Nginx 教程从入门到精通

WWW.TTLSA.COM 网站作品
作者：凉白开，漠北
由 DONAN 整理
QQ：305765814　QQ 群:6690706

Nginx 教程从入门到精通

目　录

Nginx 基础

(1)nginx 安装

当今 nginx 的劲头越来越猛，记得 2011 年版本才 1.0.6,现在已经更新到了 1.5.1,nginx 的更新速度越来越快。一直想记录一系列的 nginx 教程，处于各种原因没去做.今年抽出时间完成平时工作上用到的 nginx.后续将会以视频教程的方式来做。当然，还是文章先出，下一篇文章将会讲 nginx 虚拟主机配置。有什么建议，望大家留言。

1、必要软件准备

- 安装 pcre
  为了支持 rewrite 功能，我们需要安装 pcre
  # yum install pcre* //如过你已经装了，请跳过这一步
- 安装 openssl
  需要 ssl 的支持，如果不需要 ssl 支持，请跳过这一步

```
# yum install openssl*
```

2、安装 nginx

执行如下命令：

```
# ./configure --prefix=/usr/local/nginx-1.5.1 \
--with-http_ssl_module --with-http_spdy_module \
--with-http_stub_status_module --with-pcre
```

-with-http_stub_status_module：支持 nginx 状态查询
-with-http_ssl_module：支持 https
-with-http_spdy_module：支持 google 的 spdy,想了解请百度 spdy,这个必须有 ssl 的支持
-with-pcre：为了支持 rewrite 重写功能，必须制定 pcre

最后输出如下内容，表示 configure OK 了。
…..

```
checking for zlib library ... found
  creating objs/Makefile
Configuration summary
  + using system PCRE library
  + using system OpenSSL library
  + md5: using OpenSSL library
  + sha1: using OpenSSL library
  + using system zlib library
nginx path prefix: "/usr/local/nginx-1.5.1"
  nginx binary file: "/usr/local/nginx-1.5.1/sbin/nginx"
  nginx configuration prefix: "/usr/local/nginx-1.5.1/conf"
  nginx configuration file: "/usr/local/nginx-1.5.1/conf/nginx.conf"
  nginx pid file: "/usr/local/nginx-1.5.1/logs/nginx.pid"
  nginx error log file: "/usr/local/nginx-1.5.1/logs/error.log"
  nginx http access log file: "/usr/local/nginx-1.5.1/logs/access.log"
```

```
    nginx http client request body temporary files:  "client_body_temp"
    nginx http proxy temporary files:  "proxy_temp"
    nginx http fastcgi temporary files:  "fastcgi_temp"
    nginx http uwsgi temporary files:  "uwsgi_temp"
    nginx http scgi temporary files:  "scgi_temp"
  # make //确定你的服务器有安装 make，如果没有安装请执行 yum install make
    # make install
```

3、启动、关闭、重置 nginx

启动：直接执行以下命令,nginx 就启动了,不需要改任何配置文件,nginx 配置多域名虚拟主机请参考后续文章.

```
  /usr/local/nginx-1.5.1/sbin/nginx
```

试试访问：我这边不贴图，直接使用 curl 命令来读取 web 信息

```
  [root@ns conf]# curl -s http://localhost | grep nginx.com
  nginx.com.
```

关闭：

```
  /usr/local/nginx-1.5.1/sbin/nginx -s stop
```

重置：当你有修改配置文件的时候，只需要 reload 以下即可

```
  /usr/local/nginx-1.5.1/sbin/nginx -s reload
```

整个 nginx 的安装就到这里结束了。

(2)nginx 编译参数详解

标题是不是很欠揍,个人认为确实值得一看,如果你不了解 nginx，或者你刚学 nginx，或者已经使用 nginx 一段时间了。但是 nginx 很多参数你还没去了解，nginx 有很多你不知道的用处。不废话，上内容。内容是从网络上摘抄的。

内容有些多，一眼看来难免头昏脑胀，但坚持看完，相信你一定会有所收获。

nginx 参数：

-prefix= 指向安装目录

-sbin-path 指向（执行）程序文件（nginx）

-conf-path= 指向配置文件（nginx.conf）

-error-log-path= 指向错误日志目录

-pid-path= 指向 pid 文件（nginx.pid）

-lock-path= 指向 lock 文件（nginx.lock）（安装文件锁定，防止安装文件被别人利用，或自己误操作。）

-user= 指定程序运行时的非特权用户

-group= 指定程序运行时的非特权用户组

-builddir= 指向编译目录

-with-rtsig_module 启用 rtsig 模块支持（实时信号）

-with-select_module 启用 select 模块支持（一种轮询模式,不推荐在高载环境下使用）禁用：-without-select_module

-with-poll_module 启用 poll 模块支持（功能与 select 相同，与 select 特性相同，为一种轮询模式,不推荐在高载环境下使用）

-with-file-aio 启用 file aio 支持（一种 APL 文件传输格式）

-with-ipv6 启用 ipv6 支持

-with-http_ssl_module 启用 ngx_http_ssl_module 支持（使支持 https 请求，需已安装 openssl）

-with-http_realip_module 启用 ngx_http_realip_module 支持（这个模块允许从请求标头更改客户端的 IP 地址值，默认为关）

-with-http_addition_module 启用 ngx_http_addition_module 支持（作为一个输出过滤器，支持不完全缓冲，分部分响应请求）
-with-http_xslt_module 启用 ngx_http_xslt_module 支持（过滤转换 XML 请求）
-with-http_image_filter_module 启用 ngx_http_image_filter_module 支持（传输 JPEG/GIF/PNG 图片的一个过滤器）（默认为不启用。gd 库要用到）
-with-http_geoip_module 启用 ngx_http_geoip_module 支持（该模块创建基于与 MaxMind GeoIP 二进制文件相配的客户端 IP 地址的 ngx_http_geoip_module 变量）
-with-http_sub_module 启用 ngx_http_sub_module 支持（允许用一些其他文本替换 nginx 响应中的一些文本）
-with-http_dav_module 启用 ngx_http_dav_module 支持（增加 PUT,DELETE,MKCOL：创建集合,COPY 和 MOVE 方法）默认情况下为关闭，需编译开启
-with-http_flv_module 启用 ngx_http_flv_module 支持（提供寻求内存使用基于时间的偏移量文件）
-with-http_gzip_static_module 启用 ngx_http_gzip_static_module 支持（在线实时压缩输出数据流）
-with-http_random_index_module 启用 ngx_http_random_index_module 支持（从目录中随机挑选一个目录索引）
-with-http_secure_link_module 启用 ngx_http_secure_link_module 支持（计算和检查要求所需的安全链接网址）
-with-http_degradation_module　启用 ngx_http_degradation_module 支持（允许在内存不足的情况下返回 204 或 444 码）
-with-http_stub_status_module 启用 ngx_http_stub_status_module 支持（获取 nginx 自上次启动以来的工作状态）
-without-http_charset_module 禁用 ngx_http_charset_module 支持（重新编码 web 页面，但只能是一个方向-服务器端到客户端，并且只有一个字节的编码可以被重新编码）
-without-http_gzip_module 禁用 ngx_http_gzip_module 支持（该模块同-with-http_gzip_static_module 功能一样）
-without-http_ssi_module 禁用 ngx_http_ssi_module 支持（该模块提供了一个在输入端处理处理服务器包含文件（SSI）的过滤器，目前支持 SSI 命令的列表是不完整的）
-without-http_userid_module 禁用 ngx_http_userid_module 支持（该模块用来处理用来确定客户端后续请求的 cookies）
-without-http_access_module 禁用 ngx_http_access_module 支持（该模块提供了一个简单的基于主机的访问控制。允许/拒绝基于 ip 地址）
-without-http_auth_basic_module 禁用 ngx_http_auth_basic_module（该模块是可以使用用户名和密码基于 http 基本认证方法来保护你的站点或其部分内容）
-without-http_autoindex_module 禁用 disable ngx_http_autoindex_module 支持（该模块用于自动生成目录列表，只在 ngx_http_index_module 模块未找到索引文件时发出请求。）
-without-http_geo_module 禁用 ngx_http_geo_module 支持（创建一些变量，其值依赖于客户端的 IP 地址）
-without-http_map_module 禁用 ngx_http_map_module 支持（使用任意的键/值对设置配置变量）
-without-http_split_clients_module 禁用 ngx_http_split_clients_module 支持（该模块用来基于某些条件划分用户。条件如：ip 地址、报头、cookies 等等）
-without-http_referer_module 禁用 disable ngx_http_referer_module 支持（该模块用来过滤请求，拒绝报头中 Referer 值不正确的请求）
-without-http_rewrite_module 禁用 ngx_http_rewrite_module 支持（该模块允许使用正则表达式改变 URI，并且根据变量来转向以及选择配置。如果在 server 级别设置该选项，那么他们将在 location 之前生效。如果在 location 还有更进一步的重写规则，location 部分的规则依然会被执行。如果这个 URI 重写是因为 location 部分的规则造成的，那么 location 部分会再次被执行作为新的 URI。 这个循环会执行 10 次，然后 Nginx 会返回一个 500 错误。）
-without-http_proxy_module 禁用 ngx_http_proxy_module 支持（有关代理服务器）
-without-http_fastcgi_module 禁用 ngx_http_fastcgi_module 支持（该模块允许 Nginx 与 FastCGI 进程交互，并通过传递参数来控制 FastCGI 进程工作。 ）FastCGI 一个常驻型的公共网关接口。

--without-http_uwsgi_module 禁用 ngx_http_uwsgi_module 支持（该模块用来医用 uwsgi 协议，uWSGI 服务器相关）
--without-http_scgi_module 禁用 ngx_http_scgi_module 支持（该模块用来启用 SCGI 协议支持，SCGI 协议是 CGI 协议的替代。它是一种应用程序与 HTTP 服务接口标准。它有些像 FastCGI 但他的设计 更容易实现。）
--without-http_memcached_module 禁用 ngx_http_memcached_module 支持（该模块用来提供简单的缓存，以提高系统效率）
-without-http_limit_zone_module 禁用 ngx_http_limit_zone_module 支持（该模块可以针对条件，进行会话的并发连接数控制）
--without-http_limit_req_module 禁用 ngx_http_limit_req_module 支持（该模块允许你对于一个地址进行请求数量的限制用一个给定的 session 或一个特定的事件）
--without-http_empty_gif_module 禁用 ngx_http_empty_gif_module 支持（该模块在内存中常驻了一个 1*1 的透明 GIF 图像，可以被非常快速的调用）
--without-http_browser_module 禁用 ngx_http_browser_module 支持（该模块用来创建依赖于请求报头的值。如果浏览器为 modern ，则$modern_browser 等于 modern_browser_value 指令分配的值；如 果浏览器为 old，则 $ancient_browser 等于 ancient_browser_value 指令分配的值；如果浏览器为 MSIE 中的任意版本，则 $msie 等于 1）
--without-http_upstream_ip_hash_module 禁用 ngx_http_upstream_ip_hash_module 支持（该模块用于简单的负载均衡）
--with-http_perl_module 启用 ngx_http_perl_module 支持（该模块使 nginx 可以直接使用 perl 或通过 ssi 调用 perl）
--with-perl_modules_path= 设定 perl 模块路径
--with-perl= 设定 perl 库文件路径
--http-log-path= 设定 access log 路径
--http-client-body-temp-path= 设定 http 客户端请求临时文件路径
--http-proxy-temp-path= 设定 http 代理临时文件路径
--http-fastcgi-temp-path= 设定 http fastcgi 临时文件路径
--http-uwsgi-temp-path= 设定 http uwsgi 临时文件路径
--http-scgi-temp-path= 设定 http scgi 临时文件路径
-without-http 禁用 http server 功能
--without-http-cache 禁用 http cache 功能
--with-mail 启用 POP3/IMAP4/SMTP 代理模块支持
--with-mail_ssl_module 启用 ngx_mail_ssl_module 支持
--without-mail_pop3_module 禁用 pop3 协议（POP3 即邮局协议的第 3 个版本,它是规定个人计算机如何连接到互联网上的邮件服务器进行收发邮件的协议。是因特网电子邮件的第一个离线协议标 准,POP3 协议允许用户从服务器上把邮件存储到本地主机上,同时根据客户端的操作删除或保存在邮件服务器上的邮件。POP3 协议是 TCP/IP 协议族中的一员，主要用于 支持使用客户端远程管理在服务器上的电子邮件）
--without-mail_imap_module 禁用 imap 协议（一种邮件获取协议。它的主要作用是邮件客户端可以通过这种协议从邮件服务器上获取邮件的信息，下载邮件等。IMAP 协议运行在 TCP/IP 协议之上， 使用的端口是 143。它与 POP3 协议的主要区别是用户可以不用把所有的邮件全部下载，可以通过客户端直接对服务器上的邮件进行操作。）
--without-mail_smtp_module 禁用 smtp 协议（SMTP 即简单邮件传输协议,它是一组用于由源地址到目的地址传送邮件的规则，由它来控制信件的中转方式。SMTP 协议属于 TCP/IP 协议族，它帮助每台计算机在发送或中转信件时找到下一个目的地。）
--with-google_perftools_module 启用 ngx_google_perftools_module 支持（调试用，剖析程序性能瓶颈）
--with-cpp_test_module 启用 ngx_cpp_test_module 支持
--add-module= 启用外部模块支持
--with-cc= 指向 C 编译器路径
--with-cpp= 指向 C 预处理路径

-with-cc-opt= 设置 C 编译器参数（PCRE 库，需要指定 -with-cc-opt="-I /usr/local/include"，如果使用 select()函数则需要同时增加文件描述符数量，可以通过 -with-cc- opt="-D FD_SETSIZE=2048"指定。）
-with-ld-opt= 设置连接文件参数。（PCRE 库，需要指定 -with-ld-opt="-L /usr/local/lib"。）
-with-cpu-opt= 指定编译的 CPU，可用的值为: pentium, pentiumpro, pentium3, pentium4, athlon, opteron, amd64, sparc32, sparc64, ppc64
-without-pcre 禁用 pcre 库
-with-pcre 启用 pcre 库
-with-pcre= 指向 pcre 库文件目录
-with-pcre-opt= 在编译时为 pcre 库设置附加参数
-with-md5= 指向 md5 库文件目录（消息摘要算法第五版，用以提供消息的完整性保护）
-with-md5-opt= 在编译时为 md5 库设置附加参数
-with-md5-asm 使用 md5 汇编源
-with-sha1= 指向 sha1 库目录（数字签名算法，主要用于数字签名）
-with-sha1-opt= 在编译时为 sha1 库设置附加参数
-with-sha1-asm 使用 sha1 汇编源
-with-zlib= 指向 zlib 库目录
-with-zlib-opt= 在编译时为 zlib 设置附加参数
-with-zlib-asm= 为指定的 CPU 使用 zlib 汇编源进行优化，CPU 类型为 pentium, pentiumpro
-with-libatomic 为原子内存的更新操作的实现提供一个架构
-with-libatomic= 指向 libatomic_ops 安装目录
-with-openssl= 指向 openssl 安装目录
-with-openssl-opt 在编译时为 openssl 设置附加参数
-with-debug 启用 debug 日志

(3)nginx 安装配置+清缓存模块安装

经过一段时间的使用，发现 nginx 在并发与负载能力方面确实优于 apache，现在已经将大部分站点从 apache 转到了 nginx 了。以下是 nginx 的一些简单的安装配置。

环境

操作系统：CentOS、RedHat

IP 地址：192.168.1.202

下载软件包

```
# mkdir /usr/local/src/tarbag
# mkdir /usr/local/src/software
# cd /usr/local/src/tarbag/
Nginx
# wget  http://www.nginx.org/download/nginx-1.0.6.tar.gz
Nginx cache purge 模块(可选)
# wget  http://labs.frickle.com/files/ngx_cache_purge-1.3.tar.gz
```

编译安装

```
# cd /usr/local/src/tarbag/
# tar -xzvf nginx-1.0.6.tar.gz -C /usr/local/src/software
# tar -xzvf ngx_cache_purge-1.3.tar.gz -C /usr/local/src/software
# cd /usr/local/src/software/
# ./configure \
 -prefix=/usr/local/nginx-1.0.6 \    #  安装路径
```

```
-with-http_stub_status_module \ #  启用 nginx 状态模块
-with-http_ssl_module \ #  启用 SSL 模块
-with-http_realip_module \ #  启用 realip 模块（将用户 IP 转发给后端服务器）
-add-module=../ngx_cache_purge-1.3 #  添加缓存清除扩展模块
# make
# make install
```

内核参数优化

```
# vi sysctl.conf     增加以下配置
net.ipv4.netfilter.ip_conntrack_tcp_timeout_established = 1800
net.ipv4.ip_conntrack_max = 16777216   #   如果使用默认参数,容易出现网络丢包
net.ipv4.netfilter.ip_conntrack_max = 16777216#   如果使用默认参数,容易出现网络丢包
net.ipv4.tcp_max_syn_backlog = 65536
net.core.netdev_max_backlog =     32768
net.core.somaxconn = 32768
net.core.wmem_default = 8388608
net.core.rmem_default = 8388608
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216
net.ipv4.tcp_timestamps = 0
net.ipv4.tcp_synack_retries = 2
net.ipv4.tcp_syn_retries =
net.ipv4.tcp_tw_recycle = 1
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_mem = 94500000 915000000 927000000
net.ipv4.tcp_max_orphans = 3276800
net.ipv4.ip_local_port_range = 1024     65535
配置生效
  # sysctl  -p
修改 iptables 启动脚本,在 star()函数里面加上
  # vi /etc/init.d/iptables
  /sbin/sysctl    -p
```

配置范例站点站点

```
序号
域名
目录
1
www.heytool.com
/www/html/www.heytool.com
2
bbs.heytool.com
/www/html/bbs.heytool.com
```

修改 nginx 配置文件:

```
  # vi nginx.conf
  user     nobody nobody; #  运行 nginx 的所属组和所有者
```

```
worker_processes    2; #  开启两个 nginx 工作进程,一般几个 CPU 核心就写几
error_log    logs/error.log    notice; #  错误日志路径
pid          logs/nginx.pid; # pid 路径
events {
  worker_connections    1024; #  一个进程能同时处理 1024 个请求
}
http {
  include             mime.types;
  default_type    application/octet-stream;

  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; #  默认访问日志路径
  sendfile                on;
  keepalive_timeout    65; # keepalive 超市时间
#  开始配置一个域名,一个 server 配置段一般对应一个域名
server {
  listen              80; #
  #  在本机所有 ip 上监听 80,也可以写为 192.168.1.202:80,这样的话,就只监听 192.168.1.202 上的 80 口
  server_name    www.heytool.com; #  域名
  root      /www/html/www.heytool.com; #  站点根目录（程序目录）
  index    index.html index.htm; #  索引文件
  location / {   #  可以有多个 location
      root      /www/html/www.heytool.com; #  站点根目录（程序目录）
  }
  error_page      500 502 503 504    /50x.html;
  #  定义错误页面,如果是 500 错误,则把站点根目录下的 50x.html 返回给用户
  location = /50x.html {
    root      /www/html/www.heytool.com;
  }
}
#  开始配置站点 bbs.heytool.com
server {
  listen              80;
  server_name    bbs.heytool.com;
  root      /www/html/bbs.heytool.com;
  index    index.html index.htm; #  索引文件
  location / {
      root      /www/html/bbs.heytool.com;
  }
  error_page      500 502 503 504    /50x.html;
  location = /50x.html {
      root      /www/html/bbs.heytool.com;
  }
}
}
```

Nginx 启动关闭

```
# /usr/local/nginx-1.0.6/sbin/nginx    //启动 nginx
```

```
# /usr/local/nginx-1.0.6/sbin/nginx  -t //测试 nginx 配置文件的准确性
# /usr/local/nginx-1.0.6/sbin/nginx  -s reload //重载 nginx
# /usr/local/nginx-1.0.6/sbin/nginx  -s stop //关闭 nginx
```

测试

创建测试站点

```
# mkdir  -p /www/html/www.heytool.com
# mkdir  -p /www/html/bbs.heytool.com
# echo  “www.heytool.com”  > /www/html/www.heytool.com/index.html
#  echo  “bbs.heytool.com”  > /www/html/bbs.heytool.com/index.html
```

启动 nginx

```
# /usr/local/nginx-1.0.6/sbin/nginx  -t //看到 ok 和 successful,说明配置文件没问题
nginx: the configuration file /usr/local/ nginx-1.0.6/conf/nginx.conf syntax is ok
nginx: configuration file /usr/local/ nginx-1.0.6/conf/nginx.conf test is successful
# /usr/local/nginx-1.0.6/sbin/nginx
```

绑定 hosts,测试

把两个域名指向 192.168.1.202

192.168.1.202　　www.heytool.com

192.168.1.202　　bbs.heytool.com

打开 www.heytool.com,如下图：

nginx

打开 bbs.heytool.com,如下图：

nginx

完毕！！！！

(4)nginx 连接 PHP 5.5

这两年 IT 更新换代的速度太快了,nginx 从 2 年前的 1.0 版本到现在的 1.5 版本，各个版本共同开发。PHP 从一开始的 php 5.2 到现在的 5.3/5.4/5.5.起草这篇文章的时候发现官方已经停止了 5.3 的开发,最后的版本定格到了 5.3.27,只会修复一些 bug，宣告了 5.3 寿终正寝了.并且建议大家升级到 5.4 或者 5.5，鉴于版本更新得如此之快,决定写一片 **nginx 连接 php5.5** 的文章.于是乎,本草文写完了.

1.安装 PHP 5.5.0

- 下载

```
cd /usr/local/src/
wget http://www.php.net/get/php-5.5.0.tar.bz2/from/jp1.php.net/mirror
```

如果以上 PHP 不存在了,大家可以直接到官方下载. 如果还是找不到可以留言,我将会通过邮箱发送.

- 安装依赖包

确保安装之前有安装 gd,png,curl,xml 等等 lib 开发库。如果不确定，执行以下命令：

```
yum install gcc make gd-devel libjpeg-devel libpng-devel libxml2-devel bzip2-devel libcurl-devel -y
```

- 编译安装 PHP 5.5.0

以下参数支持,ftp,图片函数,pdo 等支持,因为使用了 php 自带的 mysqlnd，所以不需要额外安装 mysql 的 lib 库了.如果你是 64 位系统，参数后面加上 –with-libdir=lib64，如果不是可以跳过。

```
tar -xjf php-5.5.0.tar.bz2
cd php-5.5.0
./configure --prefix=/usr/local/php-5.5.0 --with-config-file-path=/usr/local/php-5.5.0/etc --with-bz2 --with-curl --enable-ftp --enable-sockets --disable-ipv6 --with-gd --with-jpeg-dir=/usr/local --with-png-dir=/usr/local --with-freetype-dir=/usr/local --enable-gd-native-ttf --with-iconv-dir=/usr/local --enable-mbstring --enable-calendar --with-gettext --with-libxml-dir=/usr/local --with-zlib --with-pdo-mysql=mysqlnd --with-mysqli=mysqlnd --with-mysql=mysqlnd --enable-dom --enable-xml --enable-fpm --with-libdir=lib64
make
make install
```

备注：如果 PHP 不需要 curl 和 ftp 的支持,可以将以上的 –with-curl –enable-ftp 去掉. 如果你是专业的 linux 从业人员,你完全可以看着 help 来选择你的安装参数，如果你不是的话,我建议你直接复制黏贴我的配置参数.这样可以少走一些弯路.

- 配置 php

```
cp php.ini-production /usr/local/php-5.5.0/etc/php.ini
cp /usr/local/php-5.5.0/etc/php-fpm.conf.default /usr/local/php-5.5.0/etc/php-fpm.conf
```

- 启动 php-fpm

```
/usr/local/php-5.5.0/sbin/php-fpm
```

执行以上命令，如果没报错一般情况下表示启动正常,如果不放心,也可以通过端口判断是 PHP 否启动

```
# netstat -lnt | grep 9000
tcp 0 0 127.0.0.1:9000 0.0.0.0:* LISTEN
```

2、安装配置 nginx

- 安装 nginx

请看<ttlsa 教程系列之 nginx – nginx 安装>

- 配置测试站点 test.ttlsa.com

```
mkdir /data/logs/nginx/# 用于存放 nginx 日志.请看 2.3 的配置文件
mkdir -p /data/site/test.ttlsa.com/# 站点根目录
vim /data/site/test.ttlsa.com/info.php
```

保存退出

- nginx 配置
  在 nginx.conf 的 http 断中加上如下内容：

```
server {
  listen 80;
  server_name test.ttlsa.com;
  access_log /data/logs/nginx/test.ttlsa.com.access.log main;
  index index.php index.html index.html;
  root /data/site/test.ttlsa.com;
  location /
  {
    try_files $uri $uri/ /index.php?$args;
  }
  location ~ .*\.(php)?$
  {
    expires -1s;
    try_files $uri =404;
    fastcgi_split_path_info ^(.+\.php)(/.+)$;
    include fastcgi_params;
    fastcgi_param PATH_INFO $fastcgi_path_info;
    fastcgi_index index.php;
    fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    fastcgi_pass 127.0.0.1:9000;
  }
}
```

- 配置讲解
  nginx 将会连接回环地址 9000 端口执行 PHP 文件,需要使用 tcp/ip 协议,速度比较慢.建议大家换成使用 socket 方式连接。将 fastcgi_pass 127.0.0.1:9000;改成 fastcgi_pass unix:/var/run/phpfpm.sock;

- 启动 nginx
  /usr/local/nginx-1.4.1/sbin/nginx

3. 访问测试

```
# curl http://test.ttlsa.com/info.php
test php
```

出现如上内容,说明 PHP 安装完成。

(5)nginx 配置虚拟主机

本节主要讲解如果使用 nginx 配置多个虚拟主机,也就是我们通常说的配置域名.接下来我们配置两个域名 a.ttlsa.com，b.ttlsa.com。
如果你还不会安装 nginx 的话，请看第一节内容：ttlsa 教程系列之 nginx - nginx 安装(1)

准备站点

我们站点统一放到/data/site 下，每个站点根目录名称都和域名相同,具体如下。
新建 a.ttlsa.com 的站点根目录

```
# mkdir -p /data/site/a.ttlsa.com
```

- 新建 a 站的首页 index.html

```
# cat /data/site/a.ttlsa.com/index.html
this is a.ttlsa.com!
```

- 新建 b.ttlsa.com 站点根目录

```
# mkdir -p /data/site/b.ttlsa.com
```

- 新建 b 站首页 index.html,内容如 this is b.ttlsa.com!

```
# cat /data/site/b.ttlsa.com/index.html
this is b.ttlsa.com!
```

- 新建日志文件目录

```
# mkdir -p /data/logs/nginx
```

我们统一讲日志存放到/data/logs 下,这边是存放 nginx 日志,所以 nginx 日志保持在当前的 nginx 目录下.日志统一存放相对来说比较规范（如果你不习惯,你可以按自己的方式来做）

配置 nginx 虚拟主机

- 增加 nginx 主配置文件 nginx.conf

先配置 nginx 日志格式，在 nginx.conf 找到如下内容，并且将#注释标志去掉

```
#log_format main '$remote_addr - $remote_user [$time_local] "$request" '
# '$status $body_bytes_sent "$http_referer" '
# '"$http_user_agent" "$http_x_forwarded_for"';
```

- 配置 nginx 主配置文件

```
# vim /usr/local/nginx-1.5.1/conf/nginx.conf
server{
  server_name a.ttlsa.com;
  listen 80;
  root /data/site/a.ttlsa.com;
  access_log /data/logs/nginx/a.ttlsa.com-access.log main;
  location /
  {
  }
}
server{
  server_name b.ttlsa.com;
  listen 80;
  root /data/site/b.ttlsa.com;
  access_log /data/logs/nginx/b.ttlsa.com-access.log main;
  location /
  {
  }
}
```

- 配置讲解

server{}：配置虚拟主机必须有这个段。

server_name：虚拟主机的域名，可以写多个域名，类似于别名，比如说你可以配置成
server_name b.ttlsa.com c.ttlsa.com d.ttlsa.com，这样的话，访问任何一个域名，内容都是一样的
listen 80，监听 ip 和端口，这边仅仅只有端口，表示当前服务器所有 ip 的 80 端口，如果只想监听 127.0.0.1 的 80，写法如下：
listen 127.0.0.1:80
root /data/site/b.ttlsa.com：站点根目录，你网站文件存放的地方。注：站点目录和域名尽量一样，养成一个好习惯
access_log /data/logs/nginx/b.ttlsa.com-access.log main：访问日志
location /{} 默认 uri,location 具体内容后续讲解,大家关注一下.

