大功能

- **web**服务器

> 作为web服务器，Nginx是一个轻量级，而且能够处理的并发量更大。

- 反向代理服务器

> 何为反向代理服务器？



一个青楼mm同意时刻只能接待一个用户。

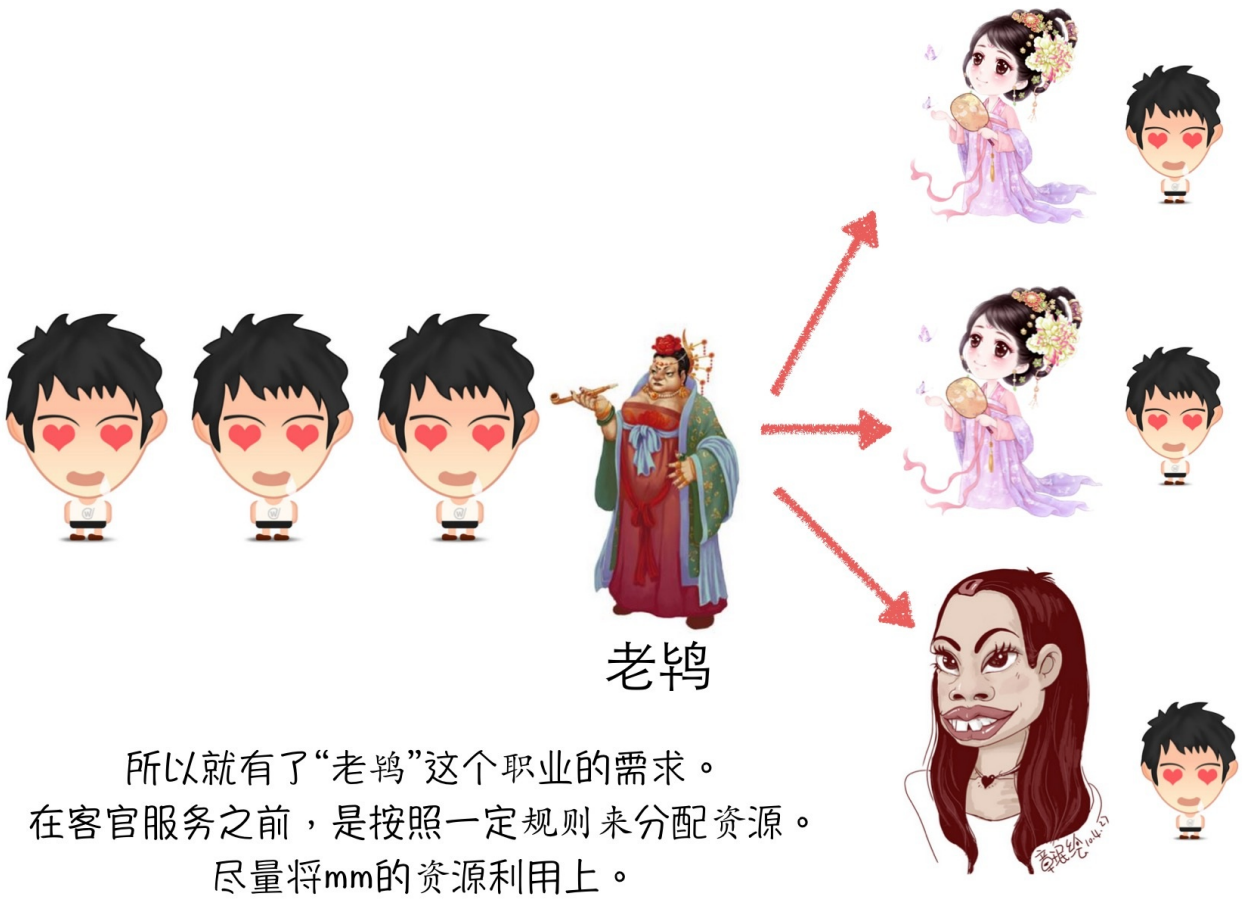
排队等待

客官　客官　客官

青楼mm

即使增加了服务人员，
但有时候依然无法平均分配用户。

青楼新来的mm

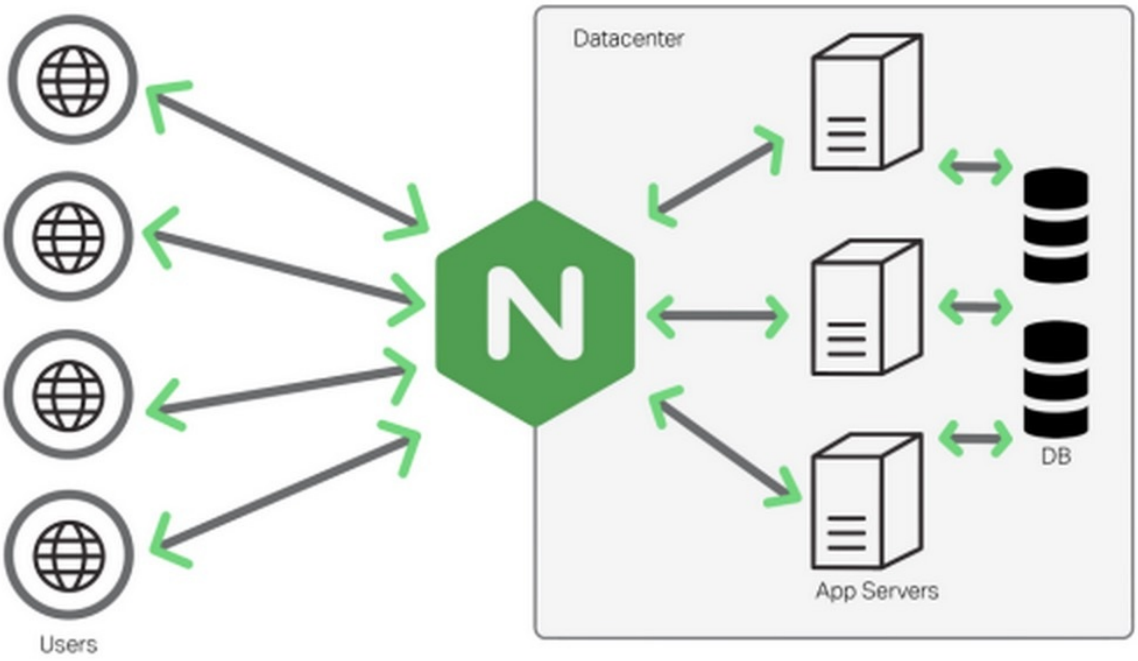


上述例子中，

客官----------客户端请求

青楼mm---------web服务器

老鸨----------反向代理服务器（负载均衡）



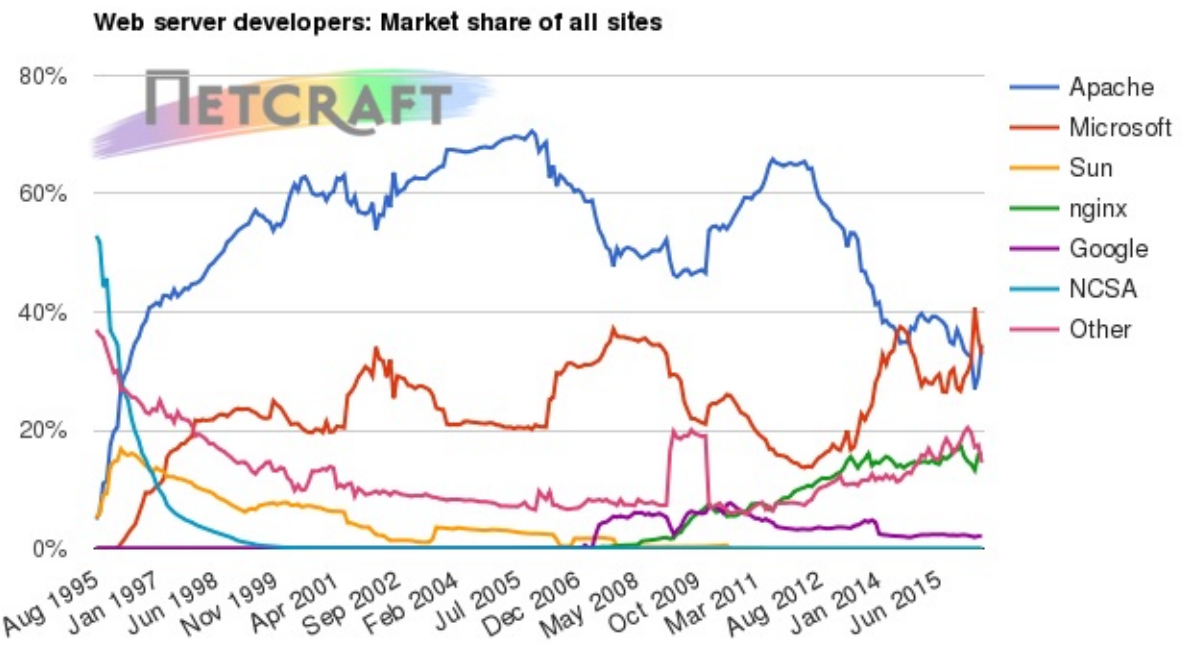
Nginx就是干这个活的。通过将用户端的请求，透明的转送给应用服务器。这样所有的客户端只需要访问同一个Nginx服务器就可以了。然后Nginx本身内部会有一些负载均衡的算法和规则来平均给身后的Server分发链接，达到每个服务器负载量均衡。

