### 走一遍

systemctl stop firewalld.service #停止firewall

systemctl disable firewalld.service #禁止firewall开机启动

firewall-cmd --state #查看默认防火墙状态（关闭后显示notrunning，开启后显示running）

# 进程与线程的一个简单解释

<http://www.ruanyifeng.com/blog/2013/04/processes_and_threads.html>

线程就好比车间里的工人。一个进程可以包括多个线程。

# Nginx简介

<https://aceld.gitbooks.io/nginx-zh/content/33_nginxde_mo_kuai_hua_ti_xi.html>

**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

1: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;

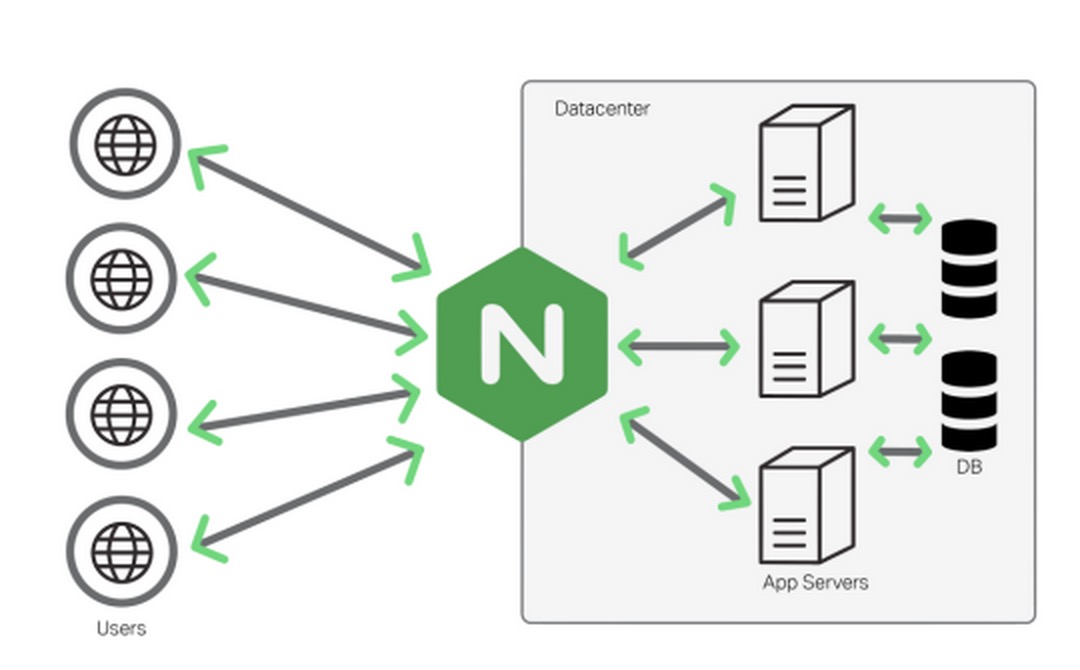
2:反向代理服务器

何为反向代理服务器

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。

3:邮件服务器

Nginx也可用充当一个IMAP/POP3/SMTP服务器。

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。

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/>

# 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。

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。

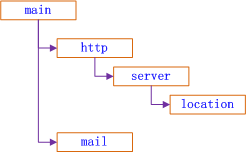
3:如果worker进程的异常退出，肯定是程序有bug了，异常退出，会导致当前worker上的所有请求失败，不过不会影响到所有请求，所以降低了风险。

当然，好处还有很多，大家可以慢慢体会。

# Nginx配置系统

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。

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：

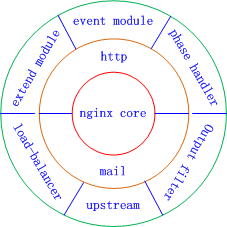
* index
* root

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.

# Nginx模块化体系

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。

# Nginx相关配置

nginx配置通用语法

一个master进程来管理多个work进程

work进程数量和cpu的核心数相同(进程间切换的代价最小)

# 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'

# 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Nginx

**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server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 |revbuf=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;

}

}

# 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。

# 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, NULL}

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\_CONF | 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\_module);

//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 configuration \*/

NULL, /\* init main configuration \*/

NULL, /\* create server configuration \*/

NULL, /\* merge server configuration \*/

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，ngxmodule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\_module);

//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\_CONF | 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\_module.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/openSource\_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 