重启并打开站点

nginx -t 检查 nginx 配置是否 ok，命令如下：

```
# /usr/local/nginx-1.5.1/sbin/nginx -t
nginx: the configuration file /usr/local/nginx-1.5.1/conf/nginx.conf syntax is ok
nginx: configuration file /usr/local/nginx-1.5.1/conf/nginx.conf test is successful
```

如果看到以上两行 ok 和 successful 就表示配置问题，那接下来我们启动 nginx

启动 nginx

```
# /usr/local/nginx-1.5.1/sbin/nginx
```

访问 a.ttlsa.com、b.ttlsa.com（我这边 DNS 已经解析到了 192.168.1.201，在测试的情况下，我们可以通过版本 hosts 即可），绑定 host 方法如下：
讲如下内容增加到 C:\Windows\System32\Drivers\etc\hosts

```
192.168.1.201 a.ttlsa.com
192.168.1.201 b.ttlsa.com
```

以上是 windows 绑定 hosts 方式，如下是 linux 方式

```
echo "192.168.1.201 a.ttlsa.com
192.168.1.201 b.ttlsa.com" >> /etc/hosts
```

使用浏览器访问这两个站点。我这边使用 curl 来访问。

```
[root@ns conf]# curl http://a.ttlsa.com
this is a.ttlsa.com! //a 站点内容
[root@ns conf]# curl http://b.ttlsa.com
this is b.ttlsa.com! //b 站点内容
```

其他指令

- 关闭 nginx
  `/usr/local/nginx-1.5.1/sbin/nginx -s stop`
- 重启 nginx
  `/usr/local/nginx-1.5.1/sbin/nginx -s reload //修改配置之后 reload,实际上严格意义来说这不是`

(6)nginx location 配置

今天讲下 location 的用法，部分内容是直接从网络上摘取的，这边做了一个整理，为了便于理解和学习，我这边做了一些例子。

语法规则： location [=|~|~*|^~] /uri/ { … }
= 表示精确匹配,这个优先级也是最高的
^~ 表示 uri 以某个常规字符串开头，理解为匹配 url 路径即可。nginx 不对 url 做编码，因此请求为 /static/20%/aa，可以被规则^~ /static/ /aa 匹配到（注意是空格）。
~ 表示区分大小写的正则匹配
~* 表示不区分大小写的正则匹配(和上面的唯一区别就是大小写)
!~和!~*分别为区分大小写不匹配及不区分大小写不匹配的正则
/ 通用匹配，任何请求都会匹配到，默认匹配.
下面讲讲这些语法的一些规则和优先级
多个 location 配置的情况下匹配顺序为（参考资料而来，还未实际验证，试试就知道了，不必拘泥，仅供参考）：

优先级=>^~>
首先匹配 =，其次匹配^~，其次是按文件中顺序的正则匹配，最后是交给 / 通用匹配。当有匹配成功时候，停止匹配，按当前匹配规则处理请求。
例子，有如下匹配规则：

```
location / {
   echo "/"; //需要安装 echo 模块才行,这边大家可以改成各自的规则
}
location = / {
   echo "=/";
}
location = /nginx {
   echo "=/nginx";
}
location ~ \.(gif|jpg|png|js|css)$ {
   echo "small-gif/jpg/png";
}
location ~* \.png$ {
   echo "all-png";
}
location ^~ /static/ {
   echo "static";
}
```

以下是各种的访问情况
访问 http://a.ttlsa.com/.因为/是完全匹配的
如下：

```
# curl http://a.ttlsa.com/
 =/
```

访问 http://a.ttlsa.com/nginx,因为完全匹配了”=/nginx”

```
# curl http://a.ttlsa.com/nginx
=/nginx
```

访问 http://a.ttlsa.com/nginx,从第一个开始尝试匹配,最后匹配到了~* \.png$.

```
# curl http://a.ttlsa.com/xxx/1111.PNG （注意,这是大写）
all-png
```

访问 http://a.ttlsa.com/static/1111.png,虽然 static 放在最后面,但是因为有^的缘故,他是最匹配的.

```
# curl http://a.ttlsa.com/static/1111.png
Static
```

好了，最后给出我们先上环境的静态文件的匹配规则

```
location ~* .*\.(js|css)?$
{
        expires 7d; //7 天过期,后续讲解
        access_log off; //不保存日志
}
location ~* .*\.(png|jpg|gif|jpeg|bmp|ico)?$
{
        expires 7d;
        access_log off;
}
location ~* .*\.(zip|rar|exe|msi|iso|gho|mp3|rmvb|mp4|wma|wmv|rm)?$
```

```
{
        deny all; //禁止这些文件下载，大家可以根据自己的环境来配置
}
```

(7)nginx root&alias 文件路径配置

nginx 指定文件路径有两种方式 root 和 alias，这两者的用法区别，使用方法总结了下，方便大家在应用过程中，快速响应。root 与 alias 主要区别在于 nginx 如何解释 location 后面的 uri，这会使两者分别以不同的方式将请求映射到服务器文件上。

[root]
语法：root path
默认值：root html
配置段：http、server、location、if

[alias]
语法：alias path
配置段：location

实例：

```
location ~ ^/weblogs/ {
        root /data/weblogs/www.ttlsa.com;
        autoindex on;
        auth_basic            "Restricted";
        auth_basic_user_file  passwd/weblogs;
}
```

如果一个请求的 URI 是/weblogs/httplogs/www.ttlsa.com-access.log 时，web 服务器将会返回服务器上的/data/weblogs/www.ttlsa.com/weblogs/httplogs/www.ttlsa.com-access.log 的文件。
[info]root 会根据完整的 URI 请求来映射，也就是/path/uri。[/info]
因此，前面的请求映射为 path/weblogs/httplogs/www.ttlsa.com-access.log。

```
location ^~ /binapp/ {
        limit_conn limit 4;
        limit_rate 200k;
        internal;
        alias /data/statics/bin/apps/;
}
```

alias 会把 location 后面配置的路径丢弃掉，把当前匹配到的目录指向到指定的目录。如果一个请求的 URI 是/binapp/a.ttlsa.com/favicon 时，web 服务器将会返回服务器上的/data/statics/bin/apps/a.ttlsa.com/favicon.jgp 的文件。
[warning]1. 使用 alias 时，目录名后面一定要加”/”。
2. alias 可以指定任何名称。
3. alias 在使用正则匹配时，必须捕捉要匹配的内容并在指定的内容处使用。
4. alias 只能位于 location 块中。[/warning]

ngx_http_core_module 模块在处理请求时，会有大量的变量，这些变量可以通过访问日志来记录下来，也可以用于其它 nginx 模块。在我们对请求做策略如改写等等都会使用到一些变量，顺便对 ngx_http_core_module 模块提供的变量总结了下，如下所示：

参数名称	注释
$arg_PARAMETER	HTTP 请求中某个参数的值，如/index.php?site=www.ttlsa.com，可以用$arg_site 取得 www.ttlsa.com 这个值.
$args HTTP	请求中的完整参数。例如，在请求/index.php?width=400&height=200 中，$args 表示字符串 width=400&height=200.
$binary_remote_addr	二进制格式的客户端地址。例如：\x0A\xE0B\x0E
$body_bytes_sent	表示在向客户端发送的 http 响应中，包体部分的字节数
$content_length	表示客户端请求头部中的 Content-Length 字段
$content_type	表示客户端请求头部中的 Content-Type 字段
$cookie_COOKIE	表示在客户端请求头部中的 cookie 字段
$document_root	表示当前请求所使用的 root 配置项的值
$uri	表示当前请求的 URI，不带任何参数
$document_uri	与$uri 含义相同
$request_uri	表示客户端发来的原始请求 URI，带完整的参数。$uri 和$document_uri 未必是用户的原始请求，在内部重定向后可能是重定向后的 URI，而$request_uri 永远不会改变，始终是客户端的原始 URI.
$host	表示客户端请求头部中的 Host 字段。如果 Host 字段不存在，则以实际处理的 server（虚拟主机）名称代替。如果 Host 字段中带有端口，如 IP:PORT，那么$host 是去掉端口的，它的值为 IP。$host 是全小写的。这些特性与 http_HEADER 中的 http_host 不同，http_host 只取出 Host 头部对应的值。
$hostname	表示 Nginx 所在机器的名称，与 gethostbyname 调用返回的值相同
$http_HEADER	表示当前 HTTP 请求中相应头部的值。HEADER 名称全小写。例如，示请求中 Host 头部对应的值　用 $http_host 表
$sent_http_HEADER	表示返回客户端的 HTTP 响应中相应头部的值。HEADER 名称全小写。例如，用 $sent_http_content_type 表示响应中 Content-Type 头部对应的值
$is_args	表示请求中的 URI 是否带参数，如果带参数，$is_args 值为 ?，如果不带参数，则是空字符串
$limit_rate	表示当前连接的限速是多少，0 表示无限速
$nginx_version	表示当前 Nginx 的版本号
$query_string	请求 URI 中的参数，与 $args 相同，然而 $query_string 是只读的不会改变
$remote_addr	表示客户端的地址
$remote_port	表示客户端连接使用的端口
$remote_user	表示使用 Auth Basic Module 时定义的用户名
$request_filename	表示用户请求中的 URI 经过 root 或 alias 转换后的文件路径
$request_body	表示 HTTP 请求中的包体，该参数只在 proxy_pass 或 fastcgi_pass 中有意义
$request_body_file	表示 HTTP 请求中的包体存储的临时文件名
$request_completion	当请求已经全部完成时，其值为 “ok”。若没有完成，就要返回客户端，则其值为空字符串；或者在断点续传等情况下使用 HTTP range 访问的并不是文件的最后一块，那么其值也是空字符串。
$request_method	表示 HTTP 请求的方法名，如 GET、PUT、POST 等
$scheme	表示 HTTP scheme，如在请求 https://nginx.com/中表示 https
$server_addr	表示服务器地址
$server_name	表示服务器名称
$server_port	表示服务器端口
$server_protocol	表示服务器向客户端发送响应的协议，如 HTTP/1.1 或 HTTP/1.0

(8)nginx 日志配置

日志对于统计排错来说非常有利的。本文总结了 nginx 日志相关的配置如 access_log、log_format、open_log_file_cache、log_not_found、log_subrequest、rewrite_log、error_log。
nginx 有一个非常灵活的日志记录模式。每个级别的配置可以有各自独立的访问日志。日志格式通过 log_format 命令来定义。ngx_http_log_module 是用来定义请求日志格式的。

1. access_log 指令

语法: access_log path [format [buffer=size [flush=time]]];
access_log path format gzip[=level] [buffer=size] [flush=time];
access_log syslog:server=address[,parameter=value] [format];
access_log off;
默认值: access_log logs/access.log combined;
配置段: http, server, location, if in location, limit_except
gzip 压缩等级。
buffer 设置内存缓存区大小。
flush 保存在缓存区中的最长时间。
不记录日志：access_log off;
使用默认 combined 格式记录日志：access_log logs/access.log 或 access_log logs/access.log combined;

2. log_format 指令

语法: log_format name string …;
默认值: log_format combined “…”;
配置段: http
name 表示格式名称，string 表示等义的格式。log_format 有一个默认的无需设置的 combined 日志格式，相当于 apache 的 combined 日志格式，如下所示：

```
log_format   combined    '$remote_addr - $remote_user   [$time_local]   '
                         '"$request"   $status   $body_bytes_sent   '
                         '"$http_referer"   "$http_user_agent"';
```

如果 nginx 位于负载均衡器，squid，nginx 反向代理之后，web 服务器无法直接获取到客户端真实的 IP 地址了。$remote_addr 获取反向代理的 IP 地址。反向代理服务器在转发请求的 http 头信息中，可以增加 X-Forwarded-For 信息，用来记录 客户端 IP 地址和客户端请求的服务器地址。PS: 获取用户真实 IP 参见 http://www.ttlsa.com/html/2235.html 如下所示：

```
log_format   porxy    '$http_x_forwarded_for - $remote_user   [$time_local]   '
                      '"$request"   $status $body_bytes_sent '
                      '"$http_referer"   "$http_user_agent"';
```

日志格式允许包含的变量注释如下：

view sourceprint?
$remote_addr, $http_x_forwarded_for 记录客户端 IP 地址
$remote_user 记录客户端用户名称
$request 记录请求的 URL 和 HTTP 协议
$status 记录请求状态
$body_bytes_sent 发送给客户端的字节数，不包括响应头的大小； 该变量与 Apache 模块 mod_log_config 里的“%B”参数兼容。
$bytes_sent 发送给客户端的总字节数。
$connection 连接的序列号。
$connection_requests 当前通过一个连接获得的请求数量。
$msec 日志写入时间。单位为秒，精度是毫秒。
$pipe 如果请求是通过 HTTP 流水线(pipelined)发送，pipe 值为“p”，否则为“.”。

$http_referer 记录从哪个页面链接访问过来的
$http_user_agent 记录客户端浏览器相关信息
$request_length 请求的长度（包括请求行，请求头和请求正文）。
$request_time 请求处理时间，单位为秒，精度毫秒； 从读入客户端的第一个字节开始，直到把最后一个字符发送给客户端后进行日志写入为止。
$time_iso8601 ISO8601 标准格式下的本地时间。
$time_local 通用日志格式下的本地时间。

[warning]发送给客户端的响应头拥有“sent_http_”前缀。 比如$sent_http_content_range。[/warning]

实例如下：

```
http {
    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                                    '"$status" $body_bytes_sent "$http_referer" '
                                    '"$http_user_agent" "$http_x_forwarded_for" '
                                    '"$gzip_ratio" $request_time $bytes_sent $request_length';

    log_format srcache_log '$remote_addr - $remote_user [$time_local] "$request" '
                                '"$status"      $body_bytes_sent      $request_time      $bytes_sent
$request_length '
                                '[$upstream_response_time]                  [$srcache_fetch_status]
[$srcache_store_status] [$srcache_expire]';

    open_log_file_cache max=1000 inactive=60s;

    server {
        server_name ~^(www\.)?(.+)$;
        access_log logs/$2-access.log main;
        error_log logs/$2-error.log;

        location /srcache {
            access_log logs/access-srcache.log srcache_log;
        }
    }
}
```

3. open_log_file_cache 指令

语法: open_log_file_cache max=N [inactive=time] [min_uses=N] [valid=time];
open_log_file_cache off;
默认值: open_log_file_cache off;
配置段: http, server, location

对于每一条日志记录，都将是先打开文件，再写入日志，然后关闭。可以使用 open_log_file_cache 来设置日志文件缓存(默认是 off)，格式如下：
参数注释如下：
max:设置缓存中的最大文件描述符数量，如果缓存被占满，采用 LRU 算法将描述符关闭。
inactive:设置存活时间，默认是 10s
min_uses:设置在 inactive 时间段内，日志文件最少使用多少次后，该日志文件描述符记入缓存中，默认是 1 次
valid:设置检查频率，默认 60s

off：禁用缓存
实例如下：

```
open_log_file_cache max=1000 inactive=20s valid=1m min_uses=2;
```

4. log_not_found 指令

语法：log_not_found on | off;
默认值：log_not_found on;
配置段：http, server, location
是否在 error_log 中记录不存在的错误。默认是。

5. log_subrequest 指令

语法：log_subrequest on | off;
默认值：log_subrequest off;
配置段：http, server, location
是否在 access_log 中记录子请求的访问日志。默认不记录。

6. rewrite_log 指令

由 ngx_http_rewrite_module 模块提供的。用来记录重写日志的。对于调试重写规则建议开启。 Nginx 重写规则指南
语法：rewrite_log on | off;
默认值：rewrite_log off;
配置段：http, server, location, if
启用时将在 error log 中记录 notice 级别的重写日志。

7. error_log 指令

语法：error_log file | stderr | syslog:server=address[,parameter=value] [debug | info | notice | warn | error | crit | alert | emerg];
默认值：error_log logs/error.log error;
配置段：main, http, server, location
配置错误日志。

(9)apache 和 nginx 支持 SSI 配置

一. 前言

SSI 是一种类似于 ASP 的基于服务器的网页制作技术。将内容发送到浏览器之前，可以使用“服务器端包含（SSI）”指令将文本、图形或应用程序信息包含到网页中。例如，可以使用 SSI 包含时间/日期戳、版权声明或供客户填写并返回的表单。对于在多个文件中重复出现的文本或图形，使用包含文件是一种简便的方法。将内容存入一个包含文件中即可，而不必将内容输入所有文件。通过一个非常简单的语句即可调用包含文件，此语句指示 Web 服务器将内容插入适当网页。而且，使用包含文件时，对内容的所有更改只需在一个地方就能完成。
因为包含 SSI 指令的文件要求特殊处理，所以必须为所有 SSI 文件赋予 SSI 文件扩展名。默认扩展名是 .stm、.shtm 和 .shtml

二. apache 配置

apache 默认不支持 ssi 的，可以在 apache 下做如下设置：
修改 Apache 配置文件 httpd.conf
1. 确认加载 include.so 模块，将注释去掉：

```
LoadModule include_module libexec/apache2/mod_include.so
```

2. AddType 部分去掉这两段注释：

AddType text/html .shtml

AddOutputFilter INCLUDES .shtml

3. Directory 目录权限里面找到

Options Indexes FollowSymLinks

增加 Includes 修改为：

Options Indexes FollowSymLinks Includes

4. 重新启动 Apache

三. nginx 配置

1. 相关指令说明：

ssi

语法：ssi [on | off]

默认值：ssi off

配置段段：http, server, location, if

启用 SSI 处理。

[warning]注意如果启用 SSI，那么 Last-Modified 头和 Content-Length 头不会传递。[/warning]

ssi_silent_errors

语法：ssi_silent_errors [on|off]

默认值：ssi_silent_errors off

配置段：http, server, location

如果在处理 SSI 的过程中出现“[an error occurred while processing the directive]”错误，禁止将其输出。

ssi_types

语法：ssi_types mime-type [mime-type …]

默认值：ssi_types text/html

配置段：http, server, location

默认只解析 text/html 类型，这个参数可以指定其他的 MIME 类型。

ssi_value_length

语法：ssi_value_length length

默认值：ssi_value_length 256

配置段：http, server, location

定义允许 SSI 使用的参数值的长度。

2. 在 nginx 下做如下设置：

在 http 段添加：

```
ssi on;
ssi_silent_errors off;
ssi_types text/shtml;
```

或

```
location ~* \.shtml$ {
        ssi on;
        ssi_silent_errors off;
        ssi_value_length 1024;
        ssi_types text/shtml;
}
```

或

```
location / {
        ssi on;
        ssi_silent_errors on;
        ssi_value_length 1024;
        ssi_types text/shtml;
}
```


(10)nginx 日志切割

nginx 日志默认情况下统统写入到一个文件中，文件会变的越来越大，非常不方便查看分析。以日期来作为日志的切割是比较好的，通常我们是以每日来做统计的。下面来说说 nginx 日志切割。

关于 nginx 相关日志配置参见：《nginx 日志配置》一文。logrotate 用法参见《logrotate 日志管理工具》。

1. 定义日志轮滚策略

```
# vim nginx-log-rotate
/data/weblogs/*.log {
    nocompress
    daily
    copytruncate
    create
    notifempty
    rotate 7
    olddir /data/weblogs/old_log
    missingok
    dateext
    postrotate
        /bin/kill -HUP `cat /var/run/nginx.pid 2> /dev/null` 2> /dev/null || true
    endscript
}
```

[warning]/data/weblogs/*.log 使用通配符时，/data/weblogs/目录下的所有匹配到的日志文件都将切割。如果要切割特定日志文件，就指定到该文件。[/warning]

2. 设置计划任务

```
# vim /etc/crontab
59 23 * * * root ( /usr/sbin/logrotate -f /PATH/TO/nginx-log-rotate)
```

这样每天 23 点 59 分钟执行日志切割。

(11)Nginx 重写规则指南

当运维遇到要重写情况时，往往是要程序员把重写规则写好后，发给你，你再到生产环境下配置。对于重写规则说到底就是正则匹配，做运维的岂能对正则表达式不了解的？最起码最基本的正则表达式会写。套用一句阿里的话(某网友说是阿里说的，不清楚到底是不是出自阿里)“不懂程序的运维，不是好运维；不懂运维的开发，不是好开发。”。 正则表达式也是一门语言哈。当你学习一门语言时，必然会遇到该门语言的正则表达式这章节的。在这里推荐一本非常好的正则表达式书，包含常用的语言的正则写法如 sed、perl、bash、awk、php、c#、java、javascript、python、ruby 等等，《Regular Expressions Cookbook, 2nd Edition》，也有中文版的，大家可以到网络上找找。
介绍 nginx 的重写模块，创建重写规则向导，便于快捷正确的创建新的重写规则，不求救于人。同时，如果想把 apache 转换成 nginx，重写规则也是要改的咯。

一. rewrite 模块介绍

nginx 的重写模块是一个简单的正则表达式匹配与一个虚拟堆叠机结合。依赖于 PCRE 库，因此需要安装 pcre。根据相关变量重定向和选择不同的配置，从一个 location 跳转到另一个 location，不过这样的循环最多可以执行 10 次，超过后 nginx 将返回 500 错误。同时，重写模块包含 set 指令，来创建新的变量并设其值，这在有些情景下非常有用的，如记录条件标识、传递参数到其他 location、记录做了什么等等。

二. rewrite 模块指令

break
语法：break
默认值：none
使用字段：server, location, if
完成当前设置的重写规则，停止执行其他的重写规则。

if
语法：if (condition) { … }
默认值：none
使用字段：server, location
注意：尽量考虑使用 trp_files 代替。
判断的条件可以有以下值：
1. 一个变量的名称：空字符传”“或者一些“0”开始的字符串为 false。
2. 字符串比较：使用=或!=运算符
3. 正则表达式匹配：使用~(区分大小写)和~*(不区分大小写)，取反运算!~和!~*。
4. 文件是否存在：使用-f 和!-f 操作符
5. 目录是否存在：使用-d 和!-d 操作符
7. 文件、目录、符号链接是否存在：使用-e 和!-e 操作符
8. 文件是否可执行：使用-x 和!-x 操作符

return
语法：return code
默认值：none
使用字段：server, location, if

停止处理并为客户端返回状态码。非标准的 444 状态码将关闭连接，不发送任何响应头。可以使用的状态码有：204，400，402-406，408，410，411，413，416 与 500-504。如果状态码附带文字段落，该文本将被放置在响应主体。相反，如果状态码后面是一个 URL，该 URL 将成为 location 头补值。没有状态码的 URL 将被视为一个 302 状态码。

rewrite
语法：rewrite regex replacement flag
默认值：none
使用字段：server，location，if
按照相关的正则表达式与字符串修改 URI，指令按照在配置文件中出现的顺序执行。可以在重写指令后面添加标记。
注意：如果替换的字符串以 http://开头，请求将被重定向，并且不再执行多余的 rewrite 指令。
尾部的标记(flag)可以是以下的值：
last - 停止处理重写模块指令，之后搜索 location 与更改后的 URI 匹配。
break - 完成重写指令。
redirect - 返回 302 临时重定向，如果替换字段用 http://开头则被使用。
permanent - 返回 301 永久重定向。

rewrite_log
语法：rewrite_log on | off
默认值：rewrite_log off
使用字段：server，location，if
变量：无
启用时将在 error log 中记录 notice 级别的重写日志。

set
语法：set variable value
默认值：none
使用字段：server，location，if
为给定的变量设置一个特定值。

uninitialized_variable_warn
语法：uninitialized_variable_warn on|off
默认值：uninitialized_variable_warn on
使用字段：http，server，location，if
控制是否记录未初始化变量的警告信息。

三. 重写规则组成部分

3.1 任何重写规则的第一部分都是一个正则表达式
可以使用括号来捕获，后续可以根据位置来将其引用，位置变量值取决于捕获正则表达式中的顺序，$1 引用第一个括号中的值，$2 引用第二个括号中的值，以此类推。如：

```
^/images/([a-z]{2})/([a-z0-9]{5})/(.*)\.(png|jpg|gif)$
```

$1 是两个小写字母组成的字符串，$2 是由小写字母和 0 到 9 的数字组成的 5 个字符的字符串，$3 将是个文件名，$4 是 png、jpg、gif 中的其中一个。

3.2 重写规则的第二部分是 URI
请求被改写。该 URI 可能包含正则表达式中的捕获的位置参数或这个级别下的 nginx 任何配置变量。如：

```
/data?file=$3.$4
```

如果这个 URI 不匹配 nginx 配置的任何 location，那么将给客户端返回 301(永久重定向)或 302(临时重定向)的状态码来表示重定向类型。该状态码可以通过第三个参数来明确指定。

3.3 重写规则的第三部分

第三部分也就是尾部的标记(flag)。 last 标记将导致重写后的 URI 搜索匹配 nginx 的其他 location，最多可循环 10 次。如：

```
rewrite  '^/images/([a-z]{2})/([a-z0-9]{5})/(.*)\.(png|jpg|gif)$'  /data?file=$3.$4 last;
```

break 指令可以当做自身指令。如：

```
if ($bwhog) {
    limit_rate 300k;
    break;
}
```

另一个停止重写模块处理指令是 return， 来控制主 HTTP 模块处理请求。 这意味着，nginx 直接返回信息给客户端，与 error_page 结合为客户端呈现格式化的 HTML 页面或激活不同的模块来完成请求。如果状态码附带文字段落，该文本将被放置在响应主体。相反，如果状态码后面是一个 URL，该 URL 将成为 location 头补值。没有状态码的 URL 将被视为一个 302 状态码。如：

```
location = /image404.html {
        return  404  "image not found\n";
}
```

四. 实例

```
http {
    # 定义 image 日志格式
    log_format imagelog '[$time_local] ' $image_file ' ' $image_type ' ' $body_bytes_sent ' ' $status;
    # 开启重写日志
    rewrite_log on;

    server {
        root /home/www;

        location / {
                # 重写规则信息
                error_log logs/rewrite.log notice;
                # 注意这里要用 ‘’ 单引号引起来，避免{}
                rewrite '^/images/([a-z]{2})/([a-z0-9]{5})/(.*)\.(png|jpg|gif)$' /data?file=$3.$4;
                # 注意不能在上面这条规则后面加上“last”参数，否则下面的 set 指令不会执行
                set $image_file $3;
                set $image_type $4;
        }
        location /data {
                # 指定针对图片的日志格式，来分析图片类型和大小
                access_log logs/images.log mian;
                root /data/images;
                # 应用前面定义的变量。判断首先文件在不在，不在再判断目录在不在，如果还不在就跳转到最后一个 url 里
                try_files /$arg_file /image404.html;
        }
        location = /image404.html {
                # 图片不存在返回特定的信息
                return 404 "image not found\n";
        }
}
```

五.创建新的重新规则

在接到要创建新的重写规则时，要弄清楚需求是什么样的，再决定怎么做。毕竟重写也是耗资源的有效率之分的。 下面的这些问题有些帮助的：

1. 你的 URL 的模式是什么样的?
2. 是否有一个以上的方法来实现?
3. 是否需要捕获 URL 部分作为变量?
4. 重定向到另一个 web 上可以看到我的规则?
5. 是否要替换查询的字符串参数?

检查网站或应用程序布局，清楚 URL 模式。啰嗦一句：我一而再再而三的强调，运维不能与开发脱节，运维要参与到开发当中。如果有不止一种方法实现，创建一个永久重定向。同时，定义一个重写规范，来使网址清洁，还可以帮助网站更容易被找到。

实例 1. 要将 home 目录重定向到主页面上，目录结构如下：

```
/
/home
/home/
/home/index
/home/index/
/index
/index.php
/index.php/
```

重写规则如下：

```
rewrite ^/(home(/index)?|index(\.php)?)/?$ $scheme:
//$host/ permanent;
```

指定$scheme 和$host 变量，因为要做一个永久重定向并希望 nginx 使用相同的参数来构造 URL。

实例 2. 如果想分别记录各个部分的 URL，可以使用正则表达式来捕获 URI，然后，给变量分配指定位置变量，见上面的实例。

实例 3. 当重写规则导致内部重定向或指示客户端调用该规则本身被定义的 location 时，必须采取特殊的动作来避免重写循环。如：在 server 配置段定义了一条规则带上 last 标志，在引用 location 时，必须使用 break 标志。

```
server {
    rewrite ^(/images)/(.*)\.(png|jpg|gif)$ $1/$3/$2.$3 last;
    location /images/ {
        rewrite ^(/images)/(.*)\.(png|jpg|gif)$ $1/$3/$2.$3 break;
    }
}
```

实例 4. 作为重写规则的一部分，传递新的查询字符串参数是使用重写规则的目标之一。 如：

```
rewrite ^/images/(.*)_(\d+)x(\d+)\.(png|jpg|gif)$ /resizer/$1.$4?width=$2&height=$3? last;
```

nginx 重写规则说起来挺简单的，做起来就难，重点在于正则表达式，同时，还需要考虑到 nginx 执行顺序。

(12)nginx 逻辑运算

nginx 的配置中不支持 if 条件的逻辑与&& 逻辑或|| 运算 ，而且不支持 if 的嵌套语法，否则会报下面的错误：nginx: [emerg] invalid condition。

我们可以用变量的方式来间接实现。

要实现的语句：

```
if  ($arg_unitid = 42012 && $uri ~/thumb/){
echo  "www.ttlsa.com";
}
```

如果按照这样来配置，就会报 nginx: [emerg] invalid condition 错误。
可以这么来实现，如下所示：

```
set  $flag 0;
if ($uri ~ ^/thumb/[0-9]+_160.jpg$){
 set $flag "${flag}1";
}
if ($arg_unitid = 42012){
 set $flag "${flag}1";
}
if ($flag = "011"){
 echo "www.ttlsa.com";
}
```

(13)隐藏 Nginx 版本号的安全性与方法

搭建好 nginx 或者 apache，为了安全起见我们都会隐藏他们的版本号，这边讲的是 nginx 的版本号，如果你也想隐藏 apache 的版本号，那请点前面的链接。请看 nginx 版本号信息隐藏文章。
Nginx 默认是显示版本号的，如：

```
[root@bkjz ~]# curl -I www.nginx.org
HTTP/1.1 200 OK
Server: nginx/0.8.44
Date: Tue, 13 Jul 2010 14:05:11 GMT
Content-Type: text/html
Content-Length: 8284
Last-Modified: Tue, 13 Jul 2010 12:00:13 GMT
Connection: keep-alive
Keep-Alive: timeout=15
Accept-Ranges: bytes
```

这样就给人家看到你的服务器 nginx 版本是 0.8.44，前些时间暴出了一些 Nginx 版本漏洞，就是说有些版本有漏洞，而有些版本没有。这样暴露出来的版本号就容易变成攻击者可利用的信息。所以，从安全的角度来说，隐藏版本号会相对安全些！
那 nginx 版本号可以隐藏不？其实可以的，看下面的步骤：
1、进入 nginx 配置文件的目录（此目录根据安装时决定），用 vim 编辑打开

```
# vim nginx.conf
```

在 http {—}里加上 **server_tokens off;**　如：

```
http {
……省略
sendfile on;
tcp_nopush on;
keepalive_timeout 60;
tcp_nodelay on;
server_tokens off;
…….省略
}
```

2、编辑 php-fpm 配置文件，如 fastcgi.conf 或 fcgi.conf（这个配置文件名也可以自定义的，根据具体文件名修改）：

```
找到：
fastcgi_param SERVER_SOFTWARE nginx/$nginx_version;
改为：
fastcgi_param SERVER_SOFTWARE nginx;
```

3、重新加载 nginx 配置：

```
# /etc/init.d/nginx reload
```

这样就完全对外隐藏了 nginx 版本号了，就是出现 404、501 等页面也不会显示 nginx 版本。
下面测试一下：

```
# curl -I www.abc.net
HTTP/1.1 200 OK
Server: nginx
Date: Tue, 13 Jul 2010 14:26:56 GMT
Content-Type: text/html; charset=UTF-8
Connection: keep-alive
Vary: Accept-Encoding
```

(14)CDN 调度器 HAProxy、Nginx、Varnish

CDN 功能如下：
1、将全网 IP 分为若干个 IP 段组，分组的依据通常是运营商或者地域，目的是让相同网络环境中的用户聚集到相同的组内；
2、依据 CDN 服务器们的网络和容量，确定哪些 CDN 服务器适合服务哪些 IP 段组；
3、根据以上两步得到的结论，让用户去最适合他的服务器得到服务。

说白了，就是根据用户不同的来源 IP 把用户请求重定向到不同的 CDN 服务器上去。
那么，如何实现呢？

智能 DNS 是办法之一，稳定可靠且有效。
但至少在两个环境下它不能完全满足我们：
1、需要特别精细的调度时。由于大多数 DNS Server 不支持 DNS 扩展协议，所以拿不到用户的真实 IP，只能根据 Local DNS 来调度。
2、访问特别频繁时。由于每次调度都将触发一次 DNS，如果请求变得密集，DNS 请求本身带来的开销也会相应变大；
3、需要根据服务器的带宽容量、连接数、负载情况、当机与否来调度时。由于 DNS Server 没有 CDN 节点服务器的信息，这种调度会变得困难。