- 邮件服务器

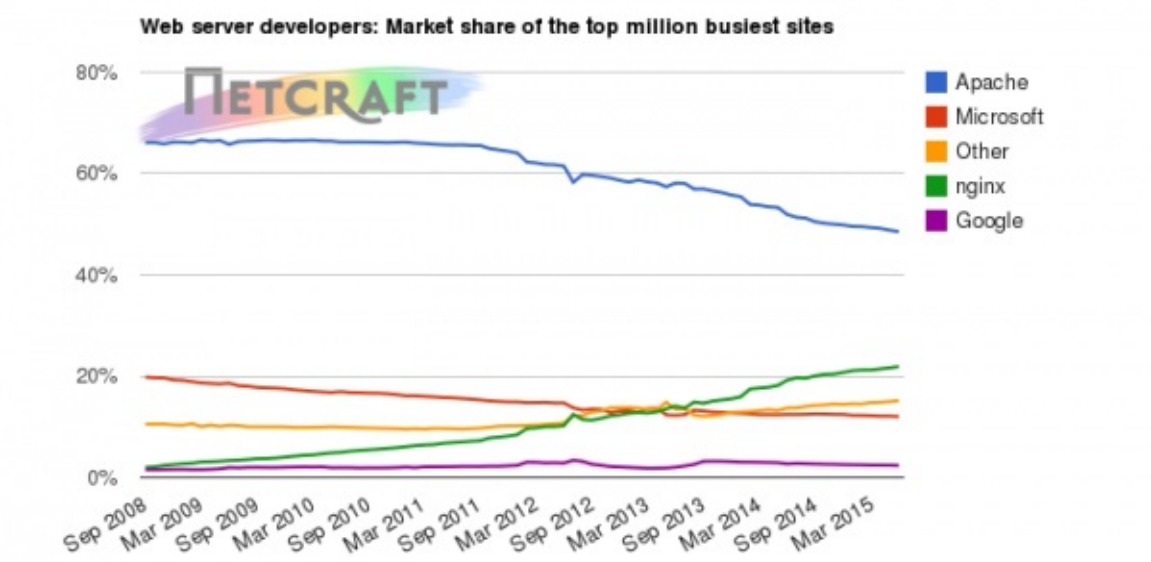
Nginx也可用充当一个IMAP/POP3/SMTP服务器。

1.1 netcraft 统计

市场占有率

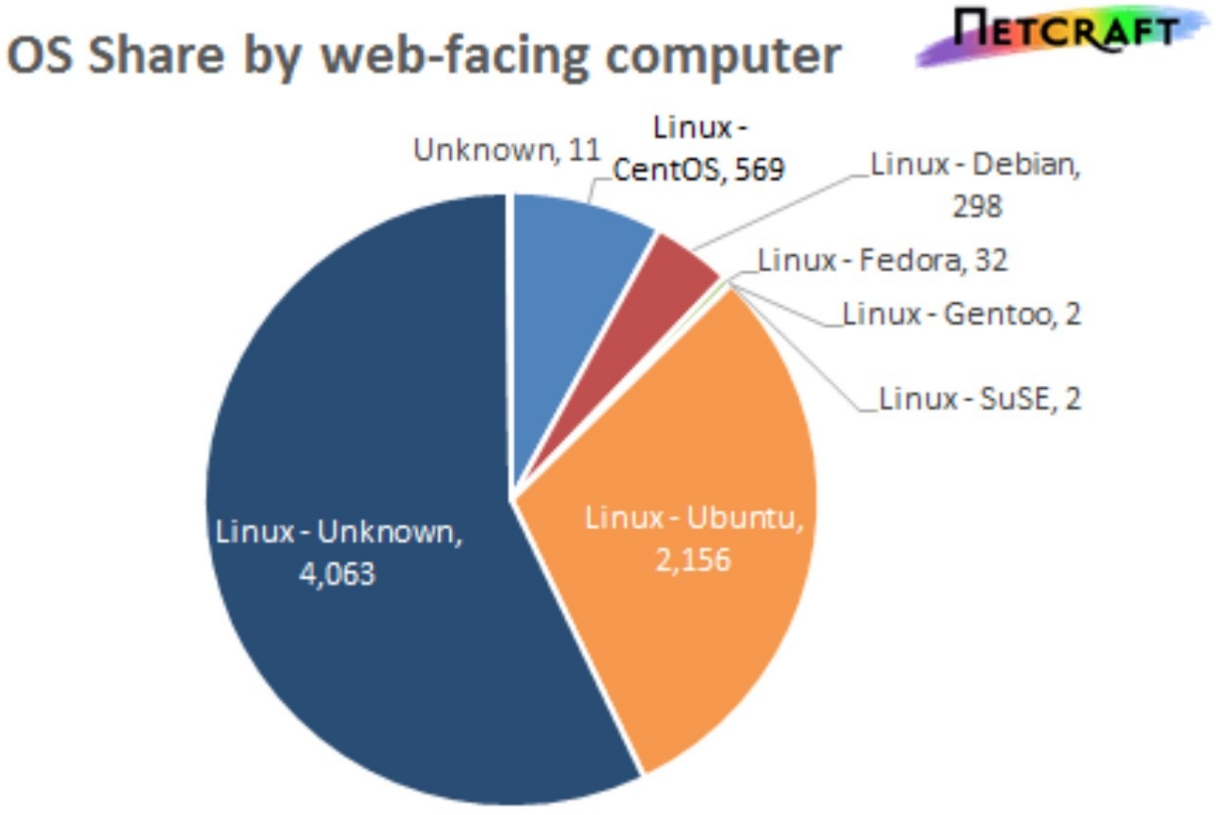


百万级别繁忙服务器占有率



Developer	May 2015	Percent	June 2015	Percent	Change
Apache	488,313	48.83%	485,315	48.53%	-0.30
nginx	216,433	21.64%	218,913	21.89%	0.25
Microsoft	121,274	12.13%	120,239	12.02%	-0.10
Google	24,170	2.42%	23,751	2.38%	-0.04

Ngnix部署操作系统



1.2 Nginx优势

1. 更快 。正常情况下单次请求得到更快的响应;高峰期(数以万计的并发时)nginx可以比其它web服务器更快的响应请求。

2. 高扩展性 。低耦合设计的模块组成,丰富的第三方模块支持。

3. 高可靠性 。经过大批网站检验,每个worker进程相对独立,master进程在一个worker 进程出错时,可以快速开启新的worker进程提供服务。

4. 低内存消耗 。一般情况下,10000个非活跃的HTTP Keep-Alive连接在nginx中仅消耗 2.5M内存,这是nginx支持高并发的基础。

5. 单机支持10万以上的并发连接 。取决于内存,10万远未封顶。

6. 热部署 。master和worker的分离设计,可实现7x24小时不间断服务的前提下,升级nginx可执行文件,当然也支持更新配置项和日志文件。

7. 最自由的BSD许可协议 。BSD许可协议允许用户免费使用nginx,修改nginx源码,然后 再发布。这吸引了无数的开发者继续为nginx贡献智慧。

1.3 Nginx的下载包和安装

http://nginx.org

是Nginx维护包的官方网站。

`注意`：这里企业一般使用的都是稳定版本，`stable`

或者

http://nginx.org/download/

1.4 有关Nginx文档

Nginx官方网站 http://nginx.org/en/docs/

淘宝团队翻译网站 http://tengine.taobao.org/nginx_docs/cn/docs/

2 快速安装及配置web服务器

2.1 准备工作

nginx可以使用各平台的默认包来安装，本文是介绍使用源码编译安装，包括具体的编译参数信息。

> 正式开始前，编译环境gcc g++ 开发库之类的需要提前装好，这里默认你已经装好。

- ububtu平台编译环境可以使用以下指令

```
apt-get install build-essential
apt-get install libtool
```

- centos平台编译环境使用如下指令

安装make

```
yum -y install gcc automake autoconf libtool make
```

安装g++

```
yum install gcc gcc-c++
```

2.2 正式安装

一般我们都需要先装pcre, zlib，前者为了重写rewrite，后者为了gzip压缩。

2.2.1 选定源码目录

可以是任何目录，本文选定的是/usr/local/src

2.2.2 安装PCRE库

ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/ 下载最新的PCRE源码包。

使用下面命令下载编译和安装PCRE包：

```
cd /usr/local/src
wget ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/pcre-8.39.tar.gz
tar -zxvf pcre-8.34.tar.gz
cd pcre-8.34
./configure
make
make install
```

2.2.3 安装zlib库

http://zlib.net/zlib-1.2.8.tar.gz 下载最新的 zlib 源码包，使用下面命令下载编译和安装 zlib包：

```
cd /usr/local/src

wget http://zlib.net/zlib-1.2.8.tar.gz
tar -zxvf zlib-1.2.8.tar.gz
cd zlib-1.2.8
./configure
make
make install
```

2.2.4 安装ssl

```
wget http://www.openssl.org/source/openssl-1.0.1t.tar.gz
tar -zxvf openssl-1.0.1c.tar.gz
./config  --prefix=/usr/local --openssldir=/usr/local/openssl

make depend
make
sudo make install

//若要生成libssl.so动态库文件 需要如下make
make clean
./config shared --prefix=/usr/local --openssldir=/usr/local/open
ssl
make depend
make
sudo make install
```

2.2.5 安装nginx

```
cd /usr/local/src
wget http://nginx.org/download/nginx-1.10.1.tar.gz
tar -zxvf nginx-1.10.1.tar.gz
cd nginx-1.10.1

./configure --sbin-path=/usr/local/nginx/nginx
--conf-path=/usr/local/nginx/nginx.conf
--pid-path=/usr/local/nginx/nginx.pid
--with-http_ssl_module
--with-pcre=/usr/local/src/pcre-8.39
--with-zlib=/usr/local/src/zlib-1.2.8
--with-openssl=/usr/local/openssl

make
make install
```

--with-pcre=/usr/src/pcre-8.34 指的是pcre-8.34 的源码路径。 --with-zlib=/usr/src/zlib-1.2.7 指的是zlib-1.2.7 的源码路径。

安装成功后 /usr/local/nginx 目录下如下:

```
fastcgi.conf
koi-win
nginx.conf.default
fastcgi.conf.default
logs
scgi_params
fastcgi_params
mime.types
scgi_params.default
fastcgi_params.default  mime.types.default
uwsgi_params
html
nginx
uwsgi_params.default
koi-utf
nginx.conf
win-utf
```

2.2.6 启动

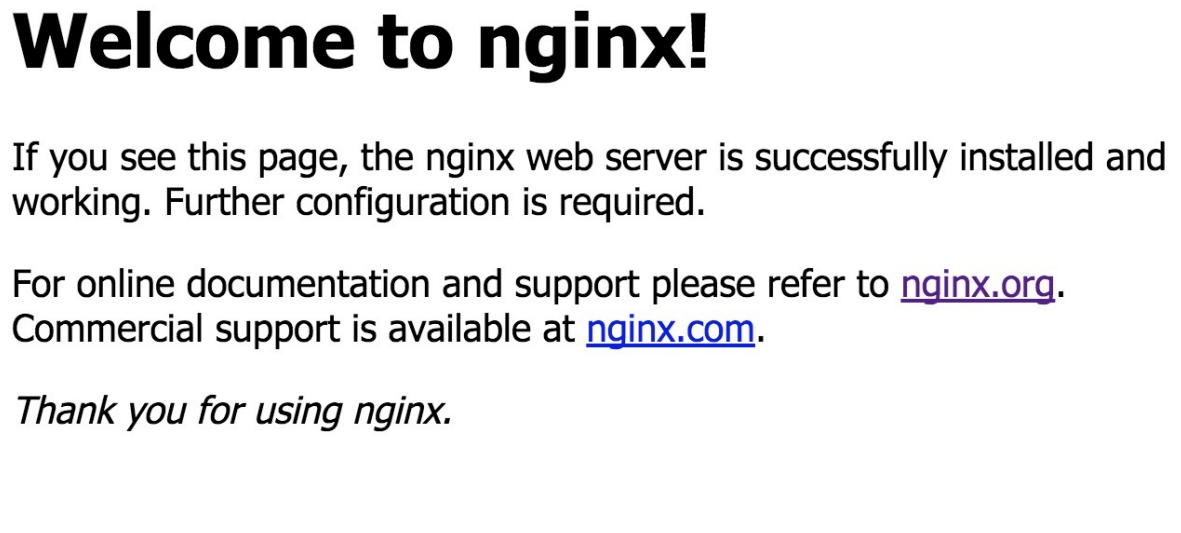
确保系统的 80 端口没被其他程序占用，运行/usr/local/nginx/nginx 命令来启动 Nginx。

```
netstat -ano|grep 80
```

如果查不到结果后执行，有结果则忽略此步骤（ubuntu下必须用sudo启动，不然只能在前台运行）

```
sudo /usr/local/nginx/nginx
```

打开浏览器访问此机器的 IP，如果浏览器出现 Welcome to nginx! 则表示 Nginx 已经安装并运行成功。



Nginx会被安装在**/usr/local/nginx**目录下**(**也可以使用参数**--prefix=**指定自己需要的位置**)**, 安装成功后 **/usr/local/nginx** 目录下有四个子目录分别是: `conf` 、 `html` 、 `logs` 、 `sbin` 。其中 **Nginx** 的配置文件存放于 conf/nginx.conf, **bin**文件是位于 **sbin** 目录下的 **nginx** 文件。

> 确保系统的 80 端口没被其他程序占用,运行 /usr/local/nginx/nginx 命令来启动 Nginx, 打开浏览器访问此机器的 IP,如果浏览器出现 Welcome to nginx! 则表示 Nginx 已经安装并运行成功

2.2.7 通用配置

```
#运行用户
```

```
user nobody;
#启动进程,通常设置成和cpu的数量相等
worker_processes  1;

#全局错误日志及PID文件
#error_log  logs/error.log;
#error_log  logs/error.log  notice;
#error_log  logs/error.log  info;

#pid        logs/nginx.pid;

#工作模式及连接数上限
events {
    #epoll是多路复用IO(I/O Multiplexing)中的一种方式,
    #仅用于linux2.6以上内核,可以大大提高nginx的性能
    use   epoll;

    #单个后台worker process进程的最大并发链接数
    worker_connections  1024;

    # 并发总数是 worker_processes 和 worker_connections 的乘积
    # 即 max_clients = worker_processes * worker_connections
    # 在设置了反向代理的情况下，max_clients = worker_processes * wor
ker_connections / 4  为什么
    # 为什么上面反向代理要除以4，应该说是一个经验值
    # 根据以上条件，正常情况下的Nginx Server可以应付的最大连接数为：4 *
8000 = 32000
    # worker_connections 值的设置跟物理内存大小有关
    # 因为并发受IO约束，max_clients的值须小于系统可以打开的最大文件数
    # 而系统可以打开的最大文件数和内存大小成正比，一般1GB内存的机器上可以打
开的文件数大约是10万左右
    # 我们来看看360M内存的VPS可以打开的文件句柄数是多少：
    # $ cat /proc/sys/fs/file-max
    # 输出 34336
    # 32000 < 34336，即并发连接总数小于系统可以打开的文件句柄总数，这样就
在操作系统可以承受的范围之内
    # 所以，worker_connections 的值需根据 worker_processes 进程数目
和系统可以打开的最大文件总数进行适当地进行设置
    # 使得并发总数小于操作系统可以打开的最大文件数目
    # 其实质也就是根据主机的物理CPU和内存进行配置
```

```
    # 当然，理论上的并发总数可能会和实际有所偏差，因为主机还有其他的工作进程
需要消耗系统资源。
    # ulimit -SHn 65535

}


http {
    #设定mime类型,类型由mime.type文件定义
    include    mime.types;
    default_type  application/octet-stream;
    #设定日志格式
    log_format  main  '$remote_addr - $remote_user [$time_local]
 "$request" '
                      '$status $body_bytes_sent "$http_referer"
'
                      '"$http_user_agent" "$http_x_forwarded_for
"';

    access_log  logs/access.log  main;

    #sendfile 指令指定 nginx 是否调用 sendfile 函数（zero copy 方式）
来输出文件，
    #对于普通应用，必须设为 on,
    #如果用来进行下载等应用磁盘IO重负载应用，可设置为 off，
    #以平衡磁盘与网络I/O处理速度，降低系统的uptime.
    sendfile     on;
    #tcp_nopush     on;

    #连接超时时间
    #keepalive_timeout  0;
    keepalive_timeout  65;
    tcp_nodelay     on;

    #开启gzip压缩
    gzip  on;
    gzip_disable "MSIE [1-6].";

    #设定请求缓冲
    client_header_buffer_size    128k;
```

```
    large_client_header_buffers  4 128k;


    #设定虚拟主机配置
    server {
        #侦听80端口
        listen    80;

        #也可以设置为 合法域名
        server_name  localhost;

        #定义服务器的默认网站根目录位置
        root html;

        #设定本虚拟主机的访问日志
        access_log  logs/nginx.access.log  main;

        #默认请求
        location / {
            #定义如果访问根目录的请求目录
            root html;
            #定义首页索引文件的名称
            index index.php index.html index.htm;

        }

        # 定义错误提示页面
        error_page   500 502 503 504 /50x.html;
        location = /50x.html {
        }

        #静态文件，nginx自己处理
        location ~ ^/(images|javascript|js|css|flash|media|stati
c)/ {

            #过期30天，静态文件不怎么更新，过期可以设大一点，
            #如果频繁更新，则可以设置得小一点。
            expires 30d;
        }
```

```
        #PHP 脚本请求全部转发到 FastCGI处理. 使用FastCGI默认配置.
        location ~ .php$ {
            fastcgi_pass 127.0.0.1:9000;
            fastcgi_index index.php;
            fastcgi_param  SCRIPT_FILENAME  $document_root$fastc
gi_script_name;
            include fastcgi_params;
        }

        #禁止访问 .htxxx 文件
            location ~ /.ht {
            deny all;
        }

    }
}
```