这时候我们可以：
1、将用户先行引导到某一台或几台统一的服务器上去；
2、让它拿到用户的真实 IP，计算出服务他的服务器；
3、通过 HTTP302 或其它方式把用户定位到最终服务器上。

部署在用户先访问到的那几台服务器上，负责定位 IP 然后重定向用户请求的那个软件，我们叫它“调度器”。

HAProxy 实现：
HAProxy 不支持形如 0.0.0.1-0.8.255.255 cn 的 IP 段表示方法，只支持 1.1.4.0/22 “CN”的 IP 段表示方法。
1、我们需要先把 IP 段转化成它认识的方式；

a> 下载 iprang.c 或者 iprang.c 本地镜像；
b> 编译 gcc -s -O3 -o iprange iprange.c；
c> 整理 IP 段列表 geo.txt 形如：

```
1 # head geo.txt
2 "1.0.0.0","1.0.0.255","AU"
3 "1.0.1.0","1.0.3.255","CN"
4 "1.0.4.0","1.0.7.255","AU"
5 "1.0.8.0","1.0.15.255","CN"
6 "1.0.16.0","1.0.31.255","JP"
7 "1.0.32.0","1.0.63.255","CN"
8 "1.0.64.0","1.0.127.255","JP"
9 "1.0.128.0","1.0.255.255","TH"
10 "1.1.0.0","1.1.0.255","CN"
11 "1.1.1.0","1.1.1.255","AU"
```

d> 输出 HAProxy 认识的 IP 段列表：

```
1# cut -d, -f1,2,5 geo.txt | ./iprange | head
2  1.0.0.0/24 "AU"
3  1.0.1.0/24 "CN"
4  1.0.2.0/23 "CN"
5  1.0.4.0/22 "AU"
6  1.0.8.0/21 "CN"
7  1.0.16.0/20 "JP"
8  1.0.32.0/19 "CN"
9  1.0.64.0/18 "JP"
10 1.0.128.0/17 "TH"
11 1.1.0.0/24 "CN"
12 1.1.1.0/24 "AU"
```

e> 便于管理的目的，将整合后的 IP 段归类到同一个文件中：

```
1  # cut -d, -f1,2,5 geo.txt | ./iprange | sed 's/"//g' | awk -F' ' '{ print $1 >> $2".subnets" }'
2  # ls *.subnets
3  A1.subnets  AX.subnets  BW.subnets  CX.subnets  FJ.subnets  GR.subnets  IR.subnets  LA.subnets
ML.subnets  NF.subnets  PR.subnets  SI.subnets  TK.subnets  VE.subnets
4  # cat AU.subnets
5  1.0.0.0/24
6  1.0.4.0/22
7  1.1.1.0/24
```

f> 把这些文件放到同一个文件夹下，我们以/etc/haproxy/conf/为例。

2、正确配置 HAProxy 以这些 IP 段为规则正确调度；
下面是一个 haproxy.cfg 的例子。配置好后重启 Haproxy 即可。

```
1  global
2    log         127.0.0.1 local2 debug
3
4    chroot      /var/lib/haproxy
5    pidfile     /var/run/haproxy.pid
```

```
    maxconn     8000
    user        haproxy
    group       haproxy
    daemon

     stats socket /var/lib/haproxy/stats

defaults
    mode                    http
    log                     global
    option                  httplog
    option                  dontlognull
    option http-server-close
    option forwardfor       except 127.0.0.0/8
    option                  redispatch
    retries                 3
    timeout http-request    10s
    timeout queue           1m
    timeout connect         10s
    timeout client          1m
    timeout server          1m
    timeout http-keep-alive 10s
    timeout check           10s
    maxconn                 8000

frontend  main *:5000
     acl geo_A1 src -f /etc/haproxy/conf/A1.subnets
    acl geo_AX src -f /etc/haproxy/conf/AX.subnets
    acl geo_BW src -f /etc/haproxy/conf/BW.subnets
    acl geo_CX src -f /etc/haproxy/conf/CX.subnets
    acl geo_FJ src -f /etc/haproxy/conf/FJ.subnets

    ...

    reqrep ^([^\ ]*)\ /(.*)\ HTTP    \1\ /\2&ipfrom=A1\ HTTP if geo_A1
    reqrep ^([^\ ]*)\ /(.*)\ HTTP    \1\ /\2&ipfrom=AX\ HTTP if geo_AX
    reqrep ^([^\ ]*)\ /(.*)\ HTTP    \1\ /\2&ipfrom=BW\ HTTP if geo_BW
    reqrep ^([^\ ]*)\ /(.*)\ HTTP    \1\ /\2&ipfrom=CX\ HTTP if geo_CX
    reqrep ^([^\ ]*)\ /(.*)\ HTTP    \1\ /\2&ipfrom=FJ\ HTTP if geo_FJ

    ...

    default_backend             static

backend static
    server      static 127.0.0.1:6081 check
```

Nginx 实现：

Nginx 可以在核心模块 HttpGeoModule（http://wiki.nginx.org/HttpGeoModule）的配合下实现调度：

```
http{

...

geo $useriprang {
  ranges;
  default a;
  0.0.0.1-0.8.255.255 a;
  0.9.0.0-0.255.255.255   a;
   1.0.0.0-1.0.0.255   a;
  1.0.1.0-1.0.1.255   b;
  1.0.2.0-1.0.3.255   b;
  1.0.4.0-1.0.7.255   a;
  ...
  223.255.252.0-223.255.253.255   c;
  223.255.254.0-223.255.254.255   a;
  223.255.255.0-223.255.255.255   a;
}

upstream backend {
  server 127.0.0.1:81;
}

server {
  listen       80;
  client_max_body_size 10240m;

  location / {
      proxy_redirect off;
      proxy_pass http://backend$request_uri&useriprang=$useriprang;
      proxy_next_upstream http_502 http_504 error timeout invalid_header;
      proxy_cache cache_one;
      proxy_cache_key $host:$server_port$uri$is_args$args;
      expires  5s;
  }

}

...

}
```

Varnish 实现：

Varnish 则有两个插件可以实现调度：

https://github.com/cosimo/varnish-geoip （Last updated: 28/05/2013）

https://github.com/meetup/varnish-geoip-plugin （Last updated: 2010）

性能问题

如上所述，使用 Haproxy、Nginx、Varnish 都能快速实现这个功能。

其中 Nginx 和 Varnish 使用了二分法在 IP 表中定位用户 IP，而 Haproxy 是逐条过滤。

所以在 IP 分得较细，IP 段组较多（归类后超过 1000 组）时，Haproxy 会出现明显的性能衰减，其余两者没有这个问题。

其它
本文使用的软件版本如下：
HAProxy1.4.22，Nginx1.2.9，Varnish3.0.4。
HAProxy 和 Varnish 都是目前的最新版本。

(15)lnmp 架构下 php 安全配置分享

以往的 lamp 网站向着 lnmp 发展，笔者工作环境使用 lnmp 多年，在这里很高兴和大家分享一下多年的 lnmp 网站的 php 安全配置，至于 lamp 安全后续与大家分享，其实内容上八成相同，这边着重讲 php 安全配置，看内容.

1. 使用 open_basedir 限制虚拟主机跨目录访问

```
[HOST=www.ttlsa.com]
open_basedir=/data/site/www.ttlsa.com/:/tmp/

[HOST=test.ttlsa.com]
open_basedir=/data/site/test.ttlsa.com/:/tmp/
```

如上配置的意思是 www.ttlsa.com 下的 php 程序被限制在 open_basedir 配置的两个目录下，不可以访问到其他目录。如果没有做以上的配置，那么 test.ttlsa.com 与 www.ttlsa.com 的程序可以互相访问.
如果其中一个站点有漏洞被黑客植入了 webshell，那么他可以通过这个站点拿下同一台服务器的其他站点，最后挂木马.

[warning]注意：目录最后一定要加上/. 比如你写/tmp，你的站点同时存在/tmp123 等等以/tmp 开头的目录，那么黑客也可以访问到这些目录，另外，php5.3 以上支持这个写法，5.2 不支持。[/warning]

2. 禁用不安全 PHP 函数

```
disable_functions = show_source,system,shell_exec,passthru,exec,popen,proc_open,proc_get_status,phpinfo
```

禁止 php 执行以上 php 函数,以上 php 程序可以执行 linux 命令，比如可以执行 ping、netstat、mysql 等等.如果你的系统有提权 bug,后果你懂得.

3. 关注软件安全资讯

积极关注 linux 内核、php 安全等信息并及时采取错误

4. php 用户只读

这个方法是我最推崇的方法，但是执行之前一定要和 php 工程师商量. 为什么？例如站点 www.ttlsa.com 根目录用户与组为 nobody，而运行 php 的用户和组为 phpuser。目录权限为 755，文件权限为 644. 如此，php 为只读，无法写入任何文件到站点目录下。也就是说用户不能上传文件，即使有漏洞，黑客也传不了后门，更不可能挂木马.　　这么干之前告知程序员将文件缓存改为 nosql 内存缓存（例如 memcached、redis 等），上传的文件通过接口传到其他服务器（静态服务器）。

[warning]备注：程序生成本地缓存是个非常糟糕的习惯，使用文件缓存速度缓慢、浪费磁盘空间、最重要一点是一般情况下服务器无法横向扩展.[/warning]

5. 关闭 php 错误日志

```
display_errors = On
改为
display_errors = Off
```

程序一旦出现错误，详细错误信息便立刻展示到用户眼前，其中包含路径、有的甚至是数据库账号密码. 注入渗透密码基本上都是通过这个报错来猜取。生产环境上强烈关闭它

6. php 上传分离

将文件上传到远程服务器，例如 nfs 等。当然也可以调用你们写好的 php 接口. 即使有上传漏洞，那么文件也被传到了静态服务器上。木马等文件根本无法执行.

举个例子：
php 站点 www.ttlsa.com，目录/data/site/www.ttlsa.com
静态文件站点 static.ttlsa.com，目录/data/site/static.ttlsa.com

文件直接被传到了/data/site/static.ttlsa.com，上传的文件无法通过 www.ttlsa.com 来访问，只能使用 static.ttlsa.com 访问，但是 static.ttlsa.com 不支持 php.

7. 关闭 php 信息

```
expose_php = On
改为
expose_php = Off
```

不轻易透露自己 php 版本信息，防止黑客针对这个版本的 php 发动攻击.

8. 禁止动态加载链接库

```
disable_dl = On;
改为
enable_dl = Off;
```

9. 禁用打开远程 url

```
allow_url_fopen = On
改为
allow_url_fopen = Off
```

其实这点算不上真正的安全，并不会导致 web 被入侵等问题,但是这个非常影响性能，笔者认为它属于狭义的安全问题.

以下方法将无法获取远程 url 内容

```
$data = file_get_contents("http://www.baidu.com/");
```

以下方法可以获取本地文件内容

```
$data = file_get_contents("1.txt");
```

如果你的站点访问量不大、数据库也运行良好，但是 web 服务器负载出奇的高，请你直接检查下是否有这个方法。笔者遇到过太多这个问题，目前生产环境已全线禁用，如果 php 工程师需要获取远程 web 的内容，建议他们使用 curl.

php curl 如何使用请查看我之前的文章《PHP 使用 curl 替代 file_get_contents》，以及 php 下 curl 与 file_get_contents 性能对比.

(16)nginx tcp 代理

nginx　tcp 代理功能由 nginx_tcp_proxy_module 模块提供，同时监测后端主机状态。该模块包括的模块有：ngx_tcp_module，ngx_tcp_core_module，ngx_tcp_upstream_module，ngx_tcp_proxy_module，ngx_tcp_upstream_ip_hash_module。

1. 安装

```
# wget http://nginx.org/download/nginx-1.4.4.tar.gz
# tar zxvf nginx-1.4.4.tar.gz
# cd nginx-1.4.4
# ./configure --add-module=/path/to/nginx_tcp_proxy_module
# make
# make install
```

2. 配置

```
http {
    listen 80;
    location /status {
        check_status;
    }
}
tcp {
    upstream cluster_www_ttlsa_com {

# simple round-robin
        server 127.0.0.1:1234;
        check interval=3000 rise=2 fall=5 timeout=1000;
#check interval=3000 rise=2 fall=5 timeout=1000 type=ssl_hello;
#check interval=3000 rise=2 fall=5 timeout=1000 type=http;
#check_http_send "GET / HTTP/1.0\r\n\r\n";
#check_http_expect_alive http_2xx http_3xx;
    }
    server {
        listen 8888;
        proxy_pass cluster_www_ttlsa_com;
    }
}
```

这会出现一个问题，就是 tcp 连接会掉线。原因在于当服务端关闭连接的时候，客户端不可能立刻发觉连接已经被关闭，需要等到当 Nginx 在执行 check 规则时认为服务端链接关闭，此时 nginx 会关闭与客户端的连接。

3. 保持连接配置

```
http {
    listen 80;
    location /status {
        check_status;
    }
}
tcp {
    timeout 1d;
    proxy_read_timeout 10d;
```

```
    proxy_send_timeout 10d;
    proxy_connect_timeout 30;
    upstream cluster_www_ttlsa_com {

# simple round-robin
        server 127.0.0.1:1234;
        check interval=3000 rise=2 fall=5 timeout=1000;
#check interval=3000 rise=2 fall=5 timeout=1000 type=ssl_hello;
#check interval=3000 rise=2 fall=5 timeout=1000 type=http;
#check_http_send "GET / HTTP/1.0\r\n\r\n";
#check_http_expect_alive http_2xx http_3xx;
    }
    server {
        listen 8888;
        proxy_pass cluster_www_ttlsa_com;
        so_keepalive on;
        tcp_nodelay on;
    }
}
```

nginx_tcp_proxy_module 模块指令具体参见: http://yaoweibin.github.io/nginx_tcp_proxy_module/README.html

(17)nginx 正向代理

我们平时用的最多的最常见的是反向代理。反向代理想必都会配置的，有不会的可以到本博客里面搜索下，有相关文档。 那么 nginx 的正向代理是如何配置的呢？

```
server {
    listen 8090;
    location / {
            resolver 218.85.157.99 218.85.152.99;
            resolver_timeout 30s;
            proxy_pass http://$host$request_uri;
    }
    access_log  /data/httplogs/proxy-$host-aceess.log;
}
```

就这么简单哈。

测试：

http://www.ttlsa.com:8090

resolver 指令
语法: resolver address … [valid=time];
默认值: —
配置段: http, server, location
配置 DNS 服务器 IP 地址。可以指定多个，以轮询方式请求。
nginx 会缓存解析的结果。默认情况下，缓存时间是名字解析响应中的 TTL 字段的值，可以通过 valid 参数更改。

resolver_timeout 指令
语法: resolver_timeout time;
默认值: resolver_timeout 30s;
配置段: http, server, location
解析超时时间。

(18)搭建 nginx 反向代理用做内网域名转发

情景

由于公司内网有多台服务器的 http 服务要映射到公司外网静态 IP，如果用路由的端口映射来做，就只能一台内网服务器的 80 端口映射到外网 80 端口，其他服务器的 80 端口只能映射到外网的非 80 端口。非 80 端口的映射在访问的时候要域名加上端口，比较麻烦。并且公司入口路由最多只能做 20 个端口映射。肯定以后不够用。
然后 k 兄就提议可以在内网搭建个 nginx 反向代理服务器，将 nginx 反向代理服务器的 80 映射到外网 IP 的 80，这样指向到公司外网 IP 的域名的 HTTP 请求就会发送到 nginx 反向代理服务器，利用 nginx 反向代理将不同域名的请求转发给内网不同机器的端口，就起到了“根据域名自动转发到相应服务器的特定端口”的效果，而路由器的端口映射做到的只是“根据不同端口自动转发到相应服务器的特定端口”，真是喜大普奔啊。

涉及的知识：nginx 编译安装，nginx 反向代理基本配置，路由端口映射知识，还有网络域名等常识。

本次实验目标是做到：在浏览器中输入 xxx123.tk 能访问到内网机器 192.168.10.38 的 3000 端口，输入 xxx456.tk 能访问到内网机器 192.168.10.40 的 80 端口。

配置步骤

服务器 ubuntu 12.04

###更新仓库

```
apt-get update -y
apt-get install wget -y
#下载 nginx 和相关软件包
```

pcre 是为了编译 rewrite 模块，zlib 是为了支持 gzip 功能。额，这里 nginx 版本有点旧，因为我还要做升级 nginx 的实验用。大家可以装新版本。

```
cd /usr/local/src
wget ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/pcre-8.33.tar.gz
wget http://zlib.net/zlib-1.2.8.tar.gz
wget http://nginx.org/download/nginx-1.4.2.tar.gz
tar xf pcre-8.33.tar.gz
tar xf zlib-1.2.8.tar.gz
#安装编译环境
apt-get  install  build-essential libtool -y
#创建 nginx 用户
```

所谓的 unprivileged user

```
useradd  -s /bin/false  -r -M -d /nonexistent www
#开始编译安装
/configure --with-pcre=/usr/local/src/pcre-8.33 --with-zlib=/usr/local/src/zlib-1.2.8 --user=www --
group=www  --with-http_stub_status_module --with-http_ssl_module --with-http_realip_module
 make
 make install
```

#给文件夹授权
chown -R www:www /usr/local/nginx

#修改配置文件
vim nginx.conf

```
user www www;
worker_processes 1;
error_log logs/error.log;
pid logs/nginx.pid;
worker_rlimit_nofile 65535;
events {
    use epoll;
    worker_connections 65535;
}
http {
    include mime.types;
    default_type application/octet-stream;
    include /usr/local/nginx/conf/reverse-proxy.conf;
    sendfile on;
    keepalive_timeout 65;
    gzip on;
    client_max_body_size 50m;
#缓冲区代理缓冲用户端请求的最大字节数,可以理解为保存到本地再传给用户
    client_body_buffer_size 256k;
    client_header_timeout 3m;
    client_body_timeout 3m;
    send_timeout 3m;
    proxy_connect_timeout 300s;
#nginx 跟后端服务器连接超时时间(代理连接超时)
    proxy_read_timeout 300s;
#连接成功后，后端服务器响应时间(代理接收超时)
    proxy_send_timeout 300s;
    proxy_buffer_size 64k;
#设置代理服务器（nginx）保存用户头信息的缓冲区大小
    proxy_buffers 4 32k;
#proxy_buffers 缓冲区，网页平均在 32k 以下的话，这样设置
    proxy_busy_buffers_size 64k;
#高负荷下缓冲大小（proxy_buffers*2）
    proxy_temp_file_write_size 64k;
#设定缓存文件夹大小，大于这个值，将从 upstream 服务器传递请求，而不缓冲到磁盘
    proxy_ignore_client_abort on;
#不允许代理端主动关闭连接
    server {
        listen 80;
        server_name localhost;
        location / {
            root html;
            index index.html index.htm;
```

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

编辑反向代理服务器配置文件：

```
vim /usr/local/nginx/conf/reverse-proxy.conf

server
{
    listen 80;
    server_name xxx123.tk;
    location / {
        proxy_redirect off;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_pass http://192.168.10.38:3000;
    }
    access_log logs/xxx123.tk_access.log;
}

server
{
    listen 80;
    server_name xxx456.tk;
    location / {
        proxy_redirect off;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_pass http://192.168.10.40:80;
    }
    access_log logs/xxx456.tk_access.log;
}
```

然后重新加载 nginx 配置文件，使之修改生效，再把 xxx123.tk 域名指向公司静态 IP，这样就成功的做到了在浏览器中输入 xxx123.tk 的时候访问的内网服务器 192.168.10.38 的 3000 端口,输入 xxx456.tk 访问 192.168.10.40 的 80 端口的作用。

如果想对后端机器做负载均衡，像下面这配置就可以把对 nagios.xxx123.tk 的请求分发给内网的 131 和 132 这两台机器做负载均衡了。

```
upstream monitor_server {
    server 192.168.0.131:80;
        server 192.168.0.132:80;
}
server
{
    listen 80;
```

```
    server_name nagios.xxx123.tk;
    location / {
        proxy_redirect off;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_pass http://monitor_server;
    }
    access_log logs/nagios.xxx123.tk_access.log;
}
```

额，关于负载均衡和缓存就不多说了，这里只是要起到一个简单的“域名转发”功能。
另外，由于 http 请求最后都是由反向代理服务器传递给后段的机器，所以后端的机器原来的访问日志记录的访问 IP 都是反向代理服务器的 IP。
要想能记录真实 IP，需要修改后端机器的日志格式，这里假设后端也是一台 nginx：
在后端配置文件里面加入这一段即可：

```
log_format access '$HTTP_X_REAL_IP - $remote_user [$time_local] "$request" '
'$status $body_bytes_sent "$http_referer" '
'"$http_user_agent" $HTTP_X_Forwarded_For';

access_log logs/access.log access;
```

再看看原来日志的格式长什么样：

```
#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;
```

看出区别了吧

遇到的问题

- 之前没配置下面这段，访问时候偶尔会出现 504 gateway timeout，由于偶尔出现，所以不太好排查

```
proxy_connect_timeout 300s;
proxy_read_timeout 300s;
proxy_send_timeout 300s;
proxy_buffer_size 64k;
proxy_buffers 4 32k;
proxy_busy_buffers_size 64k;
proxy_temp_file_write_size 64k;
proxy_ignore_client_abort on;
```

报错日志：

```
...upstream timed out (110: Connection timed out) while reading response header from upstream, client: ...(后面的省略)
```

从日志看来是连接超时了，网上一通乱查之后估计可能是后端服务器响应超时了，本着大胆假设，小心求证的原则，既然假设了错误原因就要做实验重现错误：那就调整代理超时参数，反过来把代理超时阀值设小（比如 1ms）看会不会次次出现 504。后来发现把 proxy_read_timeout 这个参数设置成 1ms 的时候，每次访问都出现 504。于是把这个参数调大，加入上面那段配置，解决问题了。

(19)nginx+keepalived+proxy_cache 配置高可用 nginx 群集和高速缓存

环境：
CentOS release 5.8 192.168.10.108 cat
CentOS release 5.5 912.168.200.208
主调度器：192.168.10.108 192.169.10.251
备调度器：192.168.200.208 192.168.200.148
real ip :
192.169.10.251
192.168.200.148
vip : 192.168.10.104
一、在主备服务器上部署 nginx

1、下载
wget http://nginx.org/download/nginx-1.0.11.tar.gz
wget http://labs.frickle.com/files/ngx_cache_purge-1.4.tar.gz
2、安装

```
yum -y install zlib-devel pcre-devel openssl-devel   # 安装依赖
tar -xvf ngx_cache_purge-1.4.tar.gz
tar -xvf nginx-1.0.11.tar.gz
cd nginx-1.0.11/
./configure - prefix=/usr/local/nginx - add-module=../ngx_cache_purge-1.4 - with-http_stub_status_module - with-http_ssl_module - with-http_flv_module - with-http_gzip_static_module
Make && make install

vi /usr/local/nginx/conf/nginx.conf
user nobody;
worker_processes 8;
#error_log logs/error.log error;
error_log /data/logs/error.log crit;
#error_log logs/error.log notice;
#error_log logs/error.log info;
#pid logs/nginx.pid;
events {
worker_connections 1024;
}
http {
include mime.types;
default_type application/octet-stream;
charset utf-8;
server_names_hash_bucket_size 128;
client_header_buffer_size 32k;
large_client_header_buffers 4 32k;
client_max_body_size 300m;
tcp_nopush on;
tcp_nodelay on;
client_body_buffer_size 512k;
proxy_connect_timeout 5;
proxy_read_timeout 60;
```

```
proxy_send_timeout 5;
proxy_buffer_size 16k;
proxy_buffers 4 64k;
proxy_busy_buffers_size 128k;
proxy_temp_file_write_size 128k;
#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 on;
#keepalive_timeout 65;
gzip on;
gzip_min_length 1k;
gzip_buffers 4 16k;
gzip_http_version 1.1;
gzip_comp_level 2;
gzip_types text/plain application/x-javascript text/css application/xml;
gzip_vary on;
proxy_temp_path /data/proxy_temp_dir;
proxy_cache_path /data/proxy_cache_dir levels=1:2 keys_zone=cache_one:50m inactive=1m max_size=2g;
upstream real_server_pool{
server 192.168.200.148:80 weight=1 max_fails=2 fail_timeout=30s;
server 192.168.10.251:80 weight=1 max_fails=2 fail_timeout=30s;
}
#tcp_nopush on;
#keepalive_timeout 0;
keepalive_timeout 65;
#gzip on;
server {
listen 80;
server_name localhost;
#charset koi8-r;
#access_log logs/host.access.log main;
location / {
root html;
index index.html index.htm;
proxy_next_upstream http_502 http_504 error timeout invalid_header;
proxy_cache cache_one;
proxy_cache_valid 200 304 12h;
proxy_cache_key $host$uri$is_args$args;
proxy_set_header Host $host;
proxy_set_header X-Forwarded-For $remote_addr;
proxy_pass http://real_server_pool;
expires 1d;

}
#error_page 404 /404.html;
# redirect server error pages to the static page /50x.html
#
```

```
error_page 500 502 503 504 /50x.html;
location = /50x.html {
root html;
}
# proxy the  PHP  scripts to Apache listening on 127.0.0.1:80
#
#location ~ \.php$ {
# proxy_pass http://127.0.0.1;
#}
location ~ .*\.(php|jsp|cgi)?$
{
proxy_set_header Host $host;
proxy_set_header X-Forwarded-For $remote_addr;
proxy_pass http://real_server_pool;
}
log_format access  '$remote_addr  -  $remote_user [$time_local]  "$request"'
 '$status $body_bytes_sent  "$http_referer"  '
 '" $http_user_agent"  $http_x_forwarded_for';
access_log /data/logs/access.log access;
}

# pass the PHP scripts to FastCGI server listening on 127.0.0.1:9000
#
#location ~ \.php$ {
# root html;
# fastcgi_pass 127.0.0.1:9000;
# fastcgi_index index.php;
# fastcgi_param SCRIPT_FILENAME /scripts$fastcgi_script_name;
# include fastcgi_params;
#}
# deny access to .htaccess files, if Apache's document root
# concurs with nginx's one
#
#location ~ /\.ht {
# deny all;
#}
}
# another virtual host using mix of IP-, name-, and port-based configuration
#
#server {
# listen 8000;
# listen somename:8080;
# server_name somename alias another.alias;
# location / {
# root html;
# index index.html index.htm;
# }
#}
# HTTPS server
```

```
#
#server {
# listen 443;
# server_name localhost;
# ssl on;
# ssl_certificate cert.pem;
# ssl_certificate_key cert.key;
# ssl_session_timeout 5m;
# ssl_protocols SSLv2 SSLv3 TLSv1;
# ssl_ciphers HIGH:!aNULL:!MD5;
# ssl_prefer_server_ciphers on;
# location / {
# root html;
# index index.html index.htm;
# }
#}
```

备用调度器的 nginx 配置文件和主调度器的配置文件一样。
启动 nginx

```
/usr/local/nginx/sbin/nginx
```

二、安装 keepalived（在 nginx 的 mater 和 backup 都安装）

1、 下载

```
wget http://www.keepalived.org/software/keepalived-1.1.19.tar.gz
```

2、 安装

```
tar zxvf keepalived-1.1.19.tar.gz
cd keepalived-1.1.19
./configure -prefix=/usr/local/keepalived
make
make install
cp /usr/local/keepalived/sbin/keepalived /usr/sbin/
cp /usr/local/keepalived/etc/sysconfig/keepalived /etc/sysconfig/
cp /usr/local/keepalived/etc/rc.d/init.d/keepalived /etc/init.d/
mkdir /etc/keepalived

vi /etc/keepalived/keepalived.conf
vrrp_instance VI_INET1 {
state MASTER
interface eth0
virtual_router_id 53
priority 100
advert_int 1
authentication {
auth_type pass
auth_pass 1111
}
virtual_ipaddress {
192.168.10.104/24
```

```
}
}
virtual_server 192.168.10.104 80 {
delay_loop 6
lb_algo rr
lb_kind NAT
nat_mask 255.255.255.0
persistence_timeout 50
protocol TCP
real_server 192.168.10.251 80 {
weight 3
TCP_CHECK {
connect_timeout 10
nb_get_retry 3
delay_before_retry 3
connect_port 80
}
}
real_server 192.168.200.148 80 {
weight 3
TCP_CHECK {
connect_timeout 10
nb_get_retry 3
delay_before_retry 3
connect_port 80
}
}
}
```

4、配置备用调度器的 keepalived,只需要将 state MASTER 改为 state BACKUP,降低 priority 100 的值:

state MASTER —> state BACKUP

priority 100 —>　priority 99 （此值必须低于主的）

主备启动

/etc/init.d/keepalived start

三、测试

建立虚拟主机（自己测试啊　　0(∩_∩)0~）

(20)Nginx 战斗准备 —— 优化指南

大多数的 Nginx 安装指南告诉你如下基础知识——通过 apt-get 安装，修改这里或那里的几行配置，好了，你已经有了一个 Web 服务器了！而且，在大多数情况下，一个常规安装的 nginx 对你的网站来说已经能很好地工作了。然而，如果你真的想挤压出 nginx 的性能，你必须更深入一些。在本指南中，我将解释 Nginx 的那些设置可以微调，以优化处理大量客户端时的性能。需要注意一点，这不是一个全面的微调指南。这是一个简单的预览——那些可以通过微调来提高性能设置的概述。你的情况可能不同。

基本的 (优化过的)配置

我们将修改的唯一文件是 **nginx.conf**，其中包含 Nginx 不同模块的所有设置。你应该能够在服务器的**/etc/nginx** 目录中找到 nginx.conf。首先，我们将谈论一些全局设置，然后按文件中的模块挨个来，谈一下哪些设置能够让你在大量客户端访问时拥有良好的性能，为什么它们会提高性能。本文的结尾有一个完整的配置文件。

高层的配置

nginx.conf 文件中，Nginx 中有少数的几个高级配置在模块部分之上。

```
user www-data;
pid /var/run/nginx.pid;
worker_processes auto;
worker_rlimit_nofile 100000;
```

user 和 **pid** 应该按默认设置 - 我们不会更改这些内容，因为更改与否没有什么不同。

worker_processes 定义了 nginx 对外提供 web 服务时的 worder 进程数。最优值取决于许多因素，包括（但不限于）CPU 核的数量、存储数据的硬盘数量及负载模式。不能确定的时候，将其设置为可用的 CPU 内核数将是一个好的开始（设置为“auto”将尝试自动检测它）。

worker_rlimit_nofile 更改 worker 进程的最大打开文件数限制。如果没设置的话，这个值为操作系统的限制。设置后你的操作系统和 Nginx 可以处理比“ulimit -a”更多的文件，所以把这个值设高，这样 nginx 就不会有“too many open files”问题了。