2.3 启动与关闭

2.3.1 重启Nginx

```
sudo /usr/local/sbin/nginx -s reload
```

2.3.2 关闭Nginx

快速停止服务

```
sudo /usr/local/sbin/nginx -s stop
```

优雅停止服务

```
sudo /usr/local/sbin/nginx -s quit #kill -s SIGQUIT pid_master
kill -s SIGWINCH pid_master
```

3 Nginx工作原理

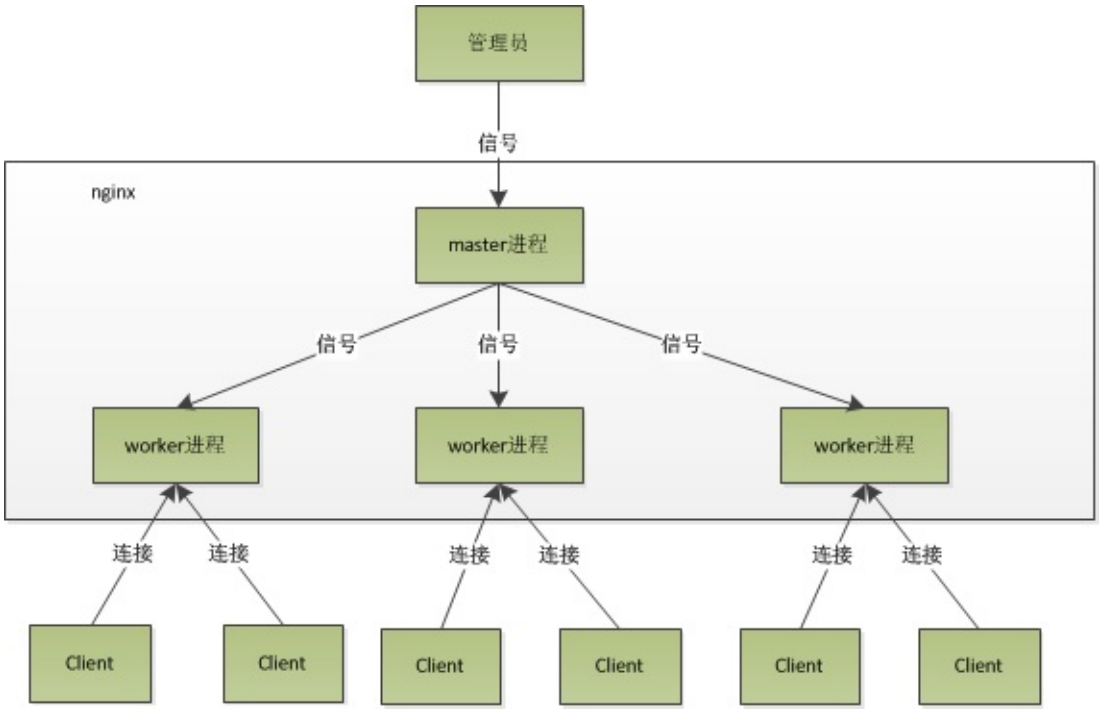
众所周知，nginx性能高，而nginx的高性能与其架构是分不开的。那么nginx究竟是怎么样的呢？这一节我们先来初识一下nginx框架吧。

3.1 Nginx进程模型

nginx在启动后，在unix系统中会以daemon的方式在后台运行，后台进程包含一个master进程和多个worker进程。

> 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。

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：



nginx进程模型的好处：

1. 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。
2. 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。
3. 如果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。

3.2 Nginx的事件处理过程

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？

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，当我们写我们可以用一段伪代码来总结一下nginx的事件处理模型：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);
}
```

3.3 Nginx配置系统

Nginx的配置系统由一个主配置文件和一些辅助配置文件构成，这些配置文件默认在 `/usr/local/nginx/` 目录下。

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：

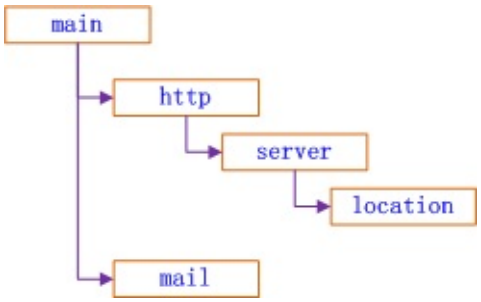
```
error_page   500 502 503 504  /50x.html;
```

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其进行分类，可以分成多个作用域或指令上下文，指令上下文层次关系如下：



- `main`：Nginx在运行时与具体业务功能无关的参数，比如工作进程数、运行身份等。
- `http`：与提供http服务相关的参数，比如keepalive、gzip等。
- `server`：http服务上支持若干虚拟机，每个虚拟机一个对应的server配置项，配置 项里包含该虚拟机相关的配置。
- `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。

> 更多配置信息可以参看Nginx安装后的缺省配置文件/usr/local/nginx/nginx.conf.default。

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。

```
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：

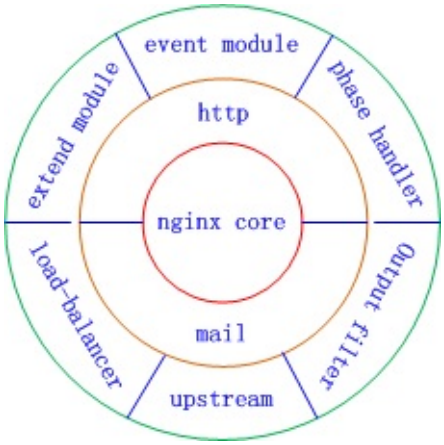
- index
- root

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.