Events 模块

events 模块中包含 nginx 中所有处理连接的设置。

```
events {
worker_connections 2048;
multi_accept on;
use epoll;
}
```

worker_connections 设置可由一个 worker 进程同时打开的最大连接数。如果设置了上面提到的 worker_rlimit_nofile，我们可以将这个值设得很高。

记住，最大客户数也由系统的可用 socket 连接数限制（~ 64K），所以设置不切实际的高没什么好处。

multi_accept 告诉 nginx 收到一个新连接通知后接受尽可能多的连接。

use 设置用于复用客户端线程的轮询方法。如果你使用 Linux 2.6+，你应该使用 epoll。如果你使用*BSD，你应该使用 kqueue。想知道更多有关事件轮询？看下维基百科吧（注意，想了解一切的话可能需要 neckbeard 和操作系统的课程基础）

（值得注意的是如果你不知道 Nginx 该使用哪种轮询方法的话，它会选择一个最适合你操作系统的）

HTTP 模块

HTTP 模块控制着 nginx http 处理的所有核心特性。因为这里只有很少的配置，所以我们只节选配置的一小部分。所有这些设置都应该在 http 模块中，甚至你不会特别的注意到这段设置。

```
http {
  server_tokens off;
  sendfile on;
   tcp_nopush on;
  tcp_nodelay on;
  ...
}
```

server_tokens　并不会让 nginx 执行的速度更快，但它可以关闭在错误页面中的 nginx 版本数字，这样对于安全性是有好处的。**sendfile** 可以让 sendfile()发挥作用。sendfile()可以在磁盘和 TCP socket 之间互相拷贝数据(或任意两个文件描述符)。Pre-sendfile 是传送数据之前在用户空间申请数据缓冲区。之后用 read()将数据从文件拷贝到这个缓冲区，write()将缓冲区数据写入网络。sendfile()是立即将数据从磁盘读到 OS 缓存。因为这种拷贝是在内核完成的，sendfile()要比组合 read()和 write()以及打开关闭丢弃缓冲更加有效(更多有关于 sendfile)

tcp_nopush　告诉 nginx 在一个数据包里发送所有头文件，而不一个接一个的发送

tcp_nodelay　告诉 nginx 不要缓存数据，而是一段一段的发送 - 当需要及时发送数据时，就应该给应用设置这个属性，这样发送一小块数据信息时就不能立即得到返回值。

```
  access_log off;
  error_log /var/log/nginx/error.log crit;
```

access_log 设置 nginx 是否将存储访问日志。关闭这个选项可以让读取磁盘 IO 操作更快
(aka, YOLO)**error_log**　告诉 nginx 只能记录严重的错误

```
  keepalive_timeout 10;
  client_header_timeout 10;
  client_body_timeout 10;
  reset_timedout_connection on;
  send_timeout 10;
```

keepalive_timeout　给客户端分配 keep-alive 链接超时时间。服务器将在这个超时时间过后关闭链接。我们将它设置低些可以让 ngnix 持续工作的时间更长。client_header_timeout　和 **client_body_timeout**　设置请求头和请求体(各自)的超时时间。我们也可以把这个设置低些。

reset_timeout_connection 告诉 nginx 关闭不响应的客户端连接。这将会释放那个客户端所占有的内存空间。

send_timeout 指定客户端的响应超时时间。这个设置不会用于整个转发器，而是在两次客户端读取操作之间。如果在这段时间内，客户端没有读取任何数据，nginx 就会关闭连接。

```
  limit_conn_zone $binary_remote_addr zone=addr:5m;
  limit_conn addr 100;
```

limit_conn_zone 设置用于保存各种 key（比如当前连接数）的共享内存的参数。5m 就是 5 兆字节，这个值应该被设置的足够大以存储（32K*5）32byte 状态或者（16K*5）64byte 状态。limit_conn 为给定的 key 设置最大连接数。这里 key 是 addr，我们设置的值是 100，也就是说我们允许每一个 IP 地址最多同时打开有 100 个连接。

```
  include /etc/nginx/mime.types;
  default_type text/html;
  charset UTF-8;
```

include 只是一个在当前文件中包含另一个文件内容的指令。这里我们使用它来加载稍后会用到的一系列的 MIME 类型。default_type 设置文件使用的默认的 MIME-type。

charset 设置我们的头文件中的默认的字符集

以下两点对于性能的提升在伟大的 WebMasters StackExchange 中有解释。

```
gzip on;

  gzip_disable "msie6";
  # gzip_static on;
  gzip_proxied any;
  gzip_min_length 1000;
  gzip_comp_level 4;
  gzip_types text/plain text/css application/json application/x-javascript text/xml application/xml
  application/xml+rss text/javascript;
```

gzip 是告诉 nginx 采用 gzip 压缩的形式发送数据。这将会减少我们发送的数据量。

gzip_disable 为指定的客户端禁用 gzip 功能。我们设置成 IE6 或者更低版本以使我们的方案能够广泛兼容。

gzip_static 告诉 nginx 在压缩资源之前，先查找是否有预先 gzip 处理过的资源。这要求你预先压缩你的文件（在这个例子中被注释掉了），从而允许你使用最高压缩比，这样 nginx 就不用再压缩这些文件了（想要更详尽的 gzip_static 的信息，请点击这里）。

gzip_proxied 允许或者禁止压缩基于请求和响应的响应流。我们设置为 any，意味着将会压缩所有的请求。

gzip_min_length 设置对数据启用压缩的最少字节数。如果一个请求小于 1000 字节，我们最好不要压缩它，因为压缩这些小的数据会降低处理此请求的所有进程的速度。

gzip_comp_level 设置数据的压缩等级。这个等级可以是 1-9 之间的任意数值，9 是最慢但是压缩比最大的。我们设置为 4，这是一个比较折中的设置。

gzip_type 设置需要压缩的数据格式。上面例子中已经有一些了，你也可以再添加更多的格式。

```
  # cache informations about file descriptors, frequently accessed files
  # can boost performance, but you need to test those values
  open_file_cache max=100000 inactive=20s;
  open_file_cache_valid 30s;
  open_file_cache_min_uses 2;
  open_file_cache_errors on;
  ##
  # Virtual Host Configs
  # aka our settings for specific servers
  ##
  include /etc/nginx/conf.d/*.conf;
  include /etc/nginx/sites-enabled/*;
```

open_file_cache 打开缓存的同时也指定了缓存最大数目，以及缓存的时间。我们可以设置一个相对高的最大时间，这样我们可以在它们不活动超过 20 秒后清除掉。

open_file_cache_valid　在 open_file_cache 中指定检测正确信息的间隔时间。

open_file_cache_min_uses　定义了 open_file_cache 中指令参数不活动时间期间里最小的文件数。

open_file_cache_errors 指定了当搜索一个文件时是否缓存错误信息，也包括再次给配置中添加文件。我们也包括了服务器模块，这些是在不同文件中定义的。如果你的服务器模块不在这些位置，你就得修改这一行来指定正确的位置。

一个完整的配置

```
user www-data;
pid /var/run/nginx.pid;
worker_processes auto;
worker_rlimit_nofile 100000;
events {
 worker_connections 2048;
 multi_accept on;
 use epoll;
}
http {
  server_tokens off;
 sendfile on;
tcp_nopush on;
 tcp_nodelay on;
 access_log off;
 error_log /var/log/nginx/error.log crit;
 keepalive_timeout 10;
 client_header_timeout 10;
 client_body_timeout 10;
reset_timedout_connection on;
 send_timeout 10;
 limit_conn_zone $binary_remote_addr zone=addr:5m;
 limit_conn addr 100;
 include /etc/nginx/mime.types;
 default_type text/html;
charset UTF-8;
gzip on;
 gzip_disable "msie6";
gzip_proxied any;
gzip_min_length 1000;
gzip_comp_level 6;
 gzip_types text/plain text/css application/json application/x-javascript text/xml application/xml
application/xml+rss text/javascript;
 open_file_cache max=100000 inactive=20s;
open_file_cache_valid 30s;
open_file_cache_min_uses 2;
open_file_cache_errors on;
 include /etc/nginx/conf.d/*.conf;
 include /etc/nginx/sites-enabled/*;
}
```

编辑完配置后，确认重启 nginx 使设置生效。

后记

就这样！你的 Web 服务器现在已经就绪，之前困扰你的众多访问者的问题来吧。这并不是加速网站的唯一途径，很快我会写更多介绍其他加速网站方法的文章的。

(21)确保 nginx 安全的 10 个技巧

Nginx 是当今最流行的 Web 服务器之一。它为世界上 7%的 web 流量提供服务而且正在以惊人的速度增长。它是个让人惊奇的服务器，我愿意部署它。
下面是一个常见安全陷阱和解决方案的列表，它可以辅助来确保你的 Nginx 部署是安全的。

1. 在配置文件中小心使用”if”

它是重写模块的一部分，不应该在任何地方使用。
“if”声明是重写模块评估指令强制性的部分。换个说法，Nginx 的配置一般来说是声明式的。在有些情况下，由于用户的需求，他们试图在一些非重写指令内使用“if”，这导致我们现在遇到的情况。大多数情况下都能正常工作，但…看上面提到的。

看起来唯一正确的解决方案是在非重写的指令内完全禁用“if”。这将更改现有的许多配置，所以还没有完成。
IfIsEvil：http://wiki.nginx.org/IfIsEvil

2. 将每个~ .php$请求转递给 PHP

我们上周发布了这个流行指令的潜在安全漏洞介绍。即使文件名为 hello.php.jpeg 它也会匹配~ .php$这个正则而执行文件。
现在有两个解决上述问题的好方法。我觉得确保你不轻易执行任意代码的混合方法很有必要。
2.1 如果没找到文件时使用 try_files 和 only(在所有的动态执行情况下都应该注意）将它转递给运行 PHP 的 FCGI 进程。
2.2 确认 php.ini 文件中 cgi.fix_pathinfo 设置为 0 （cgi.fix_pathinfo=0）。这样确保 PHP 检查文件全名(当它在文件结尾没有发现.php 它将忽略)
2.3 修复正则表达式匹配不正确文件的问题。现在正则表达式认为任何文件都包含”.php”。在站点后加“if”确保只有正确的文件才能运行。将/location ~ .php$和 location ~ ..*/.*.php$都设置为 return 403;

3. 禁用 autoindex 模块

这个可能在你使用的 Nginx 版本中已经更改了，如果没有的话只需在配置文件的 location 块中增加 autoindex off;声明即可。

4. 禁用服务器上的 ssi（服务器端引用）

这个可以通过在 location 块中添加 ssi off; 。

5. 关闭服务器标记

如果开启的话（默认情况下）所有的错误页面都会显示服务器的版本和信息。将 server_tokens off;声明添加到 Nginx 配置文件来解决这个问题。

6. 在配置文件中设置自定义缓存以限制缓冲区溢出攻击的可能性

```
client_body_buffer_size 1K;
client_header_buffer_size 1k;
client_max_body_size 1k;
large_client_header_buffers 2 1k;
```

7. 将 timeout 设低来防止 DOS 攻击

所有这些声明都可以放到主配置文件中。

```
client_body_timeout 10;
client_header_timeout 10;
keepalive_timeout 5 5;
send_timeout 10;
```

8. 限制用户连接数来预防 DOS 攻击

```
limit_zone slimits $binary_remote_addr 5m;
limit_conn slimits 5;
```

9. 试着避免使用 HTTP 认证

HTTP 认证默认使用 crypt，它的哈希并不安全。如果你要用的话就用 MD5（这也不是个好选择但负载方面比 crypt 好）。

10. 保持与最新的 Nginx 安全更新

转自：http://www.levigross.com/post/4488812448/10-tips-for-securing-nginx

个人觉得在防止 DDOS 攻击这方面，上面提到的第七第八没太大用处，特别是第八点，很扰乱用户体验度的。

NGINX 变量详解

nginx 变量使用方法详解(1)

Nginx 的配置文件使用的就是一门微型的编程语言，许多真实世界里的 Nginx 配置文件其实就是一个一个的小程序。当然，是不是“图灵完全的”暂且不论，至少据我观察，它在设计上受 Perl 和 Bourne Shell 这两种语言的影响很大。在这一点上，相比 Apache 和 Lighttpd 等其他 Web 服务器的配置记法，不能不说算是 Nginx 的一大特色了。既然是编程语言，一般也就少不了“变量”这种东西（当然，Haskell 这样奇怪的函数式语言除外了）。

熟悉 Perl、Bourne Shell、C/C++ 等命令式编程语言的朋友肯定知道，变量说白了就是存放“值”的容器。而所谓“值”，在许多编程语言里，既可以是 3.14 这样的数值，也可以是 hello world 这样的字符串，甚至可以是像数组、哈希表这样的复杂数据结构。然而，在 Nginx 配置中，变量只能存放一种类型的值，因为也只存在一种类型的值，那就是字符串。

比如我们的 nginx.conf 文件中有下面这一行配置：

```
set $a "hello world";
```

我们使用了标准 ngx_rewrite 模块的 set 配置指令对变量 $a 进行了赋值操作。特别地，我们把字符串 hello world 赋给了它。

我们看到，Nginx 变量名前面有一个 $ 符号，这是记法上的要求。所有的 Nginx 变量在 Nginx 配置文件中引用时都须带上 $ 前缀。这种表示方法和 Perl、PHP 这些语言是相似的。

虽然 $ 这样的变量前缀修饰会让正统的 Java 和 C# 程序员不舒服，但这种表示方法的好处也是显而易见的，那就是可以直接把变量嵌入到字符串常量中以构造出新的字符串：

```
set $a hello;
set $b "$a, $a";
```

这里我们通过已有的 Nginx 变量 $a 的值，来构造变量 $b 的值，于是这两条指令顺序执行完之后，$a 的值是 hello，而 $b 的值则是 hello, hello. 这种技术在 Perl 世界里被称为“变量插值”（variable interpolation），它让专门的字符串拼接运算符变得不再那么必要。我们在这里也不妨采用此术语。

我们来看一个比较完整的配置示例：

```
server {
listen 8080;
location /test {
set $foo hello;
echo "foo: $foo";
}
}
```

这个例子省略了 nginx.conf 配置文件中最外围的 http 配置块以及 events 配置块。使用 curl 这个 HTTP 客户端在命令行上请求这个 /test 接口，我们可以得到

```
$ curl 'http://localhost:8080/test'
foo: hello
```

这里我们使用第三方 ngx_echo 模块的 echo 配置指令将 $foo 变量的值作为当前请求的响应体输出。

我们看到，echo 配置指令的参数也支持“变量插值”。不过，需要说明的是，并非所有的配置指令都支持“变量插值”。事实上，指令参数是否允许“变量插值”，取决于该指令的实现模块。

如果我们想通过 echo 指令直接输出含有“美元符”（$）的字符串，那么有没有办法把特殊的 $ 字符给转义掉呢？答案是否定的（至少到目前最新的 Nginx 稳定版 1.0.10）。不过幸运的是，我们可以绕过这个限制，比如通过不支持“变量插值”的模块配置指令专门构造出取值为 $ 的 Nginx 变量，然后再在 echo 中使用这个变量。看下面这个例子：

```
geo $dollar {
  default "$";
}
server {
  listen 8080;
  location /test {
      echo "This is a dollar sign: $dollar";
  }
}
```

测试结果如下：

```
$ curl 'http://localhost:8080/test'
This is a dollar sign: $
```

这里用到了标准模块 ngx_geo 提供的配置指令 geo 来为变量 $dollar 赋予字符串 “$”，这样我们在下面需要使用美元符的地方，就直接引用我们的 $dollar 变量就可以了。其实 ngx_geo 模块最常规的用法是根据客户端的 IP 地址对指定的 Nginx 变量进行赋值，这里只是借用它以便“无条件地”对我们的 $dollar 变量赋予“美元符”这个值。

在“变量插值”的上下文中，还有一种特殊情况，即当引用的变量名之后紧跟着变量名的构成字符时（比如后跟字母、数字以及下划线），我们就需要使用特别的记法来消除歧义，例如：

```
server {
listen 8080;
location /test {
set $first "hello ";
echo "${first}world";
}
}
```

这里，我们在 echo 配置指令的参数值中引用变量 $first 的时候，后面紧跟着 world 这个单词，所以如果直接写作 “$firstworld” 则 Nginx “变量插值”计算引擎会将之识别为引用了变量 $firstworld. 为了解决这个难题，Nginx 的字符串记法支持使用花括号在 $ 之后把变量名围起来，比如这里的 ${first}. 上面这个例子的输出是：

```
$ curl 'http://localhost:8080/test
hello world
```

set 指令（以及前面提到的 geo 指令）不仅有赋值的功能，它还有创建 Nginx 变量的副作用，即当作为赋值对象的变量尚不存在时，它会自动创建该变量。比如在上面这个例子中，如果 $a 这个变量尚未创建，则 set 指令会自动创建 $a 这个用户变量。如果我们不创建就直接使用它的值，则会报错。例如

```
server {
  listen 8080;
  location /bad {
      echo $foo;
  }
}
```

此时 Nginx 服务器会拒绝加载配置：

```
[emerg] unknown "foo" variable
```

是的，我们甚至都无法启动服务！

有趣的是，Nginx 变量的创建和赋值操作发生在全然不同的时间阶段。Nginx 变量的创建只能发生在 Nginx 配置加载的时候，或者说 Nginx 启动的时候；而赋值操作则只会发生在请求实际处理的时候。这意味着不创建而直接使用变量会导致启动失败，同时也意味着我们无法在请求处理时动态地创建新的 Nginx 变量。

Nginx 变量一旦创建，其变量名的可见范围就是整个 Nginx 配置，甚至可以跨越不同虚拟主机的 server 配置块。我们来看一个例子：

```
server {
  listen 8080;
  location /foo {
    echo "foo = [$foo]";
  }
  location /bar {
    set $foo 32;
    echo "foo = [$foo]";
  }
}
```

这里我们在 location /bar 中用 set 指令创建了变量 $foo，于是在整个配置文件中这个变量都是可见的，因此我们可以在 location /foo 中直接引用这个变量而不用担心 Nginx 会报错。

下面是在命令行上用 curl 工具访问这两个接口的结果：

```
$ curl 'http://localhost:8080/foo'
foo = []

$ curl 'http://localhost:8080/bar'
foo = [32]

$ curl 'http://localhost:8080/foo'
foo = []
```

从这个例子我们可以看到，set 指令因为是在 location /bar 中使用的，所以赋值操作只会在访问 /bar 的请求中执行。而请求 /foo 接口时，我们总是得到空的 $foo 值，因为用户变量未赋值就输出的话，得到的便是空字符串。

从这个例子我们可以窥见的另一个重要特性是，Nginx 变量名的可见范围虽然是整个配置，但每个请求都有所有变量的独立副本，或者说都有各变量用来存放值的容器的独立副本，彼此互不干扰。比如前面我们请求了 /bar 接口后，$foo 变量被赋予了值 32，但它丝毫不会影响后续对 /foo 接口的请求所对应的 $foo 值（它仍然是空的！），因为各个请求都有自己独立的 $foo 变量的副本。

对于 Nginx 新手来说，最常见的错误之一，就是将 Nginx 变量理解成某种在请求之间全局共享的东西，或者说“全局变量”。而事实上，Nginx 变量的生命期是不可能跨越请求边界的。

nginx 变量使用方法详解(2)

关于 Nginx 变量的另一个常见误区是认为变量容器的生命期，是与 location 配置块绑定的。其实不然。我们来看一个涉及“内部跳转”的例子：

```
server {
    listen 8080;
    location /foo {
        set $a hello;
```

```
        echo_exec /bar;
    }
    location /bar {
        echo "a = [$a]";
    }
}
```

这里我们在 location /foo 中，使用第三方模块 ngx_echo 提供的 echo_exec 配置指令，发起到 location /bar 的“内部跳转”。所谓“内部跳转”，就是在处理请求的过程中，于服务器内部，从一个 location 跳转到另一个 location 的过程。这不同于利用 HTTP 状态码 301 和 302 所进行的“外部跳转”，因为后者是由 HTTP 客户端配合进行跳转的，而且在客户端，用户可以通过浏览器地址栏这样的界面，看到请求的 URL 地址发生了变化。内部跳转和 BourneShell（或 Bash）中的 exec 命令很像，都是“有去无回”。另一个相近的例子是 C 语言中的 goto 语句。

既然是内部跳转，当前正在处理的请求就还是原来那个，只是当前的 location 发生了变化，所以还是原来的那一套 Nginx 变量的容器副本。对应到上例，如果我们请求的是 /foo 这个接口，那么整个工作流程是这样的：先在 location /foo 中通过 set 指令将 $a 变量的值赋为字符串 hello，然后通过 echo_exec 指令发起内部跳转，又进入到 location /bar 中，再输出 $a 变量的值。因为 $a 还是原来的 $a，所以我们可以期望得到 hello 这行输出。测试证实了这一点：

```
$ curl localhost:8080/foo
a = [hello]
```

但如果我们从客户端直接访问 /bar 接口，就会得到空的 $a 变量的值，因为它依赖于 location /foo 来对 $a 进行初始化。

从上面这个例子我们看到，一个请求在其处理过程中，即使经历多个不同的 location 配置块，它使用的还是同一套 Nginx 变量的副本。这里，我们也首次涉及到了“内部跳转”这个概念。值得一提的是，标准 ngx_rewrite 模块的 rewrite 配置指令其实也可以发起“内部跳转”，例如上面那个例子用 rewrite 配置指令可以改写成下面这样的形式：

```
server {
    listen 8080;
    location /foo {
        set $a hello;
        rewrite ^ /bar;
    }
    location /bar {
        echo "a = [$a]";
    }
}
```

其效果和使用 echo_exec 是完全相同的。后面我们还会专门介绍这个 rewrite 指令的更多用法，比如发起 301 和 302 这样的“外部跳转”。

从上面这个例子我们看到，Nginx 变量值容器的生命期是与当前正在处理的请求绑定的，而与 location 无关。

前面我们接触到的都是通过 set 指令隐式创建的 Nginx 变量。这些变量我们一般称为“用户自定义变量”，或者更简单一些，“用户变量”。既然有“用户自定义变量”，自然也就有由 Nginx 核心和各个 Nginx 模块提供的“预定义变量”，或者说“内建变量”（builtin variables）。

Nginx 内建变量最常见的用途就是获取关于请求或响应的各种信息。例如由 ngx_http_core 模块提供的内建变量 $uri，可以用来获取当前请求的 URI（经过解码，并且不含请求参数），而 $request_uri 则用来获取请求最原始的 URI （未经解码，并且包含请求参数）。请看下面这个例子：

```
location /test {
    echo "uri = $uri";
    echo "request_uri = $request_uri";
}
```

这里为了简单起见，连 server 配置块也省略了，和前面所有示例一样，我们监听的依然是 8080 端口。在这个例子里，我们把 $uri 和 $request_uri 的值输出到响应体中去。下面我们用不同的请求来测试一下这个 /test 接口：

```
$ curl 'http://localhost:8080/test'
uri = /test
request_uri = /test

$ curl 'http://localhost:8080/test?a=3&b=4'
uri = /test
request_uri = /test?a=3&b=4

$ curl 'http://localhost:8080/test/hello%20world?a=3&b=4'
uri = /test/hello world
request_uri = /test/hello%20world?a=3&b=4
```

另一个特别常用的内建变量其实并不是单独一个变量，而是有无限多变种的一群变量，即名字以 arg_ 开头的所有变量，我们估且称之为 $arg_XXX 变量群。一个例子是 $arg_name，这个变量的值是当前请求名为 name 的 URI 参数的值，而且还是未解码的原始形式的值。我们来看一个比较完整的示例：

```
location /test {
    echo "name: $arg_name";
    echo "class: $arg_class";
}
```

然后在命令行上使用各种参数组合去请求这个 /test 接口：

```
$ curl 'http://localhost:8080/test'
name:
class:
$ curl 'http://localhost:8080/test?name=Tom&class=3'
name: Tom
class: 3
$ curl 'http://localhost:8080/test?name=hello%20world&class=9'
name: hello%20world
class: 9
```

其实 $arg_name 不仅可以匹配 name 参数，也可以匹配 NAME 参数，抑或是 Name，等等：

```
$ curl 'http://localhost:8080/test?NAME=Marry'
name: Marry
class:
$ curl 'http://localhost:8080/test?Name=Jimmy'
name: Jimmy
class:
```

Nginx 会在匹配参数名之前，自动把原始请求中的参数名调整为全部小写的形式。

如果你想对 URI 参数值中的 %XX 这样的编码序列进行解码，可以使用第三方 ngx_set_misc 模块提供的 set_unescape_uri 配置指令：

```
location /test {
    set_unescape_uri $name $arg_name;
    set_unescape_uri $class $arg_class;
    echo "name: $name";
    echo "class: $class";
}
```

现在我们再看一下效果：

```
$ curl 'http://localhost:8080/test?name=hello%20world&class=9'
name: hello world
class: 9
```

空格果然被解码出来了！

从这个例子我们同时可以看到，这个 set_unescape_uri 指令也像 set 指令那样，拥有自动创建 Nginx 变量的功能。后面我们还会专门介绍到 ngx_set_misc 模块。

像 $arg_XXX 这种类型的变量拥有无穷无尽种可能的名字，所以它们并不对应任何存放值的容器。而且这种变量在 Nginx 核心中是经过特别处理的，第三方 Nginx 模块是不能提供这样充满魔法的内建变量的。

类似 $arg_XXX 的内建变量还有不少，比如用来取 cookie 值的 $cookie_XXX 变量群，用来取请求头的 $http_XXX 变量群，以及用来取响应头的 $sent_http_XXX 变量群。这里就不一一介绍了，感兴趣的读者可以参考 ngx_http_core 模块的官方文档。

需要指出的是，许多内建变量都是只读的，比如我们刚才介绍的 $uri 和 $request_uri. 对只读变量进行赋值是应当绝对避免的，因为会有意想不到的后果，比如：

```
$ curl 'http://localhost:8080/test?name=hello%20world&class=9'
name: hello world
class: 9
```

这个有问题的配置会让 Nginx 在启动的时候报出一条令人匪夷所思的错误：

```
[emerg] the duplicate "uri" variable in ...
```

如果你尝试改写另外一些只读的内建变量，比如 $arg_XXX 变量，在某些 Nginx 的版本中甚至可能导致进程崩溃。

nginx 变量使用方法详解(3)

也有一些内建变量是支持改写的，其中一个例子是 $args. 这个变量在读取时返回当前请求的 URL 参数串（即请求 URL 中问号后面的部分，如果有的话 ），而在赋值时可以直接修改参数串。我们来看一个例子：

```
location /test {
    set $orig_args $args;
    set $args "a=3&b=4";
    echo "original args: $orig_args";
    echo "args: $args";
}
```

这里我们把原始的 URL 参数串先保存在 $orig_args 变量中，然后通过改写 $args 变量来修改当前的 URL 参数串，最后我们用 echo 指令分别输出 $orig_args 和 $args 变量的值。接下来我们这样来测试这个 /test 接口：

```
$ curl 'http://localhost:8080/test'
original args:
args: a=3&b=4

$ curl 'http://localhost:8080/test?a=0&b=1&c=2'
original args: a=0&b=1&c=2
args: a=3&b=4
```

在第一次测试中，我们没有设置任何 URL 参数串，所以输出 $orig_args 变量的值时便得到空。而在第一次和第二次测试中，无论我们是否提供 URL 参数串，参数串都会在 location /test 中被强行改写成 a=3&b=4.

需要特别指出的是，这里的 $args 变量和 $arg_XXX 一样，也不再使用属于自己的存放值的容器。当我们读取 $args 时，Nginx 会执行一小段代码，从 Nginx 核心中专门存放当前 URL 参数串的位置去读取数据；而当我们

改写 $args 时，Nginx 会执行另一小段代码，对相同位置进行改写。Nginx 的其他部分在需要当前 URL 参数串的时候，都会从那个位置去读数据，所以我们对 $args 的修改会影响到所有部分的功能。我们来看一个例子：

```
location /test {
    set $orig_a $arg_a;
    set $args "a=5";
    echo "original a: $orig_a";
    echo "a: $arg_a";
}
```

这里我们先把内建变量 $arg_a 的值，即原始请求的 URL 参数 a 的值，保存在用户变量 $orig_a 中，然后通过对内建变量 $args 进行赋值，把当前请求的参数串改写为 a=5 ，最后再用 echo 指令分别输出 $orig_a 和 $arg_a 变量的值。因为对内建变量 $args 的修改会直接导致当前请求的 URL 参数串发生变化，因此内建变量 $arg_XXX 自然也会随之变化。测试的结果证实了这一点：

```
$ curl 'http://localhost:8080/test?a=3'
original a: 3
a: 5
```

我们看到，因为原始请求的 URL 参数串是 a=3，所以 $arg_a 最初的值为 3，但随后通过改写 $args 变量，将 URL 参数串又强行修改为 a=5，所以最终 $arg_a 的值又自动变为了 5.

我们再来看一个通过修改 $args 变量影响标准的 HTTP 代理模块 ngx_proxy 的例子：

```
server {
    listen 8080;

    location /test {
        set $args "foo=1&bar=2";
        proxy_pass http://127.0.0.1:8081/args;
    }
}

server {

    listen 8081;
    location /args {
        echo "args: $args";
    }
}
```

这里我们在 http 配置块中定义了两个虚拟主机。第一个虚拟主机监听 8080 端口，其 /test 接口自己通过改写 $args 变量，将当前请求的 URL 参数串无条件地修改为 foo=1&bar=2. 然后 /test 接口再通过 ngx_proxy 模块的 proxy_pass 指令配置了一个反向代理，指向本机的 8081 端口上的 HTTP 服务 /args. 默认情况下，ngx_proxy 模块在转发 HTTP 请求到远方 HTTP 服务的时候，会自动把当前请求的 URL 参数串也转发到远方。

而本机的 8081 端口上的 HTTP 服务正是由我们定义的第二个虚拟主机来提供的。我们在第二个虚拟主机的 location /args 中利用 echo 指令输出当前请求的 URL 参数串，以检查 /test 接口通过 ngx_proxy 模块实际转发过来的 URL 请求参数串。

我们来实际访问一下第一个虚拟主机的 /test 接口：

```
$ curl 'http://localhost:8080/test?blah=7'
args: foo=1&bar=2
```

我们看到，虽然请求自己提供了 URL 参数串 blah=7，但在 location /test 中，参数串被强行改写成了 foo=1&bar=2. 接着经由 proxy_pass 指令将我们被改写掉的参数串转发给了第二个虚拟主机上配置的 /args 接

口，然后再把 /args 接口的 URL 参数串输出。事实证明，我们对 $args 变量的赋值操作，也成功影响到了 ngx_proxy 模块的行为。

在读取变量时执行的这段特殊代码，在 Nginx 中被称为“取处理程序”（get handler）；而改写变量时执行的这段特殊代码，则被称为“存处理程序”（set handler）。不同的 Nginx 模块一般会为它们的变量准备不同的“存取处理程序”，从而让这些变量的行为充满魔法。

其实这种技巧在计算世界并不鲜见。比如在面向对象编程中，类的设计者一般不会把类的成员变量直接暴露给类的用户，而是另行提供两个方法（method），分别用于该成员变量的读操作和写操作，这两个方法常常被称为“存取器”（accessor）。下面是 C++ 语言中的一个例子：

```
#include <string>
using namespace std;
class Person {
public:
    const string get_name() {
        return m_name;
    }

    void set_name(const string name) {
        m_name = name;
    }

private:
    string m_name;
};
```

在这个名叫 Person 的 C++ 类中，我们提供了 get_name 和 set_name 这两个公共方法，以作为私有成员变量 m_name 的“存取器”。

这样设计的好处是显而易见的。类的设计者可以在“存取器”中执行任意代码，以实现所需的业务逻辑以及“副作用”，比如自动更新与当前成员变量存在依赖关系的其他成员变量，抑或是直接修改某个与当前对象相关联的数据库表中的对应字段。而对于后一种情况，也许“存取器”所对应的成员变量压根就不存在，或者即使存在，也顶多扮演着数据缓存的角色，以缓解被代理数据库的访问压力。

与面向对象编程中的“存取器”概念相对应，Nginx 变量也是支持绑定“存取处理程序”的。Nginx 模块在创建变量时，可以选择是否为变量分配存放值的容器，以及是否自己提供与读写操作相对应的“存取处理程序”。

不是所有的 Nginx 变量都拥有存放值的容器。拥有值容器的变量在 Nginx 核心中被称为“被索引的”（indexed）；反之，则被称为“未索引的”（non-indexed）。

我们前面在 （二） 中已经知道，像 $arg_XXX 这样具有无数变种的变量群，是“未索引的”。当读取这样的变量时，其实是它的“取处理程序”在起作用，即实时扫描当前请求的 URL 参数串，提取出变量名所指定的 URL 参数的值。很多新手都会对 $arg_XXX 的实现方式产生误解，以为 Nginx 会事先解析好当前请求的所有 URL 参数，并且把相关的 $arg_XXX 变量的值都事先设置好。然而事实并非如此，Nginx 根本不会事先就解析好 URL 参数串，而是在用户读取某个 $arg_XXX 变量时，调用其“取处理程序”，即时去扫描 URL 参数串。类似地，内建变量 $cookie_XXX 也是通过它的“取处理程序”，即时去扫描 Cookie 请求头中的相关定义的。

nginx 变量使用方法详解(4)

在设置了“取处理程序”的情况下，Nginx 变量也可以选择将其值容器用作缓存，这样在多次读取变量的时候，就只需要调用“取处理程序”计算一次。我们下面就来看一个这样的例子：

```
map $args $foo {
    default     0;
    debug       1;
}
server {
listen 8080;

    location /test {
        set $orig_foo $foo;
        set $args debug;
        echo "orginal foo: $orig_foo";
        echo "foo: $foo";
    }
}
```

这里首次用到了标准 ngx_map 模块的 map 配置指令，我们有必要在此介绍一下。map 在英文中除了“地图”之外，也有“映射”的意思。比方说，中学数学里讲的“函数”就是一种“映射”。而 Nginx 的这个 map 指令就可以用于定义两个 Nginx 变量之间的映射关系，或者说是函数关系。回到上面这个例子，我们用 map 指令定义了用户变量 $foo 与 $args 内建变量之间的映射关系。特别地，用数学上的函数记法 y = f(x) 来说，我们的 $args 就是“自变量” x，而 $foo 则是“因变量” y，即 $foo 的值是由 $args 的值来决定的，或者按照书写顺序可以说，我们将 $args 变量的值映射到了 $foo 变量上。

现在我们再来看 map 指令定义的映射规则：

```
map $args $foo {
    default     0;
    debug       1;
}
```

花括号中第一行的 default 是一个特殊的匹配条件，即当其他条件都不匹配的时候，这个条件才匹配。当这个默认条件匹配时，就把“因变量” $foo 映射到值 0. 而花括号中第二行的意思是说，如果“自变量” $args 精确匹配了 debug 这个字符串，则把“因变量” $foo 映射到值 1. 将这两行合起来，我们就得到如下完整的映射规则：当 $args 的值等于 debug 的时候，$foo 变量的值就是 1，否则 $foo 的值就为 0.

明白了 map 指令的含义，再来看 location /test. 在那里，我们先把当前 $foo 变量的值保存在另一个用户变量 $orig_foo 中，然后再强行把 $args 的值改写为 debug，最后我们再用 echo 指令分别输出 $orig_foo 和 $foo 的值。
从逻辑上看，似乎当我们强行改写 $args 的值为 debug 之后，根据先前的 map 映射规则，$foo 变量此时的值应当自动调整为字符串 1，而不论 $foo 原先的值是怎样的。然而测试结果并非如此：

```
$ curl 'http://localhost:8080/test'
original foo: 0
foo: 0
```

第一行输出指示 $orig_foo 的值为 0，这正是我们期望的：上面这个请求并没有提供 URL 参数串，于是 $args 最初的取值就是空，再根据我们先前定义的映射规则，$foo 变量在第一次被读取时的值就应当是 0（即匹配默认的那个 default 条件）。
而第二行输出显示，在强行改写 $args 变量的值为字符串 debug 之后，$foo 的条件仍然是 0 ，这显然不符合映射规则，因为当 $args 为 debug 时，$foo 的值应当是 1. 这究竟是为什么呢？

其实原因很简单，那就是 $foo 变量在第一次读取时，根据映射规则计算出的值被缓存住了。刚才我们说过，Nginx 模块可以为其创建的变量选择使用值容器，作为其“取处理程序”计算结果的缓存。显然，ngx_map 模块认为变量间的映射计算足够昂贵，需要自动将因变量的计算结果缓存下来，这样在当前请求的处理过程中如果再次读取这个因变量，Nginx 就可以直接返回缓存住的结果，而不再调用该变量的“取处理程序”再行计算了。
为了进一步验证这一点，我们不妨在请求中直接指定 URL 参数串为 debug:

```
$ curl 'http://localhost:8080/test?debug'
original foo: 1
foo: 1
```

我们看到，现在 $orig_foo 的值就成了 1，因为变量 $foo 在第一次被读取时，自变量 $args 的值就是 debug，于是按照映射规则，“取处理程序”计算返回的值便是 1. 而后续再读取 $foo 的值时，就总是得到被缓存住的 1 这个结果，而不论 $args 后来变成什么样了。
map 指令其实是一个比较特殊的例子，因为它可以为用户变量注册“取处理程序”，而且用户可以自己定义这个“取处理程序”的计算规则。当然，此规则在这里被限定为与另一个变量的映射关系。同时，也并非所有使用了“取处理程序”的变量都会缓存结果，例如我们前面在 （三） 中已经看到 $arg_XXX 并不会使用值容器进行缓存。
类似 ngx_map 模块，标准的 ngx_geo 等模块也一样使用了变量值的缓存机制。

在上面的例子中，我们还应当注意到 map 指令是在 server 配置块之外，也就是在最外围的 http 配置块中定义的。很多读者可能会对此感到奇怪，毕竟我们只是在 location /test 中用到了它。这倒不是因为我们不想把 map 语句直接挪到 location 配置块中，而是因为 map 指令只能在 http 块中使用！

很多 Nginx 新手都会担心如此“全局”范围的 map 设置会让访问所有虚拟主机的所有 location 接口的请求都执行一遍变量值的映射计算，然而事实并非如此。前面我们已经了解到 map 配置指令的工作原理是为用户变量注册 “取处理程序”，并且实际的映射计算是在“取处理程序”中完成的，而“取处理程序”只有在该用户变量被实际读取时才会执行（当然，因为缓存的存在，只在请求生命期中的第一次读取中才被执行），所以对于那些根本没有用到相关变量的请求来说，就根本不会执行任何的无用计算。

这种只在实际使用对象时才计算对象值的技术，在计算领域被称为“惰性求值”（lazy evaluation）。提供“惰性求值” 语义的编程语言并不多见，最经典的例子便是 Haskell. 与之相对的便是“主动求值” （eager evaluation）。我们有幸在 Nginx 中也看到了“惰性求值”的例子，但“主动求值”语义其实在 Nginx 里面更为常见，例如下面这行再普通不过的 set 语句：

```
set  $b  "$a,$a";
```

这里会在执行 set 规定的赋值操作时，“主动”地计算出变量 $b 的值，而不会将该求值计算延缓到变量 $b 实际被读取的时候。

nginx 变量使用方法详解(5)

前面在 （二） 中我们已经了解到变量值容器的生命期是与请求绑定的，但是我当时有意避开了“请求”的正式定义。大家应当一直默认这里的“请求”都是指客户端发起的 HTTP 请求。其实在 Nginx 世界里有两种类型的“请求”，一种叫做“主请求”（main request），而另一种则叫做“子请求”（subrequest）。我们先来介绍一下它们。

所谓“主请求”，就是由 HTTP 客户端从 Nginx 外部发起的请求。我们前面见到的所有例子都只涉及到“主请求”，包括 （二） 中那两个使用 echo_exec 和 rewrite 指令发起“内部跳转”的例子。

而“子请求”则是由 Nginx 正在处理的请求在 Nginx 内部发起的一种级联请求。“子请求”在外观上很像 HTTP 请求，但实现上却和 HTTP 协议乃至网络通信一点儿关系都没有。它是 Nginx 内部的一种抽象调用，目的是为了方便用户把“主请求”的任务分解为多个较小粒度的“内部请求”，并发或串行地访问多个 location 接口，然

后由这些 location 接口通力协作，共同完成整个“主请求”。当然，“子请求”的概念是相对的，任何一个“子请求”也可以再发起更多的“子子请求”，甚至可以玩递归调用（即自己调用自己）。当一个请求发起一个“子请求”的时候，按照 Nginx 的术语，习惯把前者称为后者的“父请求”（parent request）。值得一提的是，Apache 服务器中其实也有“子请求”的概念，所以来自 Apache 世界的读者对此应当不会感到陌生。

下面就来看一个使用了“子请求”的例子：

```
location /main {
    echo_location /foo;
    echo_location /bar;
}

location /foo {
    echo foo;
}

location /bar {
    echo bar;
}
```

这里在 location /main 中，通过第三方 ngx_echo 模块的 echo_location 指令分别发起到 /foo 和 /bar 这两个接口的 GET 类型的“子请求”。由 echo_location 发起的“子请求”，其执行是按照配置书写的顺序串行处理的，即只有当 /foo 请求处理完毕之后，才会接着处理 /bar 请求。这两个“子请求”的输出会按执行顺序拼接起来，作为 /main 接口的最终输出：

```
$ curl 'http://localhost:8080/main'
foo
bar
```

我们看到，“子请求”方式的通信是在同一个虚拟主机内部进行的，所以 Nginx 核心在实现“子请求”的时候，就只调用了若干个 C 函数，完全不涉及任何网络或者 UNIX 套接字（socket）通信。我们由此可以看出“子请求”的执行效率是极高的。

回到先前对 Nginx 变量值容器的生命期的讨论，我们现在依旧可以说，它们的生命期是与当前请求相关联的。每个请求都有所有变量值容器的独立副本，只不过当前请求既可以是“主请求”，也可以是“子请求”。即便是父子请求之间，同名变量一般也不会相互干扰。让我们来通过一个小实验证明一下这个说法：

```
location /main {
    set $var main;
    echo_location /foo;
    echo_location /bar;
    echo "main: $var";
}

location /foo {
    set $var foo;
    echo "foo: $var";
}

location /bar {
    set $var bar;
    echo "bar: $var";
}
```

在这个例子中，我们分别在 /main，/foo 和 /bar 这三个 location 配置块中为同一名字的变量，$var，分别设置了不同的值并予以输出。特别地，我们在 /main 接口中，故意在调用过 /foo 和 /bar 这两个“子请求”之后，再输出它自己的 $var 变量的值。请求 /main 接口的结果是这样的：

```
$ curl 'http://localhost:8080/main'
foo: foo
bar: bar
main: main
```

显然，/foo 和 /bar 这两个“子请求”在处理过程中对变量 $var 各自所做的修改都丝毫没有影响到“主请求”/main. 于是这成功印证了“主请求”以及各个“子请求”都拥有不同的变量 $var 的值容器副本。

不幸的是，一些 Nginx 模块发起的“子请求”却会自动共享其“父请求”的变量值容器，比如第三方模块 ngx_auth_request. 下面是一个例子：

```
location /main {
    set $var main;
    auth_request /sub;
    echo "main: $var";
}

location /sub {
    set $var sub;
    echo "sub: $var";
}
```

这里我们在 /main 接口中先为 $var 变量赋初值 main，然后使用 ngx_auth_request 模块提供的配置指令 auth_request，发起一个到 /sub 接口的“子请求”，最后利用 echo 指令输出变量 $var 的值。而我们在 /sub 接口中则故意把 $var 变量的值改写成 sub. 访问 /main 接口的结果如下：

```
$ curl 'http://localhost:8080/main'
main: sub
```

我们看到，/sub 接口对 $var 变量值的修改影响到了主请求 /main. 所以 ngx_auth_request 模块发起的“子请求”确实是与其“父请求”共享一套 Nginx 变量的值容器。

对于上面这个例子，相信有读者会问：“为什么‘子请求’ /sub 的输出没有出现在最终的输出里呢？”答案很简单，那就是因为 auth_request 指令会自动忽略“子请求”的响应体，而只检查“子请求”的响应状态码。当状态码是 2XX 的时候，auth_request 指令会忽略“子请求”而让 Nginx 继续处理当前的请求，否则它就会立即中断当前（主）请求的执行，返回相应的出错页。在我们的例子中，/sub “子请求”只是使用 echo 指令作了一些输出，所以隐式地返回了指示正常的 200 状态码。

如 ngx_auth_request 模块这样父子请求共享一套 Nginx 变量的行为，虽然可以让父子请求之间的数据双向传递变得极为容易，但是对于足够复杂的配置，却也经常导致不少难于调试的诡异 bug. 因为用户时常不知道“父请求”的某个 Nginx 变量的值，其实已经在它的某个“子请求”中被意外修改了。诸如此类的因共享而导致的不好的“副作用”，让包括 ngx_echo，ngx_lua，以及 ngx_srcache 在内的许多第三方模块都选择了禁用父子请求间的变量共享。

nginx 变量使用方法详解(6)

Nginx 内建变量用在“子请求”的上下文中时，其行为也会变得有些微妙。

前面在 （三） 中我们已经知道，许多内建变量都不是简单的“存放值的容器”，它们一般会通过注册“存取处理程序”来表现得与众不同，而它们即使有存放值的容器，也只是用于缓存“存取处理程序”的计算结果。我们之前讨论过的 $args 变量正是通过它的“取处理程序”来返回当前请求的 URL 参数串。因为当前请求也可以是“子请求”，所以在“子请求”中读取 $args，其“取处理程序”会很自然地返回当前“子请求”的参数串。我们来看这样的一个例子：

```
location /main {
    echo "main args: $args";
    echo_location /sub "a=1&b=2";
}

location /sub {
    echo "sub args: $args";
}
```

这里在 /main 接口中，先用 echo 指令输出当前请求的 $args 变量的值，接着再用 echo_location 指令发起子请求 /sub. 这里值得注意的是，我们在 echo_location 语句中除了通过第一个参数指定“子请求”的 URI 之外，还提供了第二个参数，用以指定该“子请求”的 URL 参数串（即 a=1&b=2）。最后我们定义了 /sub 接口，在里面输出了一下 $args 的值。请求 /main 接口的结果如下：

```
$ curl 'http://localhost:8080/main?c=3'
main args: c=3
sub args: a=1&b=2
```

显然，当 $args 用在“主请求” /main 中时，输出的就是“主请求”的 URL 参数串，c=3；而当用在“子请求” /sub 中时，输出的则是“子请求”的参数串，a=1&b=2。这种行为正符合我们的直觉。

与 $args 类似，内建变量 $uri 用在“子请求”中时，其“取处理程序”也会正确返回当前“子请求”解析过的 URI:

```
location /main {
    echo "main uri: $uri";
    echo_location /sub;
}

location /sub {
    echo "sub uri: $uri";
}
```

请求 /main 的结果是

```
$ curl 'http://localhost:8080/main'
main uri: /main
sub uri: /sub
```

这依然是我们所期望的。

但不幸的是，并非所有的内建变量都作用于当前请求。少数内建变量只作用于“主请求”，比如由标准模块 ngx_http_core 提供的内建变量 $request_method.

变量 $request_method 在读取时，总是会得到“主请求”的请求方法，比如 GET、POST 之类。我们来测试一下：

```
location /main {
    echo "main method: $request_method";
```

```
    echo_location /sub;
}

location /sub {
    echo "sub method: $request_method";
}
```

在这个例子里，/main 和 /sub 接口都会分别输出 $request_method 的值。同时，我们在 /main 接口里利用 echo_location 指令发起一个到 /sub 接口的 GET “子请求”。我们现在利用 curl 命令行工具来发起一个到 /main 接口的 POST 请求：

```
$ curl --data hello 'http://localhost:8080/main'
main method: POST
sub method: POST
```

这里我们利用 curl 程序的 -data 选项，指定 hello 作为我们的请求体数据，同时 -data 选项会自动让发送的请求使用 POST 请求方法。测试结果证明了我们先前的预言，$request_method 变量即使在 GET “子请求” /sub 中使用，得到的值依然是“主请求” /main 的请求方法，POST.

有的读者可能觉得我们在这里下的结论有些草率，因为上例是先在“主请求”里读取（并输出）$request_method 变量，然后才发“子请求”的，所以这些读者可能认为这并不能排除 $request_method 在进入子请求之前就已经把第一次读到的值给缓存住，从而影响到后续子请求中的输出结果。不过，这样的顾虑是多余的，因为我们前面在 （五） 中也特别提到过，缓存所依赖的变量的值容器，是与当前请求绑定的，而由 ngx_echo 模块发起的“子请求”都禁用了父子请求之间的变量共享，所以在上例中，$request_method 内建变量即使真的使用了值容器作为缓存（事实上它也没有），它也不可能影响到 /sub 子请求。

为了进一步消除这部分读者的疑虑，我们不妨稍微修改一下刚才那个例子，将 /main 接口输出 $request_method 变量的时间推迟到“子请求”执行完毕之后：

```
location /main {
    echo_location /sub;
    echo "main method: $request_method";
}
location /sub {
    echo "sub method: $request_method";
}
```

让我们重新测试一下：

```
$ curl --data hello  'http://localhost:8080/main'
sub method: POST
main method: POST
```

可以看到，再次以 POST 方法请求 /main 接口的结果与原先那个例子完全一致，除了父子请求的输出顺序颠倒了过来（因为我们在本例中交换了 /main 接口中那两条输出配置指令的先后次序）。

由此可见，我们并不能通过标准的 $request_method 变量取得“子请求”的请求方法。为了达到我们最初的目的，我们需要求助于第三方模块 ngx_echo 提供的内建变量 $echo_request_method：

```
location /main {
    echo "main method: $echo_request_method";
    echo_location /sub;
}

location /sub {
    echo "sub method: $echo_request_method";
}
```

此时的输出终于是我们想要的了：

```
$ curl --data hello 'http://localhost:8080/main'
main method: POST
sub method: GET
```

我们看到，父子请求分别输出了它们各自不同的请求方法，POST 和 GET.

类似 $request_method，内建变量 $request_uri 一般也返回的是“主请求”未经解析过的 URL，毕竟“子请求”都是在 Nginx 内部发起的，并不存在所谓的“未解析的”原始形式。

如果真如前面那部分读者所担心的，内建变量的值缓存在共享变量的父子请求之间起了作用，这无疑是灾难性的。我们前面在 （五） 中已经看到 ngx_auth_request 模块发起的“子请求”是与其“父请求”共享一套变量的。下面是一个这样的可怕例子：

```
map $uri $tag {
    default     0;
    /main       1;
    /sub        2;
}

server {
    listen 8080;

    location /main {
        auth_request /sub;
        echo "main tag: $tag";
    }

    location /sub {
        echo "sub tag: $tag";
    }
}
```

这里我们使用久违了的 map 指令来把内建变量 $uri 的值映射到用户变量 $tag 上。当 $uri 的值为 /main 时，则赋予 $tag 值 1，当 $uri 取值 /sub 时，则赋予 $tag 值 2，其他情况都赋 0. 接着，我们在 /main 接口中先用 ngx_auth_request 模块的 auth_request 指令发起到 /sub 接口的子请求，然后再输出变量 $tag 的值。而在 /sub 接口中，我们直接输出变量 $tag. 猜猜看，如果我们访问接口 /main，将会得到什么样的输出呢？

```
$ curl 'http://localhost:8080/main'
main tag: 2
```

咦？我们不是分明把 /main 这个值映射到 1 上的么？为什么实际输出的是 /sub 映射的结果 2 呢？

其实道理很简单，因为我们的 $tag 变量在“子请求” /sub 中首先被读取，于是在那里计算出了值 2（因为 $uri 在那里取值 /sub，而根据 map 映射规则，$tag 应当取值 2），从此就被 $tag 的值容器给缓存住了。而 auth_request 发起的“子请求”又是与“父请求”共享一套变量的，于是当 Nginx 的执行流回到“父请求”输出 $tag 变量的值时，Nginx 就直接返回缓存住的结果 2 了。这样的结果确实太意外了。

从这个例子我们再次看到，父子请求间的变量共享，实在不是一个好主意。

nginx 变量使用方法详解(7)

在 （一） 中我们提到过，Nginx 变量的值只有一种类型，那就是字符串，但是变量也有可能压根就不存在有意义的值。没有值的变量也有两种特殊的值：一种是“不合法”（invalid），另一种是“没找到”（not found）。

举例说来，当 Nginx 用户变量 $foo 创建了却未被赋值时，$foo 的值便是“不合法”；而如果当前请求的 URL 参数串中并没有提及 XXX 这个参数，则 $arg_XXX 内建变量的值便是“没找到”。

无论是“不合法”也好，还是“没找到”也罢，这两种 Nginx 变量所拥有的特殊值，和空字符串（”“）这种取值是完全不同的，比如 JavaScript 语言中也有专门的 undefined 和 null 这两种特殊值，而 Lua 语言中也有专门的 nil 值：它们既不等同于空字符串，也不等同于数字 0，更不是布尔值 false. 其实 SQL 语言中的 NULL 也是类似的一种东西。

虽然前面在 （一） 中我们看到，由 set 指令创建的变量未初始化就用在“变量插值”中时，效果等同于空字符串，但那是因为 set 指令为它创建的变量自动注册了一个“取处理程序”，将“不合法”的变量值转换为空字符串。为了验证这一点，我们再重新看一下 （一） 中讨论过的那个例子：

```
location /foo {
    echo "foo = [$foo]";
}

location /bar {
    set $foo 32;
    echo "foo = [$foo]";
}
```

这里为了简单起见，省略了原先写出的外围 server 配置块。在这个例子里，我们在 /bar 接口中用 set 指令隐式地创建了 $foo 变量这个名字，然后我们在 /foo 接口中不对 $foo 进行初始化就直接使用 echo 指令输出。我们当时测试 /foo 接口的结果是

```
$ curl 'http://localhost:8080/foo'
foo = []
```

从输出上看，未初始化的 $foo 变量确实和空字符串的效果等同。但细心的读者当时应该就已经注意到，对于上面这个请求，Nginx 的错误日志文件（一般文件名叫做 error.log）中多出一行类似下面这样的警告：

```
[warn] 5765#0: *1 using uninitialized "foo" variable, ...
```

这一行警告是谁输出的呢？答案是 set 指令为 $foo 注册的“取处理程序”。当 /foo 接口中的 echo 指令实际执行的时候，它会对它的参数 “foo = [$foo]” 进行“变量插值”计算。于是，参数串中的 $foo 变量会被读取，而 Nginx 会首先检查其值容器里的取值，结果它看到了“不合法”这个特殊值，于是它这才决定继续调用 $foo 变量的“取处理程序”。于是 $foo 变量的“取处理程序”开始运行，它向 Nginx 的错误日志打印出上面那条警告消息，然后返回一个空字符串作为 $foo 的值，并从此缓存在 $foo 的值容器中。

细心的读者会注意到刚刚描述的这个过程其实就是那些支持值缓存的内建变量的工作原理，只不过 set 指令在这里借用了这套机制来处理未正确初始化的 Nginx 变量。值得一提的是，只有“不合法”这个特殊值才会触发 Nginx 调用变量的“取处理程序”，而特殊值“没找到”却不会。

上面这样的警告一般会指示出我们的 Nginx 配置中存在变量名拼写错误，抑或是在错误的场合使用了尚未初始化的变量。因为值缓存的存在，这条警告在一个请求的生命期中也不会打印多次。当然，ngx_rewrite 模块专门提供了一条 uninitialized_variable_warn 配置指令可用于禁止这条警告日志。

刚才提到，内建变量 $arg_XXX 在请求 URL 参数 XXX 并不存在时会返回特殊值“找不到”，但遗憾的是在 Nginx 原生配置语言（我们估且这么称呼它）中是不能很方便地把它和空字符串区分开来的，比如：

```
location /test {
    echo "name: [$arg_name]";
}
```

这里我们输出 $arg_name 变量的值同时故意在请求中不提供 URL 参数 name:

```
$ curl 'http://localhost:8080/test'
name: []
```

我们看到，输出特殊值“找不到”的效果和空字符串是相同的。因为这一回是 Nginx 的“变量插值”引擎自动把“找不到”给忽略了。

那么我们究竟应当如何捕捉到“找不到”这种特殊值的踪影呢？换句话说，我们应当如何把它和空字符串给区分开来呢？显然，下面这个请求中，URL 参数 name 是有值的，而且其值应当是空字符串：

```
$ curl 'http://localhost:8080/test?name='
name: []
```

但我们却无法将之和前面完全不提供 name 参数的情况给区分开。

幸运的是，通过第三方模块 ngx_lua，我们可以轻松地在 Lua 代码中做到这一点。请看下面这个例子：

```
location /test {
    content_by_lua '
        if ngx.var.arg_name == nil then
            ngx.say("name: missing")
        else
            ngx.say("name: [", ngx.var.arg_name, "]")
        end
    ';
}
```

这个例子和前一个例子功能上非常接近，除了我们在 /test 接口中使用了 ngx_lua 模块的 content_by_lua 配置指令，嵌入了一小段我们自己的 Lua 代码来对 Nginx 变量 $arg_name 的特殊值进行判断。在这个例子中，当 $arg_name 的值为“没找到”（或者“不合法”）时，/foo 接口会输出 name: missing 这一行结果:

```
curl 'http://localhost:8080/test'
name: missing
```

因为这是我们第一次接触到 ngx_lua 模块，所以需要先简单介绍一下。ngx_lua 模块将 Lua 语言解释器（或者 LuaJIT 即时编译器）嵌入到了 Nginx 核心中，从而可以让用户在 Nginx 核心中直接运行 Lua 语言编写的程序。我们可以选择在 Nginx 不同的请求处理阶段插入我们的 Lua 代码。这些 Lua 代码既可以直接内联在 Nginx 配置文件中，也可以单独放置在外部 .lua 文件里，然后在 Nginx 配置文件中引用 .lua 文件的路径。
回到上面这个例子，我们在 Lua 代码里引用 Nginx 变量都是通过 ngx.var 这个由 ngx_lua 模块提供的 Lua 接口。比如引用 Nginx 变量 $VARIABLE 时，就在 Lua 代码里写作 ngx.var.VARIABLE 就可以了。当 Nginx 变量 $arg_name 为特殊值“没找到”（或者“不合法”）时， ngx.var.arg_name 在 Lua 世界中的值就是 nil，即 Lua 语言里的“空”（不同于 Lua 空字符串）。我们在 Lua 里输出响应体内容的时候，则使用了 ngx.say 这个 Lua 函数，也是 ngx_lua 模块提供的，功能上等价于 ngx_echo 模块的 echo 配置指令。
现在，如果我们提供空字符串取值的 name 参数，则输出就和刚才不相同了：

```
$ curl 'http://localhost:8080/test?name='
name: []
```

在这种情况下，Nginx 变量 $arg_name 的取值便是空字符串，这既不是“没找到”，也不是“不合法”，因此在 Lua 里，ngx.var.arg_name 就返回 Lua 空字符串（“”），和刚才的 Lua nil 值就完全区分开了。
这种区分在有些应用场景下非常重要，比如有的 web service 接口会根据 name 这个 URL 参数是否存在来决定是否按 name 属性对数据集合进行过滤，而显然提供空字符串作为 name 参数的值，也会导致对数据集中取值为空串的记录进行筛选操作。

不过，标准的 $arg_XXX 变量还是有一些局限，比如我们用下面这个请求来测试刚才那个 /test 接口：

```
$ curl 'http://localhost:8080/test?name'
name: missing
```

此时，$arg_name 变量仍然读出“找不到”这个特殊值，这就明显有些违反常识。此外，$arg_XXX 变量在请求 URL 中有多个同名 XXX 参数时，就只会返回最先出现的那个 XXX 参数的值，而默默忽略掉其他实例：

```
$ curl 'http://localhost:8080/test?name=Tom&name=Jim&name=Bob'
name: [Tom]
```

要解决这些局限，可以直接在 Lua 代码中使用 ngx_lua 模块提供的 ngx.req.get_uri_args 函数。

nginx 变量使用方法详解(8)

与 $arg_XXX 类似，我们在 （二） 中提到过的内建变量 $cookie_XXX 变量也会在名为 XXX 的 cookie 不存在时返回特殊值“没找到”：

```
location /test {
    content_by_lua '
        if ngx.var.cookie_user == nil then
            ngx.say("cookie user: missing")
        else
            ngx.say("cookie user: [", ngx.var.cookie_user, "]")
        end
    ';
}
```

利用 curl 命令行工具的 -cookie name=value 选项可以指定 name=value 为当前请求携带的 cookie（通过添加相应的 Cookie 请求头）。下面是若干次测试结果：

```
$ curl --cookie user=agentzh 'http://localhost:8080/test'
cookie user: [agentzh]

$ curl --cookie user= 'http://localhost:8080/test'
cookie user: []

$ curl 'http://localhost:8080/test'
cookie user: missing
```

我们看到，cookie user 不存在以及取值为空字符串这两种情况被很好地区分开了：当 cookie user 不存在时，Lua 代码中的 ngx.var.cookie_user 返回了期望的 Lua nil 值。

在 Lua 里访问未创建的 Nginx 用户变量时，在 Lua 里也会得到 nil 值，而不会像先前的例子那样直接让 Nginx 拒绝加载配置：

```
location /test {
    content_by_lua '
       ngx.say("$blah = ", ngx.var.blah)
    ';
}
```