3.4 Nginx的模块化体系

一、模块体系

Nginx的内部结构是由核心部分和一系列功能模块组成的，这样可以使得每个模块的功能相对简单，便于对系统进行功能扩展，各模块之间的关系如下图：



nginx core实现了底层的通讯协议，为其它模块和Nginx进程构建了基本的运行时环境，并且构建了其它各模块的协作基础。

http模块和mail模块位于nginx core和各功能模块的中间层，这2个模块在nginx core之上实现了另外一层抽象，分别处理与http协议和email相关协议（SMTP/IMAP/POP3）有关的事件，并且确保这些事件能被以正确的顺序调用其它的一些功能模块。

nginx功能模块基本上分为如下几种类型：

（1）event module：搭建了独立于操作系统的事件处理机制的框架，以及提供了各具体事件的处理，包括ngx_event_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。

（2）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_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。

（3）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，比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。

（4）upstream：实现反向代理功能，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。

（5）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。

（6）extend module：根据特定业务需要编写的第三方模块。

二、请求处理

下面将会以http请求处理为例来说明请求、配置和模块是如何串起来的。

当Nginx读取到一个HTTP Request的header时，首先查找与这个请求关联的虚拟主机的配置，如果找到了则这个请求将会经历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response，Nginx必须把这个请求交给一个合适的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perl、flv、proxy_pass、mp4等。

如果一个request对应的location并没有直接配置的content handler，那么Nginx依次作如下尝试：

（1）如果一个location里面有配置random_index on，那么随机选择一个文件发送给客户端。

（2）如果一个location里面有配置index指令，那么发送index指令指定的文件给客户端。

（3）如果一个location里面有配置autoindex on，那么就发送请求地址对应的服务端路径下的文件列表给客户端。

（4）如果这个request对应的location上有设置gzip_static on，那么就查找是否有对应的.gz文件存在，如果有的话，就发送这个给客户端（客户端支持gzip的情况下）。

（5）请求的URI如果对应一个静态文件，static module就发送静态文件的内容到客户端。

内容产生阶段完成以后，生成的输出会被传递到filter模块去进行处理。filter模块也是与location相关的。所有的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。接下来就可以发送response给客户端了。

4 Nginx相关配置

4.1 重启Nginx

```
#sudo /usr/local/sbin/nginx -s reload
```

4.2 关闭Nginx

```
快速停止服务
#sudo /usr/local/sbin/nginx -s stop

优雅停止服务
#sudo /usr/local/sbin/nginx -s quit #kill -s SIGQUIT pid_master
#kill -s SIGWINCH pid_master
```

4.3 Nginx进程之间关系

一个master进程来管理多个work进程.

work进程数量和CPU的核数相同(进程间切换的代价最小)

4.4 Nginx配置通用语法

块配置项

```
块配置项由一个块配置项名和一对大括号组成. 比如
    events {
        use epoll;
    }
nginx.conf中的events,http,server,location,upstream等都是块配置项 块配置项可以嵌套,内嵌块直接继承外层块.
```

块配置项的语法格式

基本格式:
配置项名 配置项值1 配置项值2 ...;
配置项目必须是nginx的某一模块想要处理的,否则判定为非法配置项名. 配置项值可以是数字,字符串包括正则表达式,可能有多个值. 每行配置的末尾以分好';'结束

配置项的单位

```
以'#'字符开始 的一行视为注释

#pid logs/nginx.pid

指定空间大小 单位包括 K k 千字节(KB),M m 兆字节(MB)
  gzip_buffers 4 8k;
  client_max_body_size 64M;
指定时间大小 单位包括 ms,s,s,m,h,d,w,M,y; expires 10m;#有效期为10s
```

配置项中使用变量

```
$varname 比如:
log_format main '$remote_addr - $remote_user'
```

4.5 Nginx服务基本配置

Nginx服务在运行时,至少需要加载几个核心模块和一个事件类模块. 这些模块所支持的配置统称为基本配置.
主要分为4大类:

1用于调试定位为题的配置项
2正常运行的必备配置项
3优化性能的配置项
4事件类配置项

用以调试和定位问题的配置块

是否以守护进程方式运行

```
daemon on|off;
默认为on
如果调试阶段 可以设置为off 以 前台进程方式运行 这样便于跟踪调试Nginx
```

是否以master/worker方式工作

```
master_process on | off;
默认为on
如果调试阶段 可以设置为off 以 master进程自身来响应请求 这样便于跟踪调试Ngi
nx
```

errorr日志的设置

```
error_log /path/file level;
#第一个项为设置为error日志的路径和文件名
#第二项为等级 有debug,info,notice,warn,error,crit,alert,emerg 默认
为 logs/error.log error;
当第一项设置为 /dev/null 表示忽略任何日志
当设置为 stderr 这样错误日志会输出到标准错误文件中.
第二项的等级 自左向右依次增加.

最后应该保证输出日志的硬盘空间应当足够使用

**设置成debug模式的时候,需要在configure时 加上--with-debug 参数
```

仅对指定的客户端输出**debug**级别的日志

```
debug_connection IP[/port]
由于该配置属于事件类配置,需要放置在events{...}才有效 例如:
events{
    debug_connection 192.168.1.100;
    debug_connection 192.168.1.100/24;
}
仅对以上设置的IP才设置成debug级别的日志,其他请求沿用error_log 配置的级别
```

限制**coredump**核心转储文件的大小

```
worker_rlimit_core_size size;
以size来限制coredump文件的大小.
```

指定**coredump**文件的位置

```
working_directory path;
path指定coredump文件的位置
需要保证path路径有足够的写入权限和足够的使用空间.
```

正常运行配置项

引入其他配置文件

```
include /path/file;
include配置项可以将其他配置文件引入到当前的nginx.conf文件中,参数可以是绝对路径和相对(conf/)路径
include mine.types;
include vhost/*.conf
```

pid文件的位置

```
pid path/file
logs/nginx/pid
保存master进程ID的pid文件夹的存放路径
应该确保nginx在相应的目录中有创建pid文件的权限.
```

Nginx worker进程运行的用户和用户组

```
user username [groupname];
user nobody nobody;
user用于设置master进程启动后,fork出的worker子进程运行在哪个用户和用户组下.
当设置username没有设置groupname,则默认username与groupname相同.
```

指定**worker**进程可以打开的最大文件句柄描述符个数

```
worker_rlimit_nofile limit_num;
设置一个worker进程可以打开的最大文件句柄数.(应该大于最大连接数)
```

限制信号队列长度

```
worker_rlimit_sigpending limit_num;
设置每个用户发往Nginx信号队列的大小.多的将丢弃
```

优化性能配置项

Nginx worker进程的个数

```
worker_process number;
默认为1
worker进程的数量直接影响性能.合适的worker进程数量和业务息息相关.
worker进程是单线程的进程,如果确认各模块中不会出现阻塞调用那么number设置为cpu的核数
如果有可能出现阻塞调用,number设置的比cpu核数大一点.

多worker进程可以充分利用多核系统架构,如果worker进程相比CPU数量太多会增加进程间切换的消耗.
```

绑定**Nginx worker**进程到指定的**CPU**内核

```
**仅对Linux有效
worker_cpu_affinity cpumask[cpumask...]
可防止多个进程抢占同一核心

worker_processes 4;
worker_cpu_affinity 1000 0100 0010 0001;
worker_processes 2;
worker_cpu_affinity 10 01;
```

系统调用**getimeofday()**的执行频率

```
默认 timer_resolution t;

例如 timer_resolution 100ms;表示至少每100ms才调用一次gettimeofday()
目前大多数内核中,花销只是一次vsyscall()仅对共享内存页中的数据做访问.一般可以不适用这个配置
```

事件类配置项

是否打开**accept**锁

```
accept是Nginx负载均衡锁.这把锁可以让多个work进程轮流,有序的与新的客户端建立TCP连接.

默认是打开的.

如果配置关闭,建立TCP连接耗时会更短.但是多个worker之间负载不会均衡.
```

lock文件的路径

```
lock_file path/file
默认 logs/nginx.lock
accept锁可能需要这个lock文件,如果accept锁配置关闭那么lock_file配置无效
如果accept锁配置打开且由于操作系统和编译器等因素导致Nginx不支持原子锁,将利用文件锁实现accept.
```

使用**accept**锁后到真正建立连接之间的延迟时间

```
accept_mutex_delay numberms;
默认500ms;

一个worker进程试图获取到accept锁失败,经过number ms时间再次试图获取accept锁.
```

批量建立连接

```
multi_accept [on|off]
默认off

当时间模型通知有新连接时,尽量对本次调度中客户端发起的TCP请求都建立连接.
```

选择事件模型

```
use [kqueue | rtsig epoll | /dev/poll | select |poll |eventport]

默认 Nginx会自动选择最适合的事件模型.
对于Linux来说,可以供选择的时间驱动模型有select,poll,epoll的三种.
```

每个**worker**的最大连接数

```
worker_connection number;
定义每个worker进程可以同时处理的最大连接数
```

使用HTTP核心模块配置一个静态WEB服务器

```
所有的HTTP配置项都必须直属于http块,server块,location块,upstream块,if块
.

直属于指的是配置项直接所属的大括号对应的设置块
```

虚拟主机与请求的分发

```
由于IP有限,存在多个主机域名对应同一个IP地址的情况.在nginx.conf中可以通过s
erver块 来设置server_name定义虚拟主机.
每个server块就是一个虚拟主机,只处理与之相应的主机域名请求,
这样一套服务器上的Nginx就能以不同的方式处理不同的域名的HTTP请求了.
```

监听端口

```
listen address:port[default | default_server |[backlog=num |revb
uf=size

| sndbuf=size| accept_filter=filter |deferred|bind|ipv6only=[on|
off] |ssl]]
```