这里假设我们并没有在当前的 nginx.conf 配置文件中创建过用户变量 $blah，然后我们在 Lua 代码中通过 ngx.var.blah 直接引用它。上面这个配置可以顺利启动，因为 Nginx 在加载配置时只会编译 content_by_lua 配置指令指定的 Lua 代码而不会实际执行它，所以 Nginx 并不知道 Lua 代码里面引用了 $blah 这个变量。于是我们在运行时也会得到 nil 值。而 ngx_lua 提供的 ngx.say 函数会自动把 Lua 的 nil 值格式化为字符串“nil” 输出，于是访问 /test 接口的结果是：

```
curl 'http://localhost:8080/test'
$blah = nil
```

这正是我们所期望的。

上面这个例子中另一个值得注意的地方是，我们在 content_by_lua 配置指令的参数中提及了 $bar 符号，但却并没有触发“变量插值”（否则 Nginx 会在启动时抱怨 $blah 未创建）。这是因为 content_by_lua 配置指令并不支持参数的“变量插值”功能。我们前面在 （一） 中提到过，配置指令的参数是否允许“变量插值”，其实取决于该指令的实现模块。

设计返回“不合法”这一特殊值的例子是困难的，因为我们前面在 （七） 中已经看到，由 set 指令创建的变量在未初始化时确实是“不合法”，但一旦尝试读取它们时，Nginx 就会自动调用其“取处理程序”，而它们的“取处理程序”会自动返回空字符串并将之缓存住。于是我们最终得到的是完全合法的空字符串。下面这个使用了 Lua 代码的例子证明了这一点：

```
location /foo {
    content_by_lua '
        if ngx.var.foo == nil then
            ngx.say("$foo is nil")
        else
            ngx.say("$foo = [", ngx.var.foo, "]")
        end
    ';
}

location /bar {
    set $foo 32;
    echo "foo = [$foo]";
}
```

请求 /foo 接口的结果是：

```
$ curl 'http://localhost:8080/foo'
$foo = []
```

我们看到在 Lua 里面读取未初始化的 Nginx 变量 $foo 时得到的是空字符串。

最后值得一提的是，虽然前面反复指出 Nginx 变量只有字符串这一种数据类型，但这并不能阻止像 ngx_array_var 这样的第三方模块让 Nginx 变量也能存放数组类型的值。下面就是这样的一个例子：

```
location /test {
    array_split "," $arg_names to=$array;
    array_map "[$array_it]" $array;
    array_join " " $array to=$res;
    echo $res;
}
```

这个例子中使用了 ngx_array_var 模块的 array_split、 array_map 和 array_join 这三条配置指令，其含义很接近 Perl 语言中的内建函数 split、map 和 join（当然，其他脚本语言也有类似的等价物）。我们来看看访问 /test 接口的结果：

```
$ curl 'http://localhost:8080/test?names=Tom,Jim,Bob
[Tom] [Jim] [Bob]
```

我们看到，使用 ngx_array_var 模块可以很方便地处理这样具有不定个数的组成元素的输入数据，例如此例中的 names URL 参数值就是由不定个数的逗号分隔的名字所组成。不过，这种类型的复杂任务通过 ngx_lua 来做通常会更灵活而且更容易维护。

至此，本系列教程对 Nginx 变量的介绍终于可以告一段落了。我们在这个过程中接触到了许多标准的和第三方的 Nginx 模块，这些模块让我们得以很轻松地构造出许多有趣的小例子，从而可以深入探究 Nginx 变量的各种行为和特性。在后续的教程中，我们还会有很多机会与这些模块打交道。

通过前面讨论过的众多例子，我们应当已经感受到 Nginx 变量在 Nginx 配置语言中所扮演的重要角色：它是获取 Nginx 中各种信息（包括当前请求的信息）的主要途径和载体，同时也是各个模块之间传递数据的主要媒介之一。在后续的教程中，我们会经常看到 Nginx 变量的身影，所以现在很好地理解它们是非常重要的。

在下一个系列的教程，即 Nginx 配置指令的执行顺序系列 中，我们将深入探讨 Nginx 配置指令的执行顺序以及请求的各个处理阶段，因为很多 Nginx 用户都搞不清楚他们书写的众多配置指令之间究竟是按照何种时间顺序执行的，也搞不懂为什么这些指令实际执行的顺序经常和配置文件里的书写顺序大相径庭。

[link_post name=”
nginx%e5%8f%98%e9%87%8f%e4%bd%bf%e7%94%a8%e6%96%b9%e6%b3%95%e8%af%a6%e8%a7%a31″]|[link_post
name=”nginx%e5%8f%98%e9%87%8f%e4%bd%bf%e7%94%a8%e6%96%b9%e6%b3%95%e8%af%a6%e8%a7%a32″]|[link_post
name=”nginx%e5%8f%98%e9%87%8f%e4%bd%bf%e7%94%a8%e6%96%b9%e6%b3%95%e8%af%a6%e8%a7%a33″]|[link_post
name=”nginx%e5%8f%98%e9%87%8f%e4%bd%bf%e7%94%a8%e6%96%b9%e6%b3%95%e8%af%a6%e8%a7%a34″]|[link_post
name=”nginx%e5%8f%98%e9%87%8f%e4%bd%bf%e7%94%a8%e6%96%b9%e6%b3%95%e8%af%a6%e8%a7%a35″]|[link_post
name=”nginx%e5%8f%98%e9%87%8f%e4%bd%bf%e7%94%a8%e6%96%b9%e6%b3%95%e8%af%a6%e8%a7%a36″]|[link_post
name=”nginx%e5%8f%98%e9%87%8f%e4%bd%bf%e7%94%a8%e6%96%b9%e6%b3%95%e8%af%a6%e8%a7%a37″]|[link_post
name=”nginx%e5%8f%98%e9%87%8f%e4%bd%bf%e7%94%a8%e6%96%b9%e6%b3%95%e8%af%a6%e8%a7%a38″]

Nginx 模块与案例

01.如何安装 nginx 第三方模块

nginx 文件非常小但是性能非常的高效,这方面完胜 apache,nginx 文件小的一个原因之一是 nginx 自带的功能相对较少,好在 nginx 允许第三方模块,第三方模块使得 nginx 越发的强大. 在安装模块方面,nginx 显得没有 apache 安装模块方便，当然也没有 php 安装扩展方便.在原生的 nginx,他不可以动态加载模块,所以当你安装第三方模块的时候需要覆盖 nginx 文件.接下来看看如何安装 nginx 第三模块吧.

nginx 第三方模块安装方法：

```
./configure --prefix=/你的安装目录  --add-module=/第三方模块目录
```

以安装 pagespeed 模块实例

在未安装 nginx 的情况下安装 nginx 第三方模块

```
# ./configure --prefix=/usr/local/nginx-1.4.1 \
--with-http_stub_status_module \
--with-http_ssl_module --with-http_realip_module \
--with-http_image_filter_module \
--add-module=../ngx_pagespeed-master --add-module=/第三方模块目录
# make
# make isntall
# /usr/local/nginx-1.4.1/sbin/nginx
```

在已安装 nginx 情况下安装 nginx 模块

```
# ./configure --prefix=/usr/local/nginx-1.4.1 \
 --with-http_stub_status_module \
 --with-http_ssl_module --with-http_realip_module \
 --with-http_image_filter_module \
 --add-module=../ngx_pagespeed-master
# make
# /usr/local/nginx-1.4.1/sbin/nginx -s stop
# cp objs/nginx /usr/local/nginx/sbin/nginx
# /usr/local/nginx-1.4.1/sbin/nginx
```

相比之下仅仅多了一步覆盖 nginx 文件.

总结,安装 nginx 安装第三方模块实际上是使用 - add-module 重新安装一次 nginx，不要 make install 而是直接把编译目录下 objs/nginx 文件直接覆盖老的 nginx 文件.如果你需要安装多个 nginx 第三方模块,你只需要多指定几个相应的 - add-module 即可.

[warning]备注：重新编译的时候，记得一定要把以前编译过的模块一同加到 configure 参数里面.[/warning]

nginx 提供了非常多的 nginx 第三方模块提供安装,地址 http://wiki.nginx.org/3rdPartyModules

02.srcache_nginx redis 构建缓存系统应用一例

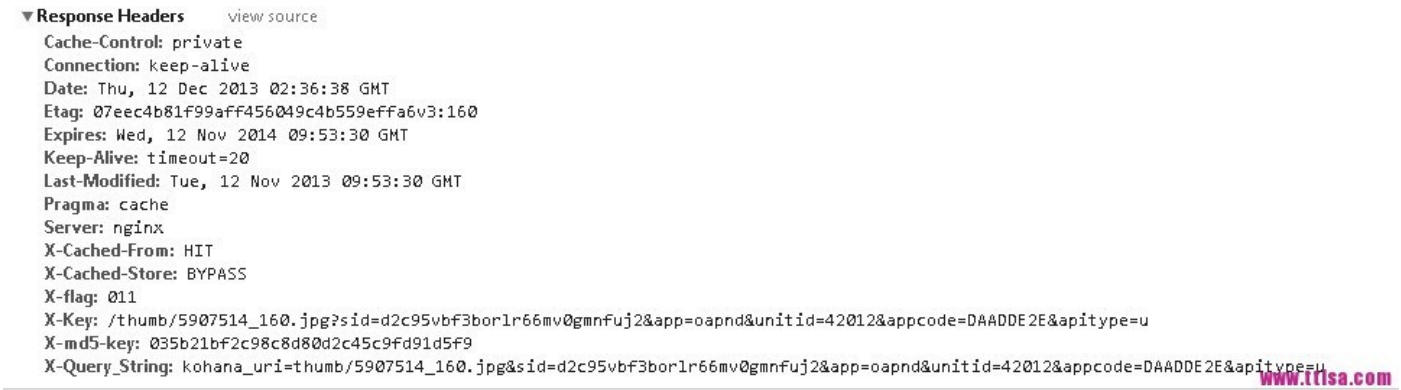
srcache_nginx 模块相关参数介绍，可以参见《memc_nginx+srcache_nginx+memcached 构建透明的动态页面缓存》。redis 是一种高效的 key-value 存储。

下面举一例应用，看配置：

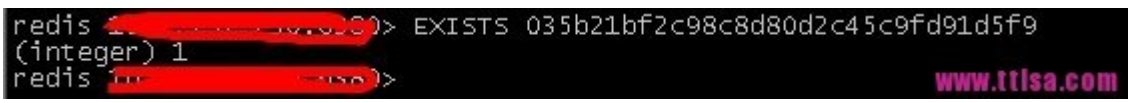
```
upstream redis {
    server 127.0.0.1:6380;
    keepalive 512;
}

server {
    listen       80 backlog=1024 default;
    server_name  www.ttlsa.com;
    index index.html index.htm index.php;
    root  /data/wwwroot/www.ttlsa.com/webroot;

    location / {
        set $flag 0;
        if ($uri ~ /thumb/[0-9]+_160.jpg$){
                set $flag "${flag}1";
        }
        if ($arg_unitid = 42012){
                set $flag "${flag}1";
        }
        if (!-e $request_filename) {
                rewrite ^/(.*)$ /index.php?kohana_uri=$1 last;
        }
    }
    location ~ .*\.php?$ {
        srcache_store_private on;
        srcache_methods GET;
        srcache_response_cache_control off;
        if ($flag = "011"){
            set $key $request_uri;
            set_escape_uri $escaped_key $key;
            srcache_fetch GET /redis $key;
            srcache_default_expire 172800;
            srcache_store PUT /redis2 key=$escaped_key&exptime=$srcache_expire;

            add_header X-flag $flag;
            add_header X-Cached-From $srcache_fetch_status;
            add_header X-Cached-Store $srcache_store_status;
            add_header X-Key $key;
            set_md5 $md5key $key;
            add_header X-md5-key $md5key;
            add_header X-Query_String $query_string;
            add_header X-expire $srcache_expire;
        }
        include fastcgi_params;
        fastcgi_pass  127.0.0.1:10080;
        fastcgi_index index.php;
        fastcgi_connect_timeout 300;
```

```
        fastcgi_send_timeout 300;
        fastcgi_read_timeout 300;
        fastcgi_buffer_size 128k;
        fastcgi_buffers 4 256k;
        fastcgi_busy_buffers_size 256k;
        fastcgi_temp_file_write_size 256k;
        fastcgi_intercept_errors on;
        fastcgi_param  SCRIPT_FILENAME  $document_root$fastcgi_script_name;
    }

    location = /redis {
        internal;
        set_md5 $redis_key $args;
        redis_pass redis;
    }

    location = /redis2 {
        internal;

        set_unescape_uri $exptime $arg_exptime;
        set_unescape_uri $key $arg_key;
        set_md5 $key;

        redis2_query set $key $echo_request_body;
        redis2_query expire $key $exptime;
        redis2_pass redis;
    }
}
```

测试：



redis 实例下：



可以记录下日志来测试加缓存前后的耗时。日志格式如下：

```
log_format srcache_log '$remote_addr - $remote_user [$time_local] "$request" '
                       '"$status" $body_bytes_sent $request_time $bytes_sent $request_length '
                       '[$upstream_response_time]  [$srcache_fetch_status]  [$srcache_store_status]
[$srcache_expire]';
```

03.nginx+lua+redis 构建高并发应用

nginx+lua+redis 构建高并发应用

ngx_lua 将 lua 嵌入到 nginx，让 nginx 执行 lua 脚本，高并发，非阻塞的处理各种请求。

url 请求 nginx 服务器，然后 lua 查询 redis，返回 json 数据。

备注：centos 或者 redhat 系统请跳转到 nginx + ngx_lua 安装测试

一.安装 lua

```
# apt-get install lua5.1
# apt-get install liblua5.1-dev
# apt-get install liblua5.1-socket2
```

二.安装 nginx

```
# apt-get install git-core
# git clone https://github.com/simpl/ngx_devel_kit.git
# git clone https://github.com/chaoslawful/lua-nginx-module.git
# git clone https://github.com/agentzh/redis2-nginx-module.git
# git clone https://github.com/agentzh/set-misc-nginx-module.git
# git clone https://github.com/agentzh/echo-nginx-module.git
# git clone https://github.com/catap/ngx_http_upstream_keepalive.git
# apt-get install libpcre3 libpcre3-dev libltdl-dev libssl-dev libjpeg62 libjpeg62-dev libpng12-0 libpng12-dev libxml2-dev libcurl4-openssl-dev libmcrypt-dev autoconf libxslt1-dev libgd2-noxpm-dev libgeoip-dev libperl-dev -y
# wget http://nginx.org/download/nginx-1.0.8.tar.gz
# tar zxvf nginx-1.0.8.tar.gz
# cd nginx-1.0.8
# ./configure --prefix=/usr/local/nginx --with-debug --with-http_addition_module \
--with-http_dav_module --with-http_flv_module --with-http_geoip_module \
--with-http_gzip_static_module --with-http_image_filter_module --with-http_perl_module \
--with-http_random_index_module --with-http_realip_module --with-http_secure_link_module \
--with-http_stub_status_module --with-http_ssl_module --with-http_sub_module \
--with-http_xslt_module --with-ipv6 --with-sha1=/usr/include/openssl \
--with-md5=/usr/include/openssl --with-mail --with-mail_ssl_module \
--add-module=../ngx_devel_kit \
--add-module=../echo-nginx-module \
--add-module=../lua-nginx-module \
--add-module=../redis2-nginx-module \
--add-module=../ngx_http_upstream_keepalive \
--add-module=../set-misc-nginx-module
# make
# make install
```

三.安装 lua-redis-parser

```
# git clone https://github.com/agentzh/lua-redis-parser.git
# export LUA_INCLUDE_DIR=/usr/include/lua5.1
# make CC=gcc
# make install CC=gcc
```

四.安装 json

```
# wget http://files.luaforge.net/releases/json/json/0.9.50/json4lua-0.9.50.zip
# unzip json4lua-0.9.50.zip
# cp json4lua-0.9.50/json/json.lua /usr/share/lua/5.1/
```

五.安装 redis-lua

```
# git clone https://github.com/nrk/redis-lua.git
# cp redis-lua/src/redis.lua /usr/share/lua/5.1/
```

六.配置

```
user www-data;
worker_processes 8;
worker_cpu_affinity 00000001 00000010 00000100 00001000 00010000 00100000 01000000 10000000;
error_log logs/error.log notice;
pid logs/nginx.pid;
worker_rlimit_nofile 60000;

events {
worker_connections 1024;
use epoll;

}

http {
include mime.types;
default_type application/octet-stream;
access_log logs/access.log;
sendfile on;
tcp_nopush on;
tcp_nodelay on;
keepalive_timeout 60;
types_hash_max_size 2048;
server_tokens off;
lua_code_cache on;

upstream redis_pool {
server 192.168.1.39:6379;
keepalive 1024 single;//定义连接池大小，当连接数达到此数后，后续的连接为短连接

}
server {
listen 80;
server_name 192.168.1.211;

location /get_redis{
#internal;
set_unescape_uri $key $arg_key;
redis2_query hgetall $key;
redis2_pass redis_pool;
}
```

```
location /json {
content_by_lua_file conf/fuck.lua;
}
}
}
# vim fuck.lua
local json = require("json")
local parser = require("redis.parser")
local res = ngx.location.capture("/get_redis",{
args = { key = ngx.var.arg_key }
})
if res.status == 200 then
reply = parser.parse_reply(res.body)
value = json.encode(reply)
ngx.say(value)
a = json.decode(value)
ngx.say(a[2])
end
```

七.测试

```
# redis-cli -h 192.168.1.39
redis 192.168.1.39:6379> HMSET ttlsa www www.ttlsa.com mail mail.ttlsa.com

OK
# curl ‘http://192.168.1.211/json?key=ttlsa’
["www","www.ttlsa.com","mail","mail.ttlsa.com"]
```

www.ttlsa.com

04.ttserver+nginx 构建高并发高可用性应用

ttserver+nginx 构建高并发高可用性应用

ttserver 一款兼容 memcached 协议，也可以通过 HTTP 协议进行数据交换，支持故障转移，高可用性，高并发的分布式 key-value 持久存储系统。key-value 分布式存储系统的特点是查询快，存储数量大，高并发，非常适合通过主键进行查询的操作。

下面的案例是将图片以二进制的方式存入到 ttserver 中，并通过 http 方式读取图片。

一.配置 nginx

nginx_upstream_check_module 模块地址： https://github.com/yaoweibin/nginx_upstream_check_module

nginx 需要添加 nginx_upstream_check_module 模块，用于对后端服务器的健康情况检测，如果后端服务器不可用，则把这台服务器移除负载均衡轮循集群，所有的请求不往这台服务器上转发，待这台服务器恢复正常后，再把这台加入到负载均衡集群。这是 LB 的基本功能。

```
# vi nginx.conf
user www-data;
worker_processes 8;
worker_cpu_affinity 00000001 00000010 00000100 00001000 00010000 00100000 01000000 10000000;
```

```
error_log /var/log/nginx/error.log crit;
pid /var/run/nginx.pid;
worker_rlimit_nofile 65535;
events {
use epoll;
worker_connections 65535;
}
http {
include /etc/nginx/mime.types;
default_type application/octet-stream;
server_tokens off;
access_log off;
sendfile on;
tcp_nopush on;
keepalive_timeout 0;
tcp_nodelay on;
client_max_body_size 200m;
gzip on;
gzip_min_length 1k;
gzip_buffers 4 16k;
gzip_http_version 1.1;
gzip_comp_level 5;
gzip_disable “MSIE [1-6]\.(?!.*SV1)”;
gzip_types text/plain application/x-javascript text/css application/xml text/javascript;
include /etc/nginx/conf.d/*.conf;
include /etc/nginx/sites-enabled/*;
}
# vi default
server {
listen 80;
server_name 192.168.1.213;
memcached_connect_timeout 1s;
location / {
root /www/web/tmp;
error_page 404 = @fallback;
}
location ~ ^/ttlsa/ttlsa_([0-9a-zA-Z]\.+)$ {
set $memcached_key $1; //memcached 键值
add_header X-ttserver-key $memcached_key; //添加一个 header 信息
memcached_pass ttserver;
memcached_next_upstream error timeout; //当发生错误或超时时，将请求转发到 upstream 下一个服务器
default_type text/html;
error_page 404 = @fallback;
}
location @fallback {
rewrite ^ http://www.ttlsa.com redirect;
}
}
# vi upstream.conf
```

```
upstream ttserver {
server 192.168.1.60:1978;
server 192.168.1.60:1979;
check interval=3000 rise=2 fall=2 timeout=1000; //interval 检测周期 3s 一次，fall 宕机标记 2 次失败后标记不可用
}
```

二. 配置 ttserver

需要将 ttserver 配置成主主结构。

ttserver 的介绍，安装，配置参见： http://www.ttlsa.com/html/1220.html

三. 测试

1. 上传界面

```
# vi upload.php
<html>
<head>
<meta http-equiv=”Content-Type” content=”text/html; charset=utf-8″ />
<title>Upload Files</title>
</head>
<body>
<h2>Select files to upload</h2>
<form enctype=”multipart/form-data” action=”/store.php” method=”post”>
<input type=”file” name=”file”><br>
<input type=”submit” name=”submit” value=”Upload”>
</form>
</body>
</html>
```

2. 存入 ttserver

```
# vi store.php
[codesyntax lang=”php”]
<?php
/*
####################################
### author: www.ttlsa.com ###
### QQ 群: 39514058 ###
### E-mail: service@ttlsa.com ###
####################################
*/
print_r($_FILES);
echo “<br>”;
if($_FILES[‘file’][‘error’] !== 0){
die(‘Error upload file. Error code ‘.$_FILES[‘file’][‘error’]);
}
$filename=$_FILES[‘file’][‘name’];
$tmpfilepath=$_FILES[‘file’][‘tmp_name’];
$content=file_get_contents($tmpfilepath);
echo ‘<img src=”’, base64_encode($content), ‘” />';
echo “<br>”;
$ttserver=new Memcache;
$ttserver->addServer(‘192.168.1.60′,1978);
```

```
$ttserver->addServer(‘192.168.1.60’,1979);
$rt=$ttserver->set($filename,$content,0,0);
if($rt){
echo “存储成功!\n”;
}else{
echo “存储失败!\n”;
}
?>
```

[/codesyntax]

3. 结果截图

上传图片：

Array ([file] => Array ([name] => 0.jpg [type] => image/jpeg [tmp_name] =

Position Jul 2012	Position Jul 2011	Delta in Position	Programming Language	Ratings Jul 2012	Delta Jul 2011	Status
1	2	↑	C	18.331%	+1.05%	A
2	1	↓	Java	16.087%	-3.16%	A
3	6	↑↑↑	Objective-C	9.335%	+4.15%	A
4	3	↓	C++	9.118%	+0.10%	A
5	4	↓	C#	6.668%	+0.45%	A
6	7	↑	(Visual) Basic	5.695%	+0.59%	A
7	5	↓↓	PHP	5.012%	-1.17%	A
8	8	=	Python	4.000%	+0.42%	A
9	9	=	Perl	2.053%	-0.28%	A
10	12	↑↑	Ruby	1.768%	+0.44%	A
11	10	↓	JavaScript	1.454%	-0.79%	A
12	14	↑↑	Delphi/Object Pascal	1.157%	+0.27%	A
13	13	=	Lisp	0.997%	+0.09%	A
14	15	↑	Transact-SQL	0.954%	+0.15%	A
15	25	↑↑↑↑↑↑↑↑↑↑	Visual Basic .NET	0.917%	+0.43%	A
16	16	=	Pascal	0.837%	+0.17%	A
17	19	↑↑	Ada	0.689%	+0.14%	B
18	11	↓↓↓↓↓↓↓	Lua	0.684%	-0.89%	B
19	21	↑↑	PL/SQL	0.645%	+0.10%	A--
20	26	↑↑↑↑↑↑	MATLAB	0.639%	+0.19%	B

存储成功!

通过 HTTP 从 ttserver 中取图片：

存在问题：

1.二进制传输问题： 二进制数据通过非纯 8-bit 的传输层传输时，会出现错误。最好是经过 base64_encode 编码后再传输。

2.序列化问题： Tokyo Tyrant 使用 memcached 协议连接，用 php 的 memcached 客户端读取时不会自动反序列化。可使用 unserialize()函数进行反序列化操作。此问题稍后再议。

表情图片，用户头像等等场景都可以使用此种方案。比如以用户 ID 号作为键值等等。

05.nginx 生成缩略图配置 – ttlsa 教程系列之 nginx

为了手机端浏览到与手机分辨率相匹配的图片，提高 app 访问速度以及减少用户的手机流量，需要将图片生成缩略图，这边共有以下解决方案。

A.发布新闻生成多重缩略图 - 无法匹配到各种尺寸图片

B.当相应缩略图不存在，则使用 php 或者 java 等程序生成相应缩略图 - 需要程序员协助

C.使用 nginx 自带模块生成缩略图 - 运维即可完成

D.使用 nginx＋lua 生成缩略图

经过多方的考虑，决定使用方案 C，使用 nginx 自带模块生成缩略图，模块： -with-http_image_filter_module.
如下是我的安装参数：

```
./configure --prefix=/usr/local/nginx-1.4.1 --with-http_stub_status_module \
--with-http_realip_module --with-http_image_filter_module --with-debug
```

修改 nginx.conf 配置文件,或者放到你相应的 server 块中.

```
location ~* /(\d+)\.(jpg)$ {
set $h $arg_h;  # 获取参数 h 的值
set $w $arg_w;  # 获取参数 w 的值
#image_filter crop $h $w;
image_filter resize $h $w;#  根据给定的长宽生成缩略图
}

location ~* /(\d+)_(\d+)x(\d+)\.(jpg)$ {
if ( -e $document_root/$1.$4 ) {  # 判断原图是否存在
rewrite /(\d+)_(\d+)x(\d+)\.(jpg)$ /$1.$4?h=$2&w=$3 last;
    }
return 404;
}
```

例如图片：

http://test.ttlsa.com/123_100x10.jpg

1、首先判断是否存在原图 123.jpg,不存在直接返回 404（如果原图都不存在，还生成缩略图干啥，对吧）
2、跳转到 http://tset.ttlsa.com/123.jpg?h=100&w=10，将参数高 h 和宽 10 带到 url 中。
3、Image_filter resize 指令根据 h 和 w 参数生成相应缩略图。
备注：长宽取小，例如原图是 100*10,你传入的是 10*2，那么他会给你生成 10*1 的图片.

生成缩略图只是 image_filter 功能中的一个，它一共支持 4 种参数：
test:返回是否真的是图片
size:返回图片长短尺寸,返回 json 格式数据
corp:截取图片的一部分,从左上角开始截取,尺寸写小了,图片会被剪切
resize:缩放图片,等比例缩放

nginx 生成缩略图优缺点
优点：
1、根据传入参数即可生成各种比例图片
2、不占用任何硬盘空间

缺点：
1、消耗 CPU,访问量大将会给服务器带来极大的负担.

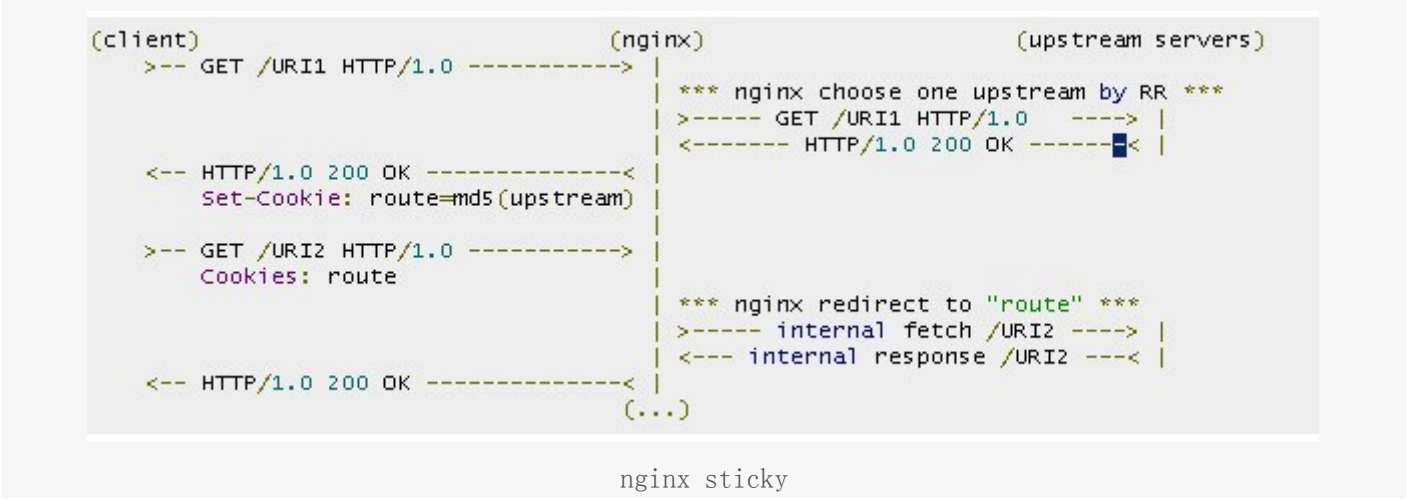
几点建议：
1、生成缩略是个消耗 cpu 的操作，如果访问量比较大的站点，最好考虑使用程序生成缩略图到硬盘上，或者在前端加上 cache 或者使用 CDN。

06.使用 nginx sticky 实现基于 cookie 的负载均衡

在多台后台服务器的环境下，我们为了确保一个客户只和一台服务器通信，我们势必使用长连接。使用什么方式来实现这种连接呢，常见的有使用 nginx 自带的 ip_hash 来做，我想这绝对不是一个好的办法，如果前端是 CDN，或者说一个局域网的客户同时访问服务器，导致出现服务器分配不均衡，以及不能保证每次访问都粘滞在同一台服务器。如果基于 cookie 会是一种什么情形，想想看，每台电脑都会有不同的 cookie，在保持长连接的同时还保证了服务器的压力均衡，nginx sticky 值得推荐。

如果浏览器不支持 cookie，那么 sticky 不生效，毕竟整个模块是给予 cookie 实现的.

1、nginx sticky 模块工作流程图



nginx sticky

2、下载安装 nginx sticky

下载地址：http://code.google.com/p/nginx-sticky-module/downloads/list

目前共有 2 个版本，一个是 1.0，一个是 1.1，1.0 已经寿终正寝了.1.1 增加了权重的参数.

安装 nginx + sticky 模块

```
# wget http://nginx-sticky-module.googlecode.com/files/nginx-sticky-module-1.1.tar.gz
# tar -xzvf nginx-sticky-module-1.1.tar.gz
# wget http://nginx.org/download/nginx-1.0.6.tar.gz
# tar -czvf nginx-1.0.6
# cd nginx-1.0.6
# ./configure --prefix=/usr/local/nginx-1.0.6 --with-http_stub_status_module --with-http_ssl_module
--with-http_realip_module --add-module=../nginx-sticky-module-1.1
# make
# make install
```

3、配置 nginx sticky

nginx 的 upstream 使用 sticky,如下

```
upstream cluster_test {
     sticky;
     server 192.168.100.209:80;
     server 192.168.100.225:80;
}
```

配置虚拟主机（以下有配置的可以忽略掉）

```
server {
        listen        80;
        server_name     test.ttlsa.com;
        index index.jsp;
        access_log /data/logs/nginx/test.ttlsa.com_access.log main;
```

```
        set $proxy_pass cluster_test;
        location /
        {
                proxy_pass http://$proxy_pass;
                include proxy.conf;
                add_header Cache-Control no-store;
        }
}
```

备注：

nginx 和 apache 不同，nginx 每次安装一个新的模块都需要重新编译一次，编译完成之后将 nginx 这一个文件拷贝到 sbin 下面即可.我这边全新安装一次,因为公司在两年前就选择了这个 nginx 版本,也没打算去换,所以大家可以把 nginx 换成自己最合适的一个版本,不用完全跟着文章来安装.

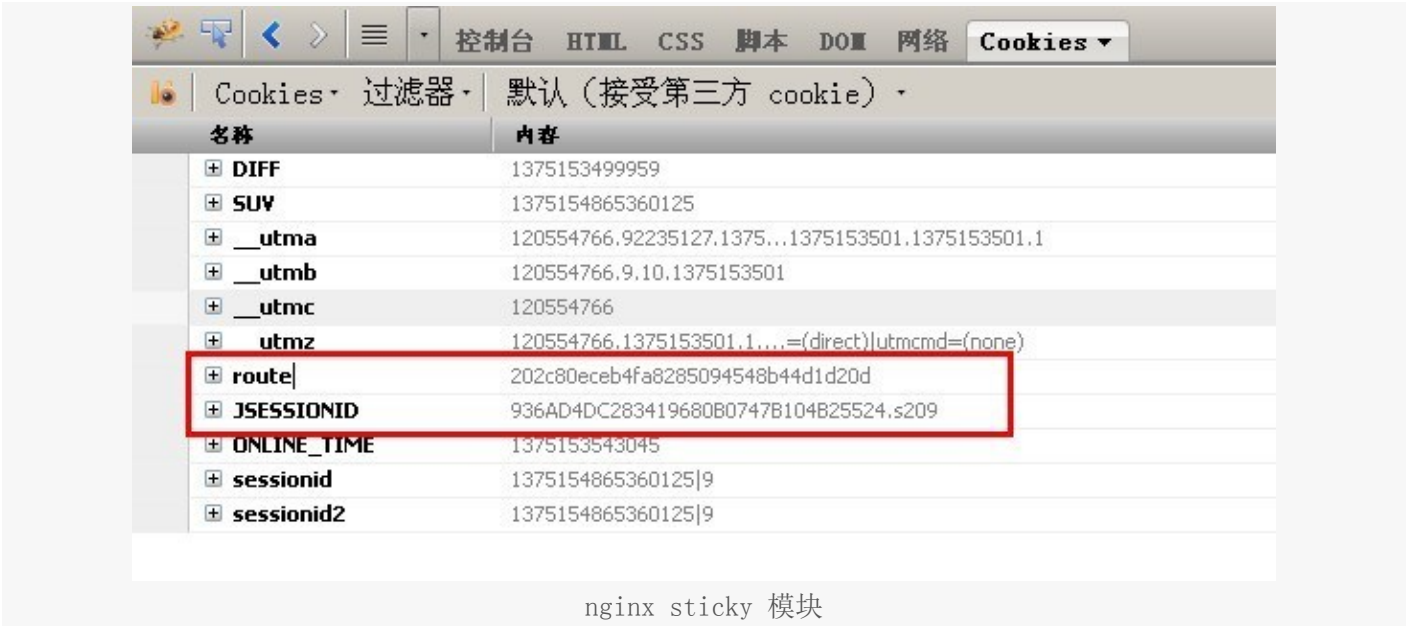
4、重启 nginx

```
/usr/local/nginx-1.0.6/sbin/nginx -t
/usr/local/nginx-1.0.6/sbin/nginx -s reload
```

5、测试 nginx sticky

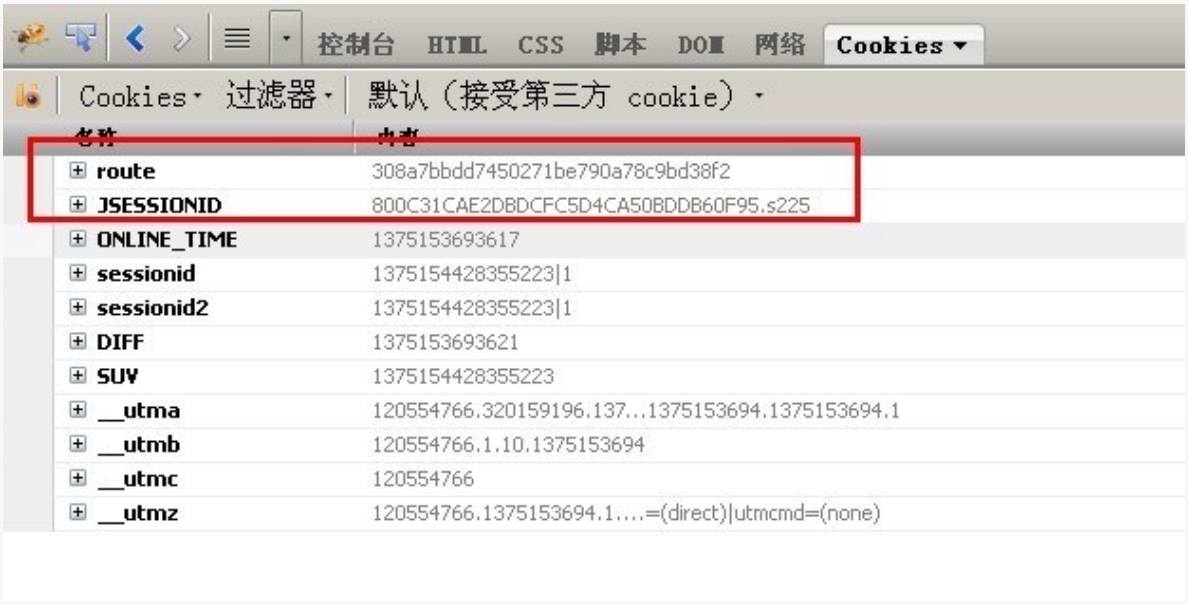
我后端是两台 tomcat 服务器，每台服务器的 JESSIONED 值都有特殊的标志。比如 209 这台是 s209,225 这台是 s225.打开页面,不管怎么刷新 JESSIONED 值都是不变.但是如果开启了 sticky，可以看到 JESSIONED 值不会发生变化.死死的粘滞在其中一台服务器上.测试图如下：

使用 sticky 的情况下，不管怎么刷新都是下面图



nginx sticky 模块

不使用 nginx sticky 模块,多刷几次就变了（有时候刷一次，有时候多刷几次,看概率,不过肯定会变）,如下图



nginx sticky 模块

备注：每台后端真实服务器都会有一个唯一的 route 值,所以不管你真实服务器前端有几个装了 sticky 的 nginx 代理,他都是不会变化的. 这个 cookie 是会话方式的，所以你浏览器关闭了，服务器会给你重新分配一台服务器。

6、nginx sticky 其他语法

```
sticky [name=route] [domain=.foo.bar] [path=/] [expires=1h] [hash=index|md5|sha1] [no_fallback];
```

name：可以为任何的 string 字符,默认是 route

domain：哪些域名下可以使用这个 cookie

path：哪些路径对启用 sticky,例如 path/test,那么只有 test 这个目录才会使用 sticky 做负载均衡

expires：cookie 过期时间，默认浏览器关闭就过期，也就是会话方式。

no_fallbackup：如果设置了这个，cookie 对应的服务器宕机了，那么将会返回 502（bad gateway 或者 proxy error），建议不启用

7、nginx sticky expires 用法

```
upstream cluster_test {
      sticky expires=1h;
      server 192.168.100.209:80;
      server 192.168.100.225:80;
}
```

启用了过期，cookie 如下截图,cookie1 个小时才过期

nginx sticky expire 用法

如下是不启用过期

nginx sticky 不启用 expire

8、nginx sticky 使用注意事项

nginx sticky 模块不能与 ip_hash 同时使用

07.nginx 上传模块—nginx upload module

nginx 上传模块

一. nginx upload module 原理

官方文档： http://www.grid.net.ru/nginx/upload.en.html

Nginx upload module 通过 nginx 服务来接受用户上传的文件，自动解析请求体中存储的所有文件上传到 upload_store 指定的目录下。这些文件信息从原始请求体中分离并根据 nginx.conf 中的配置重新组装好上传参数，交由 upload_pass 指定的段处理，从而允许处理任意上传文件。每个上传文件中的 file 字段值被一系列的 upload_set_form_field 指令值替换。每个上传文件的内容可以从$upload_tmp_path 变量读取，或者可以将文件转移到目的目录下。上传的文件移除可以通过 upload_cleanup 指令控制。如果请求的方法不是 POST，模块将返回 405 错误(405 Not Allowed)，该错误提示可以通过 error_page 指令处理。

具体的过程如下：

1. 用户访问能够选择上传文件的页面
2. 用户提交表单
3. 浏览器把文件和有关文件的信息作为请求的一部分发送给服务器
4. 服务器把文件保存到临时存储目录下 upload_store
5. upload_pass 指定的处理表单提交的 php 页面将文件从 upload_store 拷贝到持久存储位置

二.nginx upload module 配置参数

upload_pass 指明后续处理的 php 地址。文件中的字段将被分离和取代，包含必要的信息处理上传文件。

upload_resumable 是否启动可恢复上传。

upload_store 指定上传文件存放地址(目录)。目录可以散列，在这种情况下，在 nginx 启动前，所有的子目录必须存在。

upload_state_store 指定保存上传文件可恢复上传的文件状态信息目录。目录可以散列，在这种情况下，在 nginx 启动前，所有的子目录必须存在。

upload_store_access 上传文件的访问权限，user:r 是指用户可读

upload_pass_form_field 从表单原样转到后端的参数，可以正则表达式表示。:

$upload_field_name — 原始文件中的字段的名称

upload_pass_form_field “^submit$|^description$”;

意思是把 submit，description 这两个字段也原样通过 upload_pass 传递到后端 php 处理。如果希望把所有的表单字段都传给后端可以用 upload_pass_form_field “^.*$”;

upload_set_form_field 名称和值都可能包含以下特殊变量：

$upload_field_name 表单的 name 值

$upload_content_type 上传文件的类型

$upload_file_name 客户端上传的原始文件名称

$upload_tmp_path 文件上传后保存在服务端的位置

upload_aggregate_form_field 可以多使用的几个变量,文件接收完毕后生成的并传递到后端

$upload_file_md5 文件的 MD5 校验值
$upload_file_md5_uc 大写字母表示的 MD5 校验值
$upload_file_sha1 文件的 SHA1 校验值
$upload_file_sha1_uc 大写字母表示的 SHA1 校验值
$upload_file_crc32 16 进制表示的文件 CRC32 值
$upload_file_size 文件大小
$upload_file_number 请求体中的文件序号
这些字段值是在文件成功上传后计算的。
upload_cleanup 如果出现 400 404 499 500-505 之类的错误，则删除上传的文件
upload_buffer_size 上传缓冲区大小
upload_max_part_header_len 指定头部分最大长度字节。
upload_max_file_size 指定上传文件最大大小，软限制。client_max_body_size 硬限制。
upload_limit_rate 上传限速，如果设置为 0 则表示不限制。
upload_max_output_body_len 超过这个大小，将报 403 错(Request entity too large)。
upload_tame_arrays 指定文件字段名的方括号是否删除
upload_pass_args 是否转发参数。

三. nginx 配置

```
# wget http://www.nginx.org/download/nginx-1.2.2.tar.gz
# wget http://www.grid.net.ru/nginx/download/nginx_upload_module-2.2.0.tar.gz
# tar zxvf nginx_upload_module-2.2.0.tar.gz -c ../software/
# tar zxvf nginx_upload_module-2.2.0.tar.gz -C ../software/
#  ./configure  - prefix=/usr/local/nginx  - add-module=../nginx_upload_module-2.2.0  - with-
http_secure_link_module
# make
# make install
# vi nginx.conf
user www-data;
worker_processes 20;
error_log logs/error.log notice;
working_directory /usr/local/nginx;
events {
worker_connections 1024;
}
http {
include mime.types;
default_type application/octet-stream;
root /www/web/upload;
server {
listen 80;
server_name 192.168.41.129;

error_page 405 =200 @405; //处理 405 错误
location / {
index index.html index.htm index.php;
}
location @405
{
root /www/web/upload;
}
```

```
location ~ \.php$ {
try_files $uri /404.html;
fastcgi_pass 127.0.0.1:9000;
fastcgi_index index.php;
fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
include /etc/nginx/fastcgi_params;
}
client_max_body_size 100m;
# 上传页面提交到这个 location
location /upload {
# 文件上传以后转交给后端的 php 代码处理
upload_pass @test;
# 上传文件的临时存储位置，目录是散列的，应该存在子目录 0 1 2 3 4 5 6 7 8 9
upload_store /www/web/upload/tmp 1;
upload_store_access user:r;
# 设置请求体的字段
upload_set_form_field “${upload_field_name}_name” $upload_file_name;
upload_set_form_field “${upload_field_name}_content_type” $upload_content_type;
upload_set_form_field “${upload_field_name}_path” $upload_tmp_path;
# 指示后端关于上传文件的 md5 值和文件大小
upload_aggregate_form_field “${upload_field_name}_md5″ $upload_file_md5;
upload_aggregate_form_field “${upload_field_name}_size” $upload_file_size;
# 指示原样转到后端的参数，可以用正则表达式表示
upload_pass_form_field “^submit$|^description$”;
upload_pass_args on;
}
# 将请求转到后端的地址处理
location @test {
rewrite ^(.*)$ /test.php last;
}
}
}
```

四. 上传界面

```
# cat /www/web/upload/upload.html
<html>
    <head>
        <title>Test upload</title>
    </head>
  <body>
      <h2>Select files to upload</h2>
    <form enctype=” multipart/form-data” action=” /upload” method=” post” >
      <input type=” file” name=” file1″ ><br>
      <input type=” file” name=” file2″ ><br>
      <input type=” file” name=” file3″ ><br>
      <input type=” file” name=” file4″ ><br>
      <input type=” file” name=” file5″ ><br>
      <input type=” file” name=” file6″ ><br>
      <input type=” submit” name=” submit” value=” Upload” >
      <input type=” hidden” name=” test” value=” value” >
```

```
    </form>
  </body>
</html>
```

五. upload_pass 处理内容

```
# cat test.php //这里只是简单的打印出来，便于先理解上传原理。请对着输出内容理解下 nginx upload module 配置参数。
<?php
    print_r($_POST);
?>
```

对上传文件的处理请参考：http://cn.php.net/manual/en/features.file-upload.php

六. 测试

http://192.168.41.129/upload.html

输出内容如下所示：

```
Array
(
  [file1_name] => Learning  Perl, Sixth Edition.pdf
  [file1_content_type] => application/pdf
  [file1_path] => /www/web/upload/tmp/4/0000000014
  [file1_md5] => 87032cc58109f5c6bb866d2684f9b48c
  [file1_size] => 8927511
  [file2_name] => Programming Perl, 4th Edition.pdf
  [file2_content_type] => application/pdf
  [file2_path] => /www/web/upload/tmp/5/0000000015
  [file2_md5] => 82a52df177a8912c06af276581cfd5e4
  [file2_size] => 21146356
  [submit] => Upload
)
```

注意：需要修改 php.ini 以下参数

file_uploads on 是否允许通过 http 上传

upload_max_filesize 8m 允许上传文件的最大大小

post_max_size 8m 通过表单 POST 给 php 所能接收的最大值

另外 nginx.conf 中设置上传文件大小

upload_max_file_size 软限制

client_max_body_size 硬限制

08. nginx strip 模块删除不必要的空格

在 nginx 官方 wiki 的第三方模块中看到 nginx strip 模块,简单的看下功能，大意就是把网页中的空格，tab，换行删除以用来减少文件的大小，提高用户的访问速度，这个模块配合 gzip 效果更佳.

mod_strip 模块简介：

直接翻译官方的简介，mod_strip 移除 html 文档中没必要的空行（包含空格，tab，换行）. mod_strip 配合 NginxHttpGzipModule 来使用可以更好的减少文件大小以及减少页面下载时间,这个软件目前还在测试阶段，但是我用得非常 OK.mod_strip 速度非常快,而且仅仅需要非常少量的内存.

1. mod_strip 安装：

```
# cd /usr/local/src/
# wget http://wiki.nginx.org/images/6/63/Mod_strip-0.1.tar.gz
# tar -xzvf Mod_strip-0.1.tar.gz
```

```
# cd nginx-1.4.2 //提前解压好的 nginx
# ./configure --prefix=/usr/local/nginx-1.4.2 --add-module=../mod_strip
# make
# make install
```

2. mod_strip 简单用法：

```
location / {
    strip on;
}
```

strip 指令：

语法: strip on|off
默认: off
可用配置段: main, http, server, location
所有响应给用户的 MIME 类型为 text/html 将会使用该模块

3.　mod_strip 测试

nginx 配置

```
location ~* ^/2322(/.*)
{
    strip on;
}
```

3.1 换行：

```
# cat newline.html
line 1
line 2
```

打开内容如下：

```
line 1line
2# 惊呆了，好好的两行怎么变成这样了
```

3.2 tab 测试：

```
# cat tab.html
<head>
<title>test strip</title>
</head>
<body>
<strong>        tab</strong>    <strong>tab2</strong>
</body>
</html>
```

打开后内容如下：

```
<strong>tabtab2</strong>
```

直接上图：

strip 替换 tab

3.3 替换空格

```
# cat space.html
<b> space </b> <b> space2 </b>
```

打开后内容如下：

space space2

看截图：

strip 替换空格

总结：什么玩意儿，这种模块也能上 nginx 官方模块，而且作者也承认了因为技术原因换行存在问题，很想看看作者的网站长什么样，使用 nginx 的 strip 模块还能 work ok. 如果真想优化这方面内容，推荐大家使用 google 的 pagespeed 模块. 如果你是 apache 请看 apache 使用 pagespeed 加速.

09.nginx + ngx_lua 安装测试

nginx　lua 模块淘宝开发的 nginx 第三方模块,它能将 lua 语言嵌入到 nginx 配置中,从而使用 lua 就极大增强了 nginx 的能力.nginx 以高并发而知名,lua 脚本轻便,两者的搭配堪称完美.接下来请看如何安装 nginx + ngx_lua 模块.以及最后来个简单的测试.

如果你是 ubuntu 系统,请看 nginx+lua+redis 构建高并发应用

系统环境：centos/redhat

安装前准备好如下软件包

- nginx　地址：http://www.nginx.org
- luajit 地址：http://luajit.org/download.html
- HttpLuaModule 地址：http://wiki.nginx.org/HttpLuaModule

1. 下载安装 LuaJIT

```
# cd /usr/local/src
# wget http://luajit.org/download/LuaJIT-2.0.2.tar.gz
# tar -xzvf LuaJIT-2.0.2.tar.gz
# cd LuaJIT-2.0.2
# make
```

出现如下内容表示编译成功

```
OK        Successfully built LuaJIT
make[1]: Leaving directory `/usr/local/src/LuaJIT-2.0.2/src'
==== Successfully built LuaJIT 2.0.2 ====
# make install
```

出现如下内容，表示安装成功

```
==== Successfully installed LuaJIT 2.0.2 to /usr/local ====
```

2. 下载准备 nginx lua 模块

```
# cd /usr/local/src
# wget https://github.com/chaoslawful/lua-nginx-module/archive/v0.8.6.tar.gz
# tar -xzvf v0.8.6
```

3. 安装 nginx

3.1 安装

```
# cd /usr/local/src/
# wget http://nginx.org/download/nginx-1.4.2.tar.gz
# tar -xzvf nginx-1.4.2.tar.gz
# cd nginx-1.4.2
//先导入环境变量,告诉 nginx 去哪里找 luajit
# export LUAJIT_LIB=/usr/local/lib
# export LUAJIT_INC=/usr/local/include/luajit-2.0
# ./configure --prefix=/usr/local/nginx-1.4.2 --add-module=../lua-nginx-module-0.8.6
# make -j2
# make install
```

3.1 常见错误

```
# /usr/local/nginx-1.4.2/sbin/nginx -v
./objs/nginx: error while loading shared libraries: libluajit-5.1.so.2: cannot open shared object file: No suchfile or directory
```

解决方法：

```
# ln -s /usr/local/lib/libluajit-5.1.so.2 /lib64/libluajit-5.1.so.2
```

4. nginx lua 配置

nginx 配置文件加入如下配置：

```
location ~* ^/2328(/.*) {
      default_type 'text/plain';
      content_by_lua 'ngx.say("hello, ttlsa lua")';
}
```

5. 启动测试

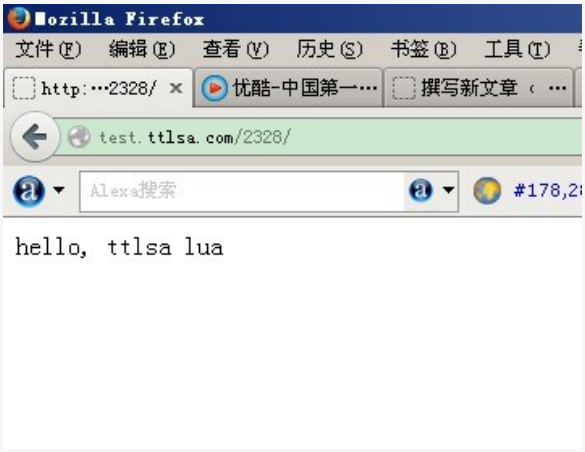
5.1 启动 nginx

```
# /usr/local/nginx-1.4.2/sbin/nginx
```

5.2 访问测试

```
# curl http://test.ttlsa.com/2328/
hello, ttlsa lua //使用 curl 测试
```

nginx lua 测试截图



nginx lua 测试

nginx ngx_lua 的安装到此结束

10.nginx 统计响应的 http 状态码信息(ngx-http-status-code-counter)

1. 介绍

ngx-http-status-code-counter 是一个用来记录 nginx 响应状态码的统计信息，作者将这个模块与 munin 结合来分析网站的 http 状态，我们也可以将这个与 nagios、zabbix 或者其他监控系统想结合，有这个模块运维可以不再使用脚本去分析日志了。

2. 安装

nginx 的安装方式不再多说，请参考运维生存时间早期的《nginx 安装》，我这边使用 nginx-1.4.2 做的测试。作者仅在 0.8.50 版本上使用，模块比较简单，新版本一般都会兼容，不过大家使用之前最好做一个测试。

```
# cd /usr/local/src/
# wget https://github.com/kennon/ngx_http_status_code_counter/archive/master.zip
# unzip master
# cd nginx-1.4.2
```

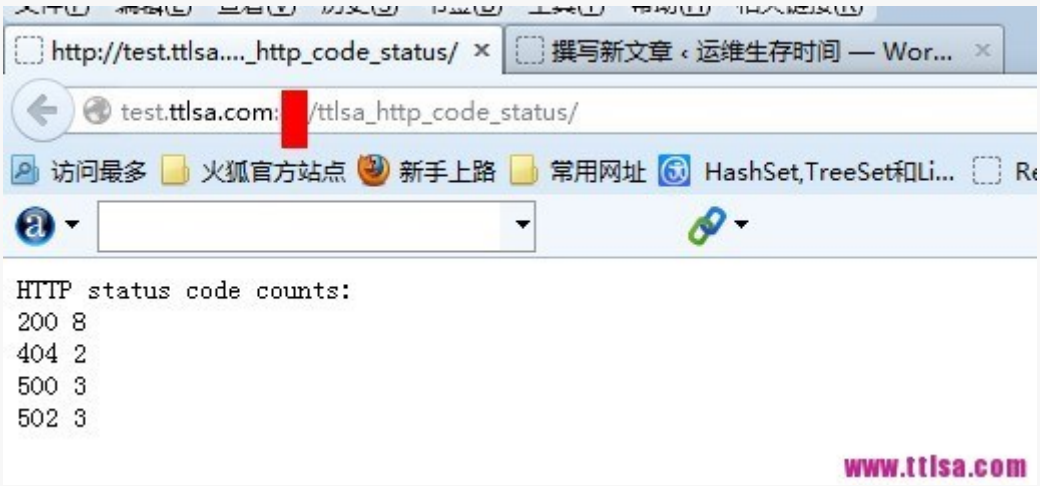
```
    #  ./configure  --prefix=/usr/local/nginx-1.4.2  --add-module=../ngx_http_status_code_counter-master
    # make
    # make install
```

3. 配置 NGINX

```
# for http_code_status
location /ttlsa_http_code_status/
{
        show_status_code_count on;
}

location /ttlsa_http_code_status500/
{
        return 500;
}
location /ttlsa_http_code_status502/
{
        return 502;
}
```

4. 测试

分别访问 http://test.ttlsa.com/ttlsa_http_code_status502/和 http://test.ttlsa.com/ttlsa_http_code_status500 来制造一个 500 和 502 的状态码，以及随意访问一个页面制造 404 响应码，一切都是为了测试。
测试输出内容如下图



ngx_http_status_code_counter

5.　缺点

缺点也很明显，所有的数据都保存在 nginx 内存中，一旦 nginx　reload 或者重开，数据就为空。还有一个缺点便是它统计的数据是持续叠加的，没有时段区分，如果你想统计各个时段的 http　响应代码，你需要定时重启 nginx。

5. 兼容性

兼容 0.8.x，但是 0.7.x 为测试。我当前的版本是 1.4.2 运行 OK

6. 结束语

有这个功能，将 nginx 统计出来的 http 响应码放入监控系统中，便于排除系统故障。后续再贴出整合到监控系统的案例，请继续关注运维生存时间。

11.nginx 流量带宽等请求状态统计(ngx_req_status)

介绍

ngx_req_status 用来展示 nginx 请求状态信息，类似于 apache 的 status，nginx 自带的模块只能显示连接数等等信息，我们并不能知道到底有哪些请求、以及各 url 域名所消耗的带宽是多少。ngx_req_status 提供了这些功能.

功能特性

- 按域名、url、ip 等等统计信息
- 统计总流量
- 统计当前带宽\峰值带宽
- 统计总请求数量

1. 安装

```
# cd /usr/local/src/
 # wget "http://nginx.org/download/nginx-1.4.2.tar.gz"
 # tar -xzvf nginx-1.4.2.tar.gz
 # wget https://github.com/zls0424/ngx_req_status/archive/master.zip -O ngx_req_status.zip
 # unzip ngx_req_status.zip
 # cd nginx-1.4.2/
 # patch -p1 < ../ngx_req_status-master/write_filter.patch
 # ./configure --prefix=/usr/local/nginx-1.4.2 --add-module=../ngx_req_status-master
 # make -j2
 # make install
```

2. 配置

```
http {
    req_status_zone server_name $server_name 256k;
    req_status_zone server_addr $server_addr 256k;
    req_status_zone server_url  $server_name$uri 256k;
    req_status server_name server_addr server_url;
  server {
    server_name test.ttlsa.com;
    location /ttlsa-req-status {
    req_status_show on;
       }
   }
 }
```

4. 指令

req_status_zone
语法: req_status_zone name string size
默认值: None
配置块: http
定义请求状态 ZONE,请求按照 string 分组来排列，例如:
req_status_zone server_url $server_name$uri 256k;
域名+uri 将会形成一条数据，可以看到所有 url 的带宽，流量，访问数

req_status
语法: req_status zone1[zone2]
默认值: None
配置块: http, server, location
在 location 中启用请求状态，你可以指定更多 zones。

req_status_show
语法: req_status_show on
默认值: None
配置块: location
展示数据

5. 测试访问

http://test.ttlsa.com/ttlsa-req-status

如下图

req_status

如上有请求的信息，例如 req-status 这个页面，中流量是 146KB，当前带宽是 0，总请求数量是 47，最大并非连接数是 1.