默认listen 80

在listen之后可以只加IP地址,端口,或者主机名

```
isten 127.0.0.1:8000;
listen localhost;
listen 8000;
listen *:8000;
listen 443 default_server ssl;
```

```
default 将所在的server块作为整个WEB服务的默认server块;如果所有的server
都没设置这个参数, nginx.conf第一个server作为默认块.

当一个请求无法匹配配置文件的任一主机名就会选用默认虚拟主机. default_server
 同上

backlog=num 表示TCP中backlog队列的大小,默认为-1表示不设置. TCP三次握手
过程中进程还没有处理监听句柄,backlog用以放置这个新连接, 如果队列已满,新客户
端3次握手建立连接失败

deferred 设置这个参数后,通过三次握手之后内核不会在建立连接时处理,而是等发来
数据的时候处理连接.
bind 绑定当前地址:端口
ssl 在当前监听的端口上建立的连接必须给予SSL协议
```

主机名称

```
server_name name[...];
默认 server_name "";

server_name之后可以跟多个主机名称
eg: server_name www.wowpai.top download.wowpai.top;

在开始处理HTTP请求时,Nginx会取出header头中的host,与server中每个server_name进行比较, 如果多个server块都匹配需要按照优先级来选择处理的server块.

首先选择所有字符完全匹配的server_name www.wowpai.top
其次选择通配符在前面的server_name *.wowpai.top
其次选择通配符在后面的server_name www.wowpai.*
最后选择使用正则匹配的server_name ~^\.wowpai\.top$

如果以上都不能匹配将按照以下的server块:
优先选择在listen配置项后加入[default |default_server]的server块 找到匹配listen端口的第一个server块

如果server_name后面跟着字符串 server_name ""表示匹配没有Host这个HTTP头部的请求

Nginx使用server_name配置项针对特定Host域名的请求提供不同的服务以实现虚拟主机的功能.
```

server_names_hash_bucket_size

```
server_names_hash_bucket_size size;
默认 server_names_hash_bucket_size 32|64|128;
可配置块 http server location

为提高查找相应的server_name的能力,Nginx使用散列表来存储. size设置了每个散列块占用的字节数.
```

server_names_hash_max_size

```
server_names_hash_max_size size;
默认 server_names_hash_max_size 512;
可配置块 http serverlocation

server_names_hash_max_size 影响散列表的冲突率,server_names_hash_max
_size越大,冲突率越低,检索速度越快.
```

重定向主机名称的处理

```
server_name_in_redirect on|off;
默认 on

该配置需要配合server_name 使用.在使用on打开的时候,表示在重定向请求时会使用
server_name里面配置的
第一个主机名代替原来的请求中的Host头部.

当使用off关闭时,表示在重定向请求时使用请求本身的Host头部.
```

location

```
location [=|~|~* |^~| @ ] | /uri/ {}

可配置块 server
location会尝试根据用户请求中的URI来匹配上面的/uri表达式,如果可以[匹配就选择location{}块中的配置来处理用户请求.

location的匹配规则:

= 表示把URI作为字符 以便与参数中的uri做完全匹配
  location = /{
    #只有当用户请求/时 才会调用该location下的配置
  }

~ 表示匹配URI时字母大小写敏感的

~* 表示匹配URI时忽略字母大小写 后面可以跟上正则表达式
  location ~* \.(gif|jpg|jpeg){
    #匹配这三种图片的资源请求
  }

^~ 表示匹配URI只需要前半部分uri参数匹配即可
  location ^~ /images/{
      #以/images/开始的请求都会匹配上
  }

@ 表示仅用于Nginx服务内部请求之间的重定向,带有@的location不直接处理用户请求

没有匹配的URI应该得到一个响应,
location /{
  #前面所有的匹配都未成功就意味着会被这个location 匹配-----捕获
}

**location匹配的存在一定的优先级:先精确匹配然后模糊匹配,最后匹配/
```

文件路径定义

1 以**root**设置资源路径

```
root path;
默认 root html;

配置项: http server location if

location /download/ {
  root /opt/web/html/;
}

如果请求的URI是/download/index/test.html,那么WEB服务器应该返回的是服务器上
  /opt/web/html/download/index/test.html
```

2 以**alias**设置资源路径

```
alias path;
配置块:location

alias也是用来设置资源路径的.与root的不同点在于如何解读紧跟location后面的参数. 这将会致使alias与root以不同的方式将用户的请求映射到真正的磁盘文件上.

如果请求的URI是 /conf/nginx.conf,实际上访问文件在/usr/local/nginx/conf/nginx.conf

location /conf{
  alias /usr/local/nginx/conf/;
}

location /conf{
  root /usr/local/nginx/;
}

使用alias时在URI向实际文件映射的过程中,已经把location后配置的/conf这部分字符串丢弃掉. 最终映射成path/nginx.conf文件
而root则不一样 最终直接映射成 path/conf/nginx.conf
```

3 设置首页

```
index file ...;
默认 index index.html;
配置块 http server

location location /{
  root html;
  index index.html index.htm index.php;
}

优先返回index.php,没有的话返回index.htm,如果还没有,再尝试放回index.html
```

4 根据**HTTP**返回码重定向

```
rror_page code[code ...] [= | = answer-code] uri|@name_location

配置块 http server location if

如果某个请求返回错误码时匹配上了error_page中设置的code,则重定向到新的URI中. 虽然重定向了URI,但返回的错误码还是和原来的相同.可以通过'='来改变返回的错误码

error_page 404 =200 /empty.html

如果不修改 URI,只是想让这样的请求重定向到另一个location中处理

location /{
  error_page 404 @failback;
}

location @failback{
  proxy_pass http://127.0.0.1:8081;
}
```

5 是否允许定义**error_page**

```
recursive_error_pages [on|off];
默认recursive_error_pages off;
配置块 http server location
确定是否允许定义error_page
```

try_files

```
try_files path1 [path2] uri;

配置块 server location
try_files 后面可以跟上多个path,且最后一定要跟上uri

按照顺序遍历每个path,如果可以有效的读取就直接返回这个path并结束请求. 否则继续向后遍历,最后就重定向到uri上
location /{
  #try_files $uri $uri/ /$uri.html $uri/index.html @other;
  try_files $uri $uri/ /error/php?c=404 =404;
}

location @other{
  proxy_pass http://backend;
}
```

对客户端请求的限制

按HTTP方法名限制用户请求

```
limit_execpt method ...{
  ...
}
配置块 location
方法名有 PUT HEAD POST DELETE MKCOL COPY MOVE OPTIONS PROPFIND PROPPATCH LOCK UNLOCK PATCH
limit GET{
  allow 192.168.1.110/32;
  deny all;
}
禁止GET 和HEAD方法,其他方法允许
```

HTTP请求包体的最大值

```
client_max_body_size size;
默认 client_max_body_size 1m;
配置块 http server location

用户打算上传一个超过10G的文件,发超过定义client_max_size的值,回复错误
```

对请求的限速

```
imit_rate speed;
limit_rate 0;
配置块 http server location if 限制客户端请求限制每秒的传输的字节数.0表示不限制
    server{
        if ( $slow)
        {
            set $limit_rate 4k;
    }
}
```

5 Nginx 搭建反向代理

反向代理 (reverse proxy)方式是指用代理服务器来接受Internet上的连接请求,然后将请求转发给内部网络中的上游服务器,并将从上游服务器上得到的结果返回给Internet请 求连接的客户端,此时代理服务器对外的表现就是一个web服务器。

一般情况下,nginx在前端抗负载和处理静态页面请求,上游服务器可以挂接Apche/Tomcat等处理复杂业务的动态Web服务器。

服务器A:IP 172.16.0.35 作为一台代理服务器，大致配置如下。

```
http {
    include       mime.types;
    default_type  application/octet-stream;

    sendfile        on;
    keepalive_timeout  65;

    #配置均衡服务器
    upstream backup.com {
        server  172.16.0.138:80;#服务器B
        server  172.16.0.70:80; #服务器C
    }

    server {
        listen       80;
        server_name  localhost;


        location / {
            #配置反向代理功能
            proxy_pass http://backup.com;
            root   html;
            index  index.html index.htm;
        }

        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   html;
        }
 }
```

服务器B:IP 172.16.0.138

服务器C:IP 172.16.0.70

均作为普通的web服务器。

在浏览器输入 服务器A的ip地址，也就是反向代理的地址。 172.16.0.35

会发现，通过访问服务器A，最终处理客户端请求的确实服务器B和服务器C。

这种就是利用Nginx实现了反向代理。

6 Nginx 源码及常见数据结构

参考淘宝翻译的Nginx官方内容： http://tengine.taobao.org/book/index.html

或者去Nginx官方网站参考。

有关Nginx内存管理方式参考博文

http://www.cnblogs.com/v-July-v/archive/2011/12/04/2316410.html

书写的很详细。

7 自定义Nginx模块

7.1 ngx_command_t 数组

commands 数组用于定义模块的配置文件参数，每一个数组元素都是 `ngx_command_t` 类型，数组的结尾是用 `ngx_numm_command` 表示。Nginx在解析配置文件中的一个配置项时首先会遍历所有的模块，对于每一个模块而言，即通过遍历commands数组进行，另外，在数组中检查到ngx_numm_command时，会停止使用当前模块解析该配置项。每一个ngx_command_t结构体定义个如下配置:

```
typedef struct ngx_command_s ngx_command_t;

struct ngx_command_s {
    //配置项名称，如"gzip"
    ngx_str_t       name;
    //配置项类型，type将制定配置项可以出现的位置，
    //例如：出现在server{}活location{}中，等等
    ngx_uint_t      type;
    //出现了name中指定的配置项后，将会调用set方法处理配置项的参数
    char            *(*set)(ngx_conf_t *cf, ngx_command_t *cmd,
void *conf);
    //在配置文件中的偏移量
    ngx_uint_t      conf;
    //通常用于使用预设的解析方法解析配置项，这是配置模块的一个优秀的设计,
    //需要与conf配合使用
    ngx_uint_t      offset;
    //配置项读取后的处理方法，必须是ngx_conf_post_t结构的指针
    void            *post;
};
```

ngx_null_command只是一个空的ngx_command_s：

```
#define ngx_null_command    {ngx_null_string, 0, NULL, 0, 0, NUL
L}
```

也就是说，对于我们在nginx.cong 中编写的配置项mytest来说，nginx首先会遍历所有的模块（modules）,而对于每个模块，会遍历他所对应的ngx_command_t数组，试图找到关于我们配置项目mytest的解析方式。

```
static ngx_command_t  ngx_http_mytest_commands[] =
{

    {
        ngx_string("mytest"),
        NGX_HTTP_MAIN_CONF | NGX_HTTP_SRV_CONF | NGX_HTTP_LOC_CO
NF | NGX_HTTP_LMT_CONF | NGX_CONF_NOARGS,
        ngx_http_mytest,
        NGX_HTTP_LOC_CONF_OFFSET,
        0,
        NULL
    },

    ngx_null_command
};
```

7.2 command中用于处理配置项参数的set方法

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

    //首先找到mytest配置项所属的配置块，clcf貌似是location块内的数据
//结构，其实不然，它可以是main、srv或者loc级别配置项，也就是说在每个
//http{}和server{}内也都有一个ngx_http_core_loc_conf_t结构体
    clcf = ngx_http_conf_get_module_loc_conf(cf, ngx_http_core_m
odule);

    //http框架在处理用户请求进行到NGX_HTTP_CONTENT_PHASE阶段时，如果
//请求的主机域名、URI与mytest配置项所在的配置块相匹配，就将调用我们
//实现的ngx_http_mytest_handler方法处理这个请求
    clcf->handler = ngx_http_mytest_handler;

    return NGX_CONF_OK;
}
```

关于ngx_http_conf_get_module_loc_conf 的定义可以参考：

http://lxr.nginx.org/source/src/http/ngx_http_config.h#0065

> 本质：就是设置 `ngx_http_mytest_handler`，匹配项被选中的时候，应该如何解析。

7.3 定义ngx_http_module_t接口

这部分的代码，是用于http框架的，相当于http框架的回掉函数，由于这里并不需要框架做任何操作。

```
static ngx_http_module_t  ngx_http_mytest_module_ctx =
{
    NULL,                                  /* preconfiguration */
    NULL,                              /* postconfiguration */

    NULL,                                  /* create main configurat
ion */
    NULL,                                  /* init main configuratio
n */

    NULL,                                  /* create server configur
ation */
    NULL,                                  /* merge server configura
tion */

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

7.4 定义处理“mytest”command的handler

我们这里需要定义配置项被匹配之后的处理方法.

```
static ngx_int_t ngx_http_mytest_handler(ngx_http_request_t *r)
{
    //必须是GET或者HEAD方法，否则返回405 Not Allowed
    if (!(r->method & (NGX_HTTP_GET | NGX_HTTP_HEAD)))
    {
        return NGX_HTTP_NOT_ALLOWED;
    }

    //丢弃请求中的包体
    ngx_int_t rc = ngx_http_discard_request_body(r);
    if (rc != NGX_OK)
    {
        return rc;
    }
```

```
    //设置返回的Content-Type。注意，ngx_str_t有一个很方便的初始化宏
//ngx_string，它可以把ngx_str_t的data和len成员都设置好
    ngx_str_t type = ngx_string("text/plain");
    //返回的包体内容
    ngx_str_t response = ngx_string("Hello World!");
    //设置返回状态码
    r->headers_out.status = NGX_HTTP_OK;
    //响应包是有包体内容的，所以需要设置Content-Length长度
    r->headers_out.content_length_n = response.len;
    //设置Content-Type
    r->headers_out.content_type = type;

    //发送http头部
    rc = ngx_http_send_header(r);
    if (rc == NGX_ERROR || rc > NGX_OK || r->header_only)
    {
        return rc;
    }

    //构造ngx_buf_t结构准备发送包体
    ngx_buf_t                  *b;
    b = ngx_create_temp_buf(r->pool, response.len);
    if (b == NULL)
    {
        return NGX_HTTP_INTERNAL_SERVER_ERROR;
    }
    //将Hello World拷贝到ngx_buf_t指向的内存中
    ngx_memcpy(b->pos, response.data, response.len);
    //注意，一定要设置好last指针
    b->last = b->pos + response.len;
    //声明这是最后一块缓冲区
    b->last_buf = 1;

    //构造发送时的ngx_chain_t结构体
    ngx_chain_t     out;
    //赋值ngx_buf_t
    out.buf = b;
    //设置next为NULL
    out.next = NULL;
```

```
    //最后一步发送包体，http框架会调ngx_http_finalize_request方法
    //结束请求
    return ngx_http_output_filter(r, &out);
}
```

7.5 定义ngx_module_t中的mytest模块

定义HTTP模块方式很简单，例如：

```
ngx_module_t ngx_http_mytest_module;
```

其中ngx_module_t是一个Nginx模块的数据结构，下面来分析一下Nginx模块中的所有成员：

```
typedef struct ngx_module_s ngx_module_t;
struct ngx_module_s {
  /*
  下面的ctx_index、index、spare0,spare1,
  spare2,spare3,version变量不需要再定义时候赋值，
  可以用Nginx准备好的宏NGX_MODULE_V1来定义，
  已经定义好了这7个值

    #define NGX_MODULE_V1  0,0,0,0,0,0,1

    对于一类模块（由下面的type成员决定类别）而言，
    ctx_index表示当前模块在这类模块中的序号。
    这个成员常常是由管理这类模块的一个Nginx核心模块设置的，
    对于所有的HTTP模块而言，ctx_index是由
    核心模块ngx_http_module设置的。
    ctx_index非常重要，Nginx的模块化设计非常依赖于各个模块的顺序，
    他们既用于表达优先级，也用于表明每个模块的位置，
    借以帮助Nginx框架快速获得某个模块的数据.
  */
  ngx_uint_t            ctx_index;


  /*
  index表示当前模块在ngx_module数组中的序号。
```

```
    注意，ctx_index表示的是当前模块在一类模块中的序号，
    而index表示当前模块在所有模块中的序号，同样很关键.
    Nginx启动时会根据ngx_modules数组设置各模块的index值，
    例如：
      ngx_max_module = 0;
      for(i = 0; ngx_module[i]; i++) {
          ngx_modules[i]->index = ngx_max_module++;
      }
    */
    ngx_uint_t            index;


    //spare系列的保留变量，暂未使用
    ngx_uint_t            spare0;
    ngx_uint_t            spare1;
    ngx_uint_t            spare2;
    ngx_uint_t            spare3;

    //模块的版本，便于将来的拓展，目前只有一种，默认为1
    ngx_uint_t            version;


    /*
    ctx用于指向一类模块的上下文结构体，
    为什么需要ctx呢？因为前面说过，
    Nginx模块有许多种类，不同类模块之间的功能差别很大。
    例如，
    事件类型的模块主要处理I/O事件的相关功能，
    HTTP类型的模块主要处理HTTP应用层的功能。
    这样每个模块都有了自己的特性，
    而ctx会指向特定类型模块的公共接口。
    例如，
    在HTTP模块中，ctx需要指向ngx_http_module_t结构体
    */
    void                  *ctx;

    /*
    commands处理nginx.conf中的配置项
    */
    ngx_command_t         *commands;
```

```
    /*
    type表示该模块的类型，它与ctx指针是紧密相关的。
    在官方Nginx中，它的取值范围有以下5种：
    NGX_HTTP_MODULE
    NGX_CORE_MODULE
    NGX_CONF_MODULE
    NGX_EVENT_MODULE
    NGX_MAIL_MODULE
    */
    ngx_uint_t             type;

    /*
    在Nginx的启动、停止过程中，以下7个函数指针表示7个
    执行点分别用调用这7种方法。对于任意一个方法而言，
    如果不需要Nginx再某个时刻执行它，那么简单地把它设为NULL
    空指针即可。
    */

    /*
    虽然从字面上理解应当在master进程启动的时，回调
    init_master,但到目前为止，框架代码从来不会调动它，
    所以设置为NULL
    */
    ngx_int_t   (*init_master)(ngx_log_t *log);

    /*
    init_module回调方法在初始化所有模块时候被调用。
    在master/worker模式下，这个阶段将在启动worker子进程前完成。
    */
    ngx_int_t   (*init_module)(ngx_cycle_t *cycle);

    /*
    init_process 回调方法在正常服务前被调用。
    在master/worker模式下，多个worker子进程已经产生。
    在每个worker进程的初始化过程会调用所有模块的init_process函数
    */
    ngx_int_t   (*init_process)(ngx_cycle_t *cycle);

    /*
```

```
    由于Nginx暂时不支持多线程模式，所以init_thread在框架中
    没有被调用过，设置为NULL
    */
    ngx_int_t   (*init_thread)(ngx_cycle_t *cycle);

    /*
    同上、exit_thread也不支持，设为NULL
    */
    void        (*exit_thread)(ngx_cycle_t *cycle);

    /*
    exit_process回调方法在服务停止前被调用。
    在/master/worker模式下，worker进程会在退出前调用
    */
    void        (*exit_process)(ngx_cycle_t *cycle);


    /*
    exit_master回调方法将在master进程退出前被调用
    */
    void        (*exit_master)(ngx_cycle_t *cycle);


    /*
    一下8个spare_hook变量也是保留字段，目前没有使用，
    但可用Nginx提供的NGX_MODULE_V1_PADDING宏来填充
    #define NGX_MODULE_V1_PADDING 0,0,0,0,0,0,0,0
    */
    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;
};
```

所以在定义一个HTTP模块的时候，务必把type字段设置为NGX_HTTP_MODULE.

对于下列回调方法：init_module、init_process、exit_process、exit_master 调用他们的是Nginx框架代码。换句话说，这4个回调方法与HTTP框架无关。 即使nginx.conf中没有设置http{...}这种开启HTTP功能的配置项，这些 回调方法仍然会被调用。因此，通常开发HTTP模块时候都把他们设置为NULL。 这样Nginx不作为web服务器使用时，不会执行HTTP模块的任何代码。

定义HTTP时候，最重要的是要设置ctx和commands这两个成员。 对于HTTP类型模块来说，ngx*module*中的ctx指针必须指向ngx_http_module_t接口。

所以最终我们定义个ngx_module_t 模块如下：

```
ngx_module_t  ngx_http_mytest_module =
{
    NGX_MODULE_V1,
    &ngx_http_mytest_module_ctx,           /* module context */
    ngx_http_mytest_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
};
```

这样， mytest 模块在编译的时候， 就可以被加入到ngx_modules的全局数组中了。

7.6 完整代码 ngx_http_mytest_module.c

所以全部的编码工作完毕，最终的代码应该如下：

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

```
#include <sys/types.h>
#include <unistd.h>


//定义处理用户请求hello world handler
static ngx_int_t ngx_http_mytest_handler(ngx_http_request_t *r)
{
    //必须是GET或者HEAD方法，否则返回405 Not Allowed
    if (!(r->method & (NGX_HTTP_GET | NGX_HTTP_HEAD)))
    {
        return NGX_HTTP_NOT_ALLOWED;
    }

    //丢弃请求中的包体
    ngx_int_t rc = ngx_http_discard_request_body(r);
    if (rc != NGX_OK)
    {
        return rc;
    }

    //设置返回的Content-Type。注意，ngx_str_t有一个很方便的初始化宏
    //ngx_string，它可以把ngx_str_t的data和len成员都设置好
    ngx_str_t type = ngx_string("text/plain");
    //返回的包体内容
    ngx_str_t response = ngx_string("Hello World!");
    //设置返回状态码
    r->headers_out.status = NGX_HTTP_OK;
    //响应包是有包体内容的，所以需要设置Content-Length长度
    r->headers_out.content_length_n = response.len;
    //设置Content-Type
    r->headers_out.content_type = type;

    //发送http头部
    rc = ngx_http_send_header(r);
    if (rc == NGX_ERROR || rc > NGX_OK || r->header_only)
    {
        return rc;
    }

    //构造ngx_buf_t结构准备发送包体
```

```
    ngx_buf_t                   *b;
    b = ngx_create_temp_buf(r->pool, response.len);
    if (b == NULL)
    {
        return NGX_HTTP_INTERNAL_SERVER_ERROR;
    }
    //将Hello World拷贝到ngx_buf_t指向的内存中
    ngx_memcpy(b->pos, response.data, response.len);
    //注意，一定要设置好last指针
    b->last = b->pos + response.len;
    //声明这是最后一块缓冲区
    b->last_buf = 1;

    //构造发送时的ngx_chain_t结构体
    ngx_chain_t     out;
    //赋值ngx_buf_t
    out.buf = b;
    //设置next为NULL
    out.next = NULL;

    //最后一步发送包体，http框架会调用ngx_http_finalize_request方法
    //结束请求
    return ngx_http_output_filter(r, &out);
}


//定义command用于处理配置项参数的set方法
    static char *
ngx_http_mytest(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
{
    ngx_http_core_loc_conf_t  *clcf;

    //首先找到mytest配置项所属的配置块，clcf貌似是location块内的数据
    //结构，其实不然，它可以是main、srv或者loc级别配置项，也就是说在每个
    //http{}和server{}内也都有一个ngx_http_core_loc_conf_t结构体
    clcf = ngx_http_conf_get_module_loc_conf(cf, ngx_http_core_m
odule);

    //http框架在处理用户请求进行到NGX_HTTP_CONTENT_PHASE阶段时，如果
    //请求的主机域名、URI与mytest配置项所在的配置块相匹配，就将调用我们
```

```
    //实现的ngx_http_mytest_handler方法处理这个请求
    clcf->handler = ngx_http_mytest_handler;

    return NGX_CONF_OK;
}

//定义ngx mytest 配置匹配的command
static ngx_command_t  ngx_http_mytest_commands[] =
{

    {
        ngx_string("mytest"),
        NGX_HTTP_MAIN_CONF | NGX_HTTP_SRV_CONF | NGX_HTTP_LOC_CO
NF | NGX_HTTP_LMT_CONF | NGX_CONF_NOARGS,
        ngx_http_mytest,
        NGX_HTTP_LOC_CONF_OFFSET,
        0,
        NULL
    },

    ngx_null_command
};


//定义ngx_http_module_t接口
static ngx_http_module_t  ngx_http_mytest_module_ctx =
{
    NULL,                          /* preconfiguration */
    NULL,                          /* 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 */
};
```

```
//定义mytest模块
ngx_module_t  ngx_http_mytest_module =
{
    NGX_MODULE_V1,
    &ngx_http_mytest_module_ctx,           /* module context */
    ngx_http_mytest_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
};
```

7.7 配置文件 config

但是之后我们还需要给该模块相同目录下提供一个配置文件"config"表示在nginx编译的时候的一些编译选项。

```
ngx_addon_name=ngx_http_mytest_module
HTTP_MODULES="$HTTP_MODULES ngx_http_mytest_module"
NGX_ADDON_SRCS="$NGX_ADDON_SRCS $ngx_addon_dir/ngx_http_mytest_m
odule.c"
```

所以我们的模块编写完毕了，当前目录应该有两个文件

```
ls

config  ngx_http_mytest_module.c
```

7.8 重新编译Nginx 并添加自定义模块

进入Nginx源码目录 执行

```
./configure --prefix=/usr/local/nginx --add-module=/home/ace/ope
nSource_test/nginx_module_http_test
```

--add-module为刚才自定义模块源码的目录

```
make
sudo make install
```

测试自定义模块

修改nginx.conf文件

```
    server {
         listen 8777;

         server_name localhost;

         location / {
             mytest;#我们自定义模块的名称
         }
     }
```

重新启动nginx

打开浏览器输入地址和端口