6. 兼容性

以下版本都兼容
1.4.2 我测试的
1.3.x (last tested: 1.3.5)
1.2.x
1.1.x
1.0.x (last tested: 1.0.2)

12.nginx 实时记录请求状态信息(ngx_realtime_request_module)

关于

ngx_realtime_request 是 nginx 用来统计虚拟主机流量的模块，　首先和大家说下这个模块是基于域名的，将会记录这个域名的请求量、发送字节、返回 http 状态码的数量，特性如下：

- 基于域名记录
- 记录请求数据量
- 记录发送、响应流量
- 记录返回各种 http 状态码统计数据

1. 安装

```
# cd /usr/local/src/
# wget "http://nginx.org/download/nginx-1.4.2.tar.gz"
# tar -xzvf nginx-1.4.2.tar.gz
```

```
    #    wget    https://github.com/magicbear/ngx_realtime_request_module/archive/master.zip    -O
ngx_realtime_request.zip
    # unzip ngx_realtime_request.zip
    # cd nginx-1.4.2/
    # ./configure --prefix=/usr/local/nginx-1.4.2 --add-module=../ngx_realtime_request_module-master
    # make
    # make install
```

2. 指令(directives)

realtime_zonesize
语法: realtime_zonesize size
默认值: 4m
配置块: http
设置 slab 大小

realtime_request
语法: realtime_request [on/off]
默认值: none
配置块: location
开启统计

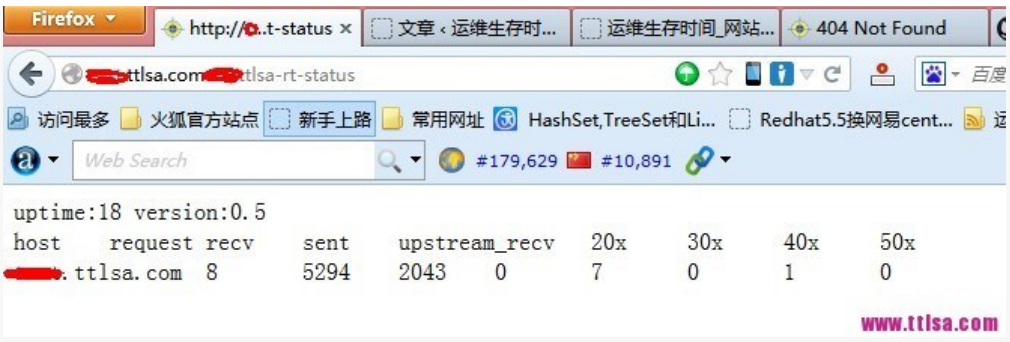
3. 配置实例

```
    http {
        realtime_zonesize  16m;
        server {
            server_name www.ttlsa.com
            location ~ /ttlsa-rt-status {
                realtime_request on;
            }
        }
    }
```

4. 测试

访问 http://www.ttlsa.com 几次，制造几个 404 和 200 等等

查看状态 http://www.ttlsa.com/ttlsa-rt-status 如下图



ngx_realtime_request

上图解释:

uptime: 18 ->nginx 运行了 18 秒
version:0.5 -> 当前插件版本
host: 当前统计的域名,如果这台服务器有多个域名，会显示多行
request: 请求量 8 个
recv: 接收 5294 字节

send：发送 2043 字节
20x：响应了 7 次 20x 的状态码
30x:返回了 0 次 30x
40x：返回了 1 次 40xhttp 状态码（我测试的 404）
50x：返回了 0 次 50x

5. 兼容性

如下版本已做测试
1.3.x (tested with 1.3.6 to 1.3.15).
1.4.2 运维时间用这个版本做了测试

6. 参考文章

项目地址：https://github.com/magicbear/ngx_realtime_request_module
参考文档：http://www.ttlsa.com/nginx/nginx-modules-ngx_realtime_request_module/

13.nginx 获取大文件 MD5 值(nginx 模块 ngx_file_md5)

HTTP 协议新增了 Content-MD5 HTTP 头，但是 nginx 并不支持这个功能，而且官方也明确表示不会增加这项功能，为什么呢？因为每次请求都需要读取整个文件来计算 MD5 值，以性能著称的 nginx 绝对不愿意干出违背软件宗旨的事情。但是有些应用中，需要验证文件的正确性，有些人通过下载当前文件，然后计算 MD5 值来比对当前文件是否正确。不仅仅浪费带宽资源也浪费了大把的时间。有需求就有解决方案，网友开发了 file-md5 模块。

1. 下载模块 file-md5

```
# cd /usr/local/src
# wget https://github.com/cfsego/file-md5/archive/master.zip -O file-md5-master.zip
# unzip file-md5-master.zip
```

2. 安装模块 file-md5

```
# wget http://nginx.org/download/nginx-1.4.2.tar.gz
# tar -xzf nginx-1.4.2.tar.gz
# cd nginx-1.4.2
# ./configure --prefix=/usr/local/nginx-1.4.2  --add-module=../file-md5-master
# make
# make isntall
```

如果你已经安装了 nginx，仅需要增加 file-md5 模块即可，具体参考《nginx 如何安装第三方模块》

3. 配置 file-md5

3.1 MD5 追加到 http 响应头中

```
server {
    listen       80;
    server_name  test.ttlsa.com;
    root /data/site/test.ttlsa.com;
    # for add content-md5 to http header
    location ~ /download
    {
            add_header    Content-MD5    $file_md5;
    }
}
```

所有请求 download 的请求，都会在响应 http 头部增加 Content-MD5，值为这个文件的 MD5，看如下测试：

```
# curl -I test.ttlsa.com/download/1.exe
HTTP/1.1 200 OK
Server: nginx
Date: Wed, 26 Feb 2014 03:00:05 GMT
Content-Type: application/octet-stream
Content-Length: 1535488
Last-Modified: Mon, 24 Feb 2014 10:08:10 GMT
Connection: keep-alive
ETag: "530b1a0a-176e00"
Content-MD5: 6adda4a06dbad3ac9b53a08f4ff9c4f8
Accept-Ranges: bytes
```

大家可以看到 Content-MD5: 6adda4a06dbad3ac9b53a08f4ff9c4f8，这个就是 1.exe 文件的 MD5 值.

3.2 直接响应 MD5 值到内容中

```
server {
    listen       80;
    server_name  test.ttlsa.com;
    root /data/site/test.ttlsa.com;

    # for add content-md5 to http header
    location ~ /download
    {
        if ( $arg_md5 ~* "true" ){
            echo $file_md5;
        }
    }
}
```

这边直接使用 echo 输出 MD5 值（echo 模块需要额外安装），只需在下载的文件后面加上参数&md5=true 即可得到 MD5 值，使用过程中，参数可以随心定义。下面来测试一下。

```
# curl test.ttlsa.com/download/1.exe?md5=true
6adda4a06dbad3ac9b53a08f4ff9c4f8
```

直接得到 md5 值，与第一种方法得到同样的 MD5。

4. 最后

使用 nginx 模块也是一种方法，这种方法有个不足支持，每个请求都需要从新计算一次 MD5 值。想减小他的压力，可以在 nginx 加缓存，或者借用 memcache 以及使用 perl 或者 lua 等模块，希望大家继续支持运维生存时间。

项目地址：https://github.com/cfsego/file-md5
项目文档：https://github.com/cfsego/file-md5/blob/master/README

14.nginx 不记录特定日志(access_log_bypass_if)

apache 可以使用 CustomLog [env=XXX]指定排除哪些日志不记录，nginx 自身没有这个功能，但是在官方的第三方模块找到了 ngx_log_if_module，它实现了类似 env 的功能.如果你想了解 nginx 更多 nginx 日志格式的资料，请参考咱们 ttlsa《nginx 日志配置》

1. nginx 第三方模块安装

参考运维生存时间之前的文章《如何安装 nginx 第三方模块》

2. 指令

语法: access_log_bypass_if (condition) [and]
默认值: -
Scope: main/srv/loc

“access_log_bypass_if” 定义的规则如果为真，那么相应的日志不会写入 access log 中.
access_log_bypass_if 你可以把它当做 if 来看待，access_log_bypass_if 可以使用 and 逻辑运算.如果当前 if 有 and，那么他和下一个 if 共同作用.

3. access_log_bypass_if 配置

如下是多个表达式的例子

```
server {
 access_log_bypass_if ($status = 400);
 access_log_bypass_if ($host ~* 'ttlsa.com');
 access_log_bypass_if ($uri = 'status.nginx') and;
 access_log_bypass_if ($status = 200);
 }
```

上面一共有 4 个表达式，前面两条分别独立，组后两条是 and 组合的。也就是说状态为 400 或者 host 为 ttlsa.com 或者（uri 是 status.nginx 并且响应状态为 200）的请求都不会记录到访问日志中.

当然，access_log_bypass_if 会在多个地方出现，官方文档叫父配置块和子配置块，默认情况下子配置块会覆盖父配置块，并且不会继承父配置块的配置.
如下例子

```
server {
   access_log_bypass_if ($status = 400);
  location / {
     access_log_bypass_if ($host ~* 'ttlsa.com');
   }
 }
```

我们可以发现$status = 400 根本就没有效果，因为他被 location /里面的 access_log_bypass_if 覆盖了

4. 参考地址

官方地址：https://github.com/cfsego/ngx_log_if/

15.nginx 快速绘制圆形图（ngx_http_circle_gif_module 模块）

nginx 官网上有各式各样的第三方模块,今天来介绍一款叫做 ngx_http_circle_gif_module。从字面意思来看，他是一个声称圆形图片的模块，实际上也是如此。此模块生成图片比直接在硬盘上读取要快很多，并且可以不用劳烦美工去设计。或者说不用麻烦同学用 windows 下的画图去话一个圆.功能很简单，指令也很简单。

安装模块参数

--add-module=path/to/circle_gif/directory
具体方法就不在讲述了，可以参考运维生存时间的《如何安装 nginx 第三方模块》

circle_gif 指令
circle_gif
语法: circle_gif
默认值: n/a
配置段: location

circle_gif_min_radius
语法: circle_min_radius
默认值: 10
配置段: location
圆形的最小半径, 单位为像素

circle_gif_max_radius
语法: circle_max_radius
默认值: 20
配置段: location
圆形的最大半径, 单位为像素

circle_gif_step_radius
语法: circle_step_radius
默认值: 2
配置段: location
The “step” in between generated circle images

circle_gif 配置

```
location /ttlsa_circles {
circle_gif;
}
```

circle_gif 用法

<背景颜色>/<前景色>/<半径>.gif

例子

http://test.ttlsa.com/ttlsa_circles/ffffff/000000/20.gif

ffffff：背景色是白色
000000：前景色是黑色
20：圆形半径为 20

circle_gif 效果图

circle_gif

参考地址

nginx_circle_gif 下载：https://github.com/evanmiller/nginx_circle_gif/

nginx_circle_gif 文档：http://wiki.nginx.org/HttpCircleGifModule

16.nginx 实现大小写字母转换（ngx_http_lower_upper_case 模块）

各种程序或脚本都有实现大小写字母互转的功能，今天讲讲 ngx_http_lower_upper_case，功能很简单，至于可以用在什么环境大家可以根据自己的情况.多一种模块多一种解决方案.本模块将字符串转换为大小写然后赋值给变量. 用 ttlsa 群组的话来说”存在即合理”，软件存在性总有他存在的道理。

1. 安装 nginx 模块

--add-module=path/to/circle_gif/directory
具体方法就不再讲述了，可以参考运维生存时间的<如何安装 nginx 第三方模块>

2.upper/lower 指令

upper
语法：upper $var string
配置段：location
小些转大写

lower
语法：lower $var string
配置段：location
大写转小写

3. nginx 配置

```
location /ttlsa_upper_lower {
upper $var1 “Hello,ttlsa.com”;
lower $var2 “HELLO,TTLSA.COM”;
echo $var1;
echo $var2;
}
```

4. 测试

```
# curl http://test.ttlsa.com/ttlsa_upper_lower/
 HELLO,TTLSA.COM
 hello,ttlsa.com
```

5. 参考地址

nginx 模块地址：https://github.com/replay/ngx_http_lower_upper_case/archive/master.zip
ngx_http_lower_upper_case 地址：https://github.com/replay/ngx_http_lower_upper_case

17.nginx 防止高负载的解决方案(sysgurad 模块)

如果 nginx 被攻击或者访问量突然变大，nginx 会因为负载变高或者内存不够用导致服务器宕机，最终导致站点无法访问。今天要谈到的解决方法来自淘宝开发的模块 nginx-http-sysguard，主要用于当负载和内存达到一定的阀值之时，会执行相应的动作，比如直接返回 503,504 或者其他的.一直等到内存或者负载回到阀值的范围内，站点恢复可用。简单的说，这几个模块是让 nginx 有个缓冲时间，缓缓.

1. 安装 nginx sysguard 模块

1.1 下载文件

```
# wget http://nginx.org/download/nginx-1.4.2.tar.gz
# wget https://github.com/alibaba/nginx-http-sysguard/archive/master.zip \
-O nginx-http-sysguard-master.zip
# unzip nginx-http-sysguard-master.zip
# tar -xzvf nginx-1.4.2.tar.gz
```

1.2 打 sysgrard 补丁

这边没找到 nginx-1.4.2 对应的补丁，只有 1.2.9 和 1.3.9 的，索性试试 1.3.9 的吧，应该差不多.

```
# cd nginx-1.4.2
# patch -p1 < ../nginx-http-sysguard-master/nginx_sysguard_1.3.9.patch
```

1.3 安装 nginx

```
# ./configure --prefix=/usr/local/nginx-1.4.2 \
--with-http_stub_status_module --add-module=../nginx-http-sysguard
# make
# make install
```

2. sysguard 指令

语法: sysguard [on | off]
默认值: sysguard off
配置段: http, server, location
开关模块

语法: sysguard_load load=number [action=/url]
默认值: none
配置段: http, server, location
指定负载阀值,当系统的负载超过这个值，所有的请求都会被重定向到 action 定义的 uri 请求中.如果没有定义 URL action 没有定义，那么服务器直接返回 503

语法: sysguard_mem swapratio=ratio% [action=/url]
默认值: none

配置段: http, server, location
定义交换分区使用的阀值，如果交换分区使用超过这个阀值，那么后续的请求全部被重定向到 action 定义的 uri 请求中.如果没有定义 URL action 没有定义，那么服务器直接返回 503

语法: sysguard_interval time
默认值: sysguard_interval 1s
配置段: http, server, location
定义系统信息更新的频率,默认 1 秒.

语法: sysguard_log_level info | notice | warn | error
默认值: sysguard_log_level error
配置段: http, server, location
定义 sysguard 的日志级别

3. sysguard 使用实例

3.1 nginx 配置

```
server {
    listen       80;
    server_name  www.ttlsa.com www.heytool.com;
    access_log  /data/logs/nginx/www.ttlsa.com.access.log  main;
    index index.html index.php index.html;
    root /data/site/www.ttlsa.com;
    sysguard on;
    # 为了方便测试，load 阀值为 0.01，平时大家一般都在 5 或 10+
    sysguard_load load=0.01 action=/loadlimit;
    sysguard_mem swapratio=20% action=/swaplimit;
    location / {
    }
    location /loadlimit {
        return 503;
    }
    location /swaplimit {
        return 503;
    }
}
```

3.2 测试
负载 OK 的情况下,访问 nginx

```
# uptime
 16:23:37 up 6 days,  8:04,  2 users,  load average: 0.00, 0.01, 0.05
# curl -I www.ttlsa.com
HTTP/1.1 403 Forbidden
Server: nginx
Date: Thu, 03 Oct 2013 16:27:13 GMT
Content-Type: text/html
Content-Length: 162
Connection: keep-alive
```

因为站点下没有文件，所以返回了 403，实际上没关系.

负载超过阀值的情况下，访问 nginx

```
# uptime
```

```
 16:25:59 up 6 days,  8:06,  2 users,  load average: 0.05, 0.04, 0.05
# curl -I www.ttlsa.com
HTTP/1.1 503 Service Temporarily Unavailable
Server: nginx
Date: Thu, 03 Oct 2013 16:26:19 GMT
Content-Type: text/html
Content-Length: 206
Connection: keep-alive
```

swap 超过阀值的功能我就不再测试了。大家回家可以自己动手测试一下.

结束语

在 nginx 是 realserver 的情况下，个人也比较推荐使用这种方法，如果服务器负载一旦爬高，一般要比较长的时间才能恢复到正常水平，在采用这个插件的情况下，负载达到阀值，nginx 返回 503，前段使用故障转移将请求发往其他服务器，这台服务器在无访问的情况下，便能很快的恢复到正常水平，并且能够立即投入工作。超过阀值的服务器处理请求速度也会大打折扣，使用这个模块，巧妙的将请求发送到了更快速的服务器上，在一定程度上避免了访问速度慢的问题. 前面说的是在集群环境下，在单点环境下，用不用大家斟酌一下。

参考文章：
nginx-http-sysguard:https://github.com/alibaba/nginx-http-sysguard
TCP Proxy:https://github.com/yaoweibin/nginx_tcp_proxy_module (相同功能软件)

18.nginx js、css 多个请求合并为一个请求(concat 模块)

mod_concat 模块由淘宝开发，目前已经包含在 tengine 中，并且淘宝已经在使用这个 nginx 模块。不过塔暂时没有包含在 nginx 中。这个模块类似于 apache 中的 modconcat。如果需要使用它，需要使用两个”?”问号.

来个范例：

http://example.com/??style1.css,style2.css,foo/style3.css

以上将原先 3 个请求合并为 1 个请求

如果你担心文件被用户的浏览器缓存而没有及时更新，你依旧可以带上一个版本号的参数，如下：

http://example.com/??style1.css,style2.css,foo/style3.css?v=102234

1.安装 nginx concat

```
# cd /usr/local/src/
# wget http://nginx.org/download/nginx-1.4.2.tar.gz
#   wget   https://github.com/alibaba/nginx-http-concat/archive/master.zip   -O   nginx-http-concat-master.zip
# unzip nginx-http-concat-master.zip
# tar -xzvf nginx-1.4.2.tar.gz
# cd nginx-1.4.2
# ./configure --prefix=/usr/local/nginx-1.4.2 --with-http_stub_status_module  \
--add-module=../nginx-http-concat-master
# make
# make install
```

2. 指令 directives

concat on | off
default: concat off
context: http, server, location
开启火关闭 concat

concat_types MIME types
default: concat_types: text/css application/x-javascript
context: http, server, location
Defines the MIME types which can be concatenated in a given context.
在给定的配置段中可以被合并的 MIME 文件类型.

concat_unique on | off
default: concat_unique on
context: http, server, location
是否只允许同类型文件（相同 MIME 文件）合并。例如，设置为 off，那么 js 和 css 文件可以合并。默认情况下，这个值是 on，意味着只有相同的类型文件才能合并。

如果希望 js 和 css 能够合并为一个请求，那么你必须设置 concat_unique off，其他类型文件也可以用同样的方式合并. 如下为 OFF 才可以的请求：

http://example.com/static/??foo.css,bar/foobaz.js

concat_max_files numberp
default: concat_max_files 10
context: http, server, location
定义一个给定的配置段里面允许合并文件的数量，默认最多 10 个. 不过一定要注意，uri 不要超过系统的规定的 page size，在 linux 里面执行 getconf PAGESIZE 可以获取系统的限制. 通常限制是 4096 字节。

concat_delimiter: string
default: NONE
context: http, server, locatione
定义文件分隔符

concat_ignore_file_error: on | off
default: off
context: http, server, location
是否忽略文件请求错误，例如 404 和 403 等

3. 配置 nginx

```
server {

    listen       80;
    server_name  www.ttlsa.com;

    root /data/site/www.ttlsa.com;
    location /static/ {
        concat on;
        concat_max_files 20;
        concat_unique off;
    }
}
```

4. 测试 nginx concat

我的站点有调用到 static/ttlsa_concat.css 和 static/ttlsa_concat.js 两个文件，为了提高站点访问速度，我决定使用 nginx 的 concat 模块将两个请求合并为一个。
合并前
www.ttlsa.com/static/css/ttsa_concat.css
www.ttlsa.com/static/js/ttsa_concat.js

合并后

http://www.ttlsa.com/static??js/ttlsa_concat.js,css/ttlsa_concat.css?ver=123

测试之前，准备一下文件.

```
# cd /data/site/www.ttlsa.com/static
# cat js/ttlsa_concat.js
// this is js file ttlsa_concat.js
# cat js/a.js
// this is js file a.js
# cat css/a.css
/** this is css a.css **/
# cat css/ttlsa_concat.css
/** this is css ttlsa_concat.css **/
```

4.1 两个 css 合并

```
# curl http://www.ttlsa.com/static/??css/ttlsa_concat.css,css/a.css
/** this is css ttlsa_concat.css **/
/** this is css a.css *
```

4.2 两个 js 合并

```
# curl http://www.ttlsa.com/static/??js/ttlsa_concat.js,js/a.js
// this is js file ttlsa_concat.js
// this is js file a.js
```

4.3 js 与 css 合并

```
# curl http://www.ttlsa.com/static/??js/ttlsa_concat.js,css/ttlsa_concat.css
// this is js file ttlsa_concat.js
/** this is css ttlsa_concat.css **/
```

4.4 带版本号参数

```
# curl http://www.ttlsa.com/static/??css/ttlsa_concat.css,css/a.css?123
/** this is css ttlsa_concat.css **/
/** this is css a.css *
```

以上仅仅用了两个文件来测试，在具体应用中，大家可以使用多个，具体几个由你的 nginx 配置来控制. 在具体的环境中，都是以下方式来调用，以下方法摘自官方文档.
js:

```
<script src="??bar1.js,bar22.css,subdir/bar3.js?v=3245" />
```

以上意思是将 ba1.Js，bar22.css 和 subdir/bar3.js 这三个请求合并为一个，并且版本号为 3245.

css:

```
<link rel="stylesheet"  href="??foo1.css,foo2.css,subdir/foo3.css?v=2345"  />
```

这边也是一个道理,只不过只包含 css.

5. 结束语

减少 web 请求在一定程度上能减少 web 服务器的压力，简单的使用 nginx concat 模块便可以实现这个功能，不过需要前端设计师来使用。如果想减少 web 请求，又不想让前端设计师来插手的话，大家可以参考下 google 出的 pagespeed 模块

6. 参考文章

官方：http://wiki.nginx.org/HttpConcatModule

19.CDN 下 nginx 获取用户真实 IP 地址

随着 nginx 的迅速崛起，越来越多公司将 apache 更换成 nginx. 同时也越来越多人使用 nginx 作为负载均衡，并且代理前面可能还加上了 CDN 加速，但是随之也遇到一个问题：nginx 如何获取用户的真实 IP 地址,如果后端是 apache,请跳转到<apache 获取用户真实 IP 地址>，如果是后端真实服务器是 nginx，那么继续往下看。

实例环境：
用户 IP 120.22.11.11
CDN 前端 61.22.22.22
CDN 中转 121.207.33.33
公司 NGINX 前端代理 192.168.50.121（外网 121.207.231.22）

1、使用 CDN 自定义 IP 头来获取

假如说你的 CDN 厂商使用 nginx,那么在 nginx 上将$remote_addr 赋值给你指定的头,方法如下:

```
proxy_set_header remote-user-ip $remote_addr;
```

//如上,后端将会收到 remote_user_ip 的 http 头，有些人可能会挑错了，说我设置的头不是 remote-user-ip 吗,怎么写成了 remote_user_ip,是不是作者写错了.请参考文章：<nginx 反向代理 proxy_set_header 自定义 header 头无效>

后端 PHP 代码 getRemoteUserIP.php

```
<?php
    $ip = getenv("HTTP_REMOTE_USER_IP");
    echo $ip;
?>
```

访问 getRemoteUserIP.php,结果如下：

120.22.11.11 //取到了真实的用户 IP,如果 CDN 能给定义这个头的话,那这个方法最佳

2、通过 HTTP_X_FORWARDED_FOR 获取 IP 地址

一般情况下 CDN 服务器都会传送 HTTP_X_FORWARDED_FOR 头,这是一个 ip 串,后端的真实服务器获取 HTTP_X_FORWARDED_FOR 头,截取字符串第一个不为 unkown 的 IP 作为用户真实 IP 地址，例如：

120.22.11.11,61.22.22.22,121.207.33.33,192.168.50.121（用户 IP,CDN 前端 IP,CDN 中转,公司 NGINX 代理）

getFor.php

```
<?php
    $ip = getenv("HTTP_X_FORWARDED_FOR");
    echo $ip;
?>
```

访问 getFor.php 结果如下：

120.22.11.11,61.22.22.22,121.207.33.33,192.168.50.121
如果你是 php 程序员,你获取第一个不为 unknow 的 ip 地址,这边就是 120.22.11.11.

3.使用 nginx 自带模块 realip 获取用户 IP 地址

安装 nginx 之时加上 realip 模块，我的参数如下：

```
./configure --prefix=/usr/local/nginx-1.4.1 --with-http_realip_module
```

真实服务器 nginx 配置

```
server {
    listen       80;
    server_name  www.ttlsa.com;
    access_log  /data/logs/nginx/www.ttlsa.com.access.log  main;

    index index.php index.html index.html;
    root /data/site/www.ttlsa.com;

    location /
    {
            root /data/site/www.ttlsa.com;
    }
    location = /getRealip.php
    {
            set_real_ip_from  192.168.50.0/24;
            set_real_ip_from  61.22.22.22;
            set_real_ip_from  121.207.33.33;
            set_real_ip_from 127.0.0.1;
            real_ip_header    X-Forwarded-For;
            real_ip_recursive on;
            fastcgi_pass  unix:/var/run/phpfpm.sock;
            fastcgi_index index.php;
            include fastcgi.conf;
    }
}
```

getRealip.php 内容

```
<?php
    $ip =  $_SERVER['REMOTE_ADDR'];
    echo $ip;
?>
```

访问 www.ttlsa.com/getRealip.php,返回：

```
120.22.11.11
```

如果注释 real_ip_recursive on 或者 real_ip_recursive off

访问 www.ttlsa.com/getRealip.php,返回：

```
121.207.33.33
```

很不幸,获取到了中继的 IP,real_ip_recursive 的效果看明白了吧.

set_real_ip_from：真实服务器上一级代理的 IP 地址或者 IP 段,可以写多行

real_ip_header：从哪个 header 头检索出要的 IP 地址

real_ip_recursive：递归排除 IP 地址,ip 串从右到左开始排除 set_real_ip_from 里面出现的 IP,如果出现了未出现这些 ip 段的 IP，那么这个 IP 将被认为是用户的 IP。例如我这边的例子，真实服务器获取到的 IP 地址串如下：

120.22.11.11,61.22.22.22,121.207.33.33,192.168.50.121

在 real_ip_recursive on 的情况下

61.22.22.22,121.207.33.33,192.168.50.121 都出现在 set_real_ip_from 中,仅仅 120.22.11.11 没出现,那么他就被认为是用户的 ip 地址，并且赋值到 remote_addr 变量

在 real_ip_recursive off 或者不设置的情况下
192.168.50.121 出现在 set_real_ip_from 中,排除掉，接下来的 ip 地址便认为是用户的 ip 地址

如果仅仅如下配置：

```
set_real_ip_from    192.168.50.0/24;
set_real_ip_from 127.0.0.1;
real_ip_header     X-Forwarded-For;
real_ip_recursive on;
```

访问结果如下：

121.207.33.33

4、三种在 CDN 环境下获取用户 IP 方法总结

4.1 CDN 自定义 header 头
优点：获取到最真实的用户 IP 地址,用户绝对不可能伪装 IP
缺点：需要 CDN 厂商提供

4.2 获取 forwarded-for 头
优点：可以获取到用户的 IP 地址
缺点：程序需要改动,以及用户 IP 有可能是伪装的

4.3 使用 realip 获取
优点：程序不需要改动，直接使用 remote_addr 即可获取 IP 地址
缺点：ip 地址有可能被伪装，而且需要知道所有 CDN 节点的 ip 地址或者 ip 段

20.nginx 实时生成缩略图到硬盘上

现在随着各终端的出现(手机，ipad 等平板),以及各种终端的手机分辨率和尺寸都不同，现在手机用户流量都是宝,网上出现了各种各样的生成缩略图功能的架构，有使用 php 实时生成缩略图的，也有用 nginx + lua 实现的，上节我也讲到了使用 nginx 生成缩略图，但是用户每次访问都需要生成一次，会给 cpu 和硬盘带来比较大的压力，今天带来了另外一种方式，这次使用 nginx 将原图生成缩略图到硬盘上.看我的配置

1. 首先建好 cache 目录

```
# mkdir /data/site_cache/
```

2. 修改 nginx 配置

```
        location ~* ^/resize {
                root /data/site_cache/$server_name;
                set $width 150;
                set $height 100;
                set $dimens "";

                if ($uri ~* "^/resize_(\d+)x(\d+)/(.*)" ) {
                        set $width $1;
                        set $height $2;
                        set $image_path $3;
                        set $demins "_$1x$2";
                }
```

```
        if ($uri ~* "^/resize/(.*)" ) {
                set $image_path $1;
        }

        set $image_uri image_resize/$image_path?width=$width&height=$height;

        if (!-f $request_filename) {
                proxy_pass http://127.0.0.1/$image_uri;
                break;
        }
        proxy_store /data/site_cache/$server_name/resize$demins/$image_path;
        proxy_store_access user:rw group:rw all:r;
        proxy_set_header Host $host;
        expires      30d;
        access_log off;
    }
    location /image_resize {
        alias /data/site/$server_name/;
        image_filter resize $arg_width $arg_height;
        image_filter_jpeg_quality 75;
        access_log off;
    }
```

生成缩略图流程如下：

1、原图在 www.ttlsa.com/image/1.jpg。我需要一份 100×100 的缩略图。

2、请求 www.ttlsa.com/resize_100x100/image/1.jpg.

3、这个请求进入了 location ~* ^/resize,接着判断 image_path 这个目录下是否存在这张图片,如果存在直接放回给用户,

4、不存在那么跳转到 http://www.ttlsa.com/image_resize/image/1.jpg?width=100&height=100;

5、location /image_resize 根据传入的 width 和 height 执行缩略功能,并且设置图像质量为 75

6、接着生成文件到/data/site_cache/www.ttlsa.com/resize_100x100/image/1.jpg

7、并且返回图片给用户

8、nginx 生成缩略图到硬盘上的功能到这里就结束了

21.perl + fastcgi + nginx 搭建

nginx + fastcgi 是 php 下最流行的一套环境了，那 perl 会不会也有 fastcgi 呢，当然有，今天来搭建下 nginx 下 perl 的 fastcgi.性能方面也不亚于 php，但是现在 web 程序 php 的流行程度 perl 无法比拟了，性能再好也枉然，但是部分小功能可以考虑使用 perl 的 fastcgi 来搞定.进入正题.

1. 准备软件环境：

nginx：http://www.nginx.org

perl：系统自带

fastcgi：http://www.cpan.org/modules/by-module/FCGI/

1.1 nginx 安装

nginx 安装过无数次,这边不在重复安装过程,如果你还没有安装 nginx 并且不知道怎么安装 nginx，那么请先参考之前的文章《nginx 安装配置》

1.2 perl 安装
一般 linux 都有自带 perl，可以不用安装，如果确实没有，请执行：

```
# yum install perl
```

1.3 perl-fastcgi 安装

```
# cd /usr/local/src
# wget http://www.cpan.org/modules/by-module/FCGI/FCGI-0.74.tar.gz
# tar -xzvf FCGI-0.74.tar.gz
# cd FCGI-0.74
# perl Makefile.PL
# make
# make install
```

2. nginx 虚拟主机配置

```
server {
        listen       80;
        server_name  test.ttlsa.com;
        #access_log  /data/logs/nginx/test.ttlsa.com.access.log  main;
        index index.html index.php index.html;
        root /data/site/test.ttlsa.com;
        location /
        {
        }
        location ~ \.pl$
        {
            include fastcgi_params;
            fastcgi_pass  127.0.0.1:8999;
            #fastcgi_pass  unix:/var/run/ttlsa.com.perl.sock;
            fastcgi_index index.pl;
        }
}
```

如果想把 tcp/ip 方式改为 socket 方式，可以修改 fastcgi-wrapper.pl.

```
$socket = FCGI::OpenSocket( "127.0.0.1:8999", 10 );#use IP sockets
改为
$socket = FCGI::OpenSocket( "/var/run/ttlsa.com.perl.sock", 10 );#use IP sockets
```

3. 配置脚本

3.1 fastcgi 监听脚本
文件路径：/usr/bin/fastcgi-wrapper.pl

```
#!/usr/bin/perl

use FCGI;
use Socket;
use POSIX qw(setsid);
require 'syscall.ph';
&daemonize;
#this keeps the program alive or something after exec'ing perl scripts
```

```
END() { } BEGIN() { }
*CORE::GLOBAL::exit = sub { die "fakeexit\nrc=".shift()."\n"; };
eval q{exit};
if ($@) {
    exit unless $@ =~ /^fakeexit/;
};
&main;
sub daemonize() {
    chdir '/'                 or die "Can't chdir to /: $!";
    defined(my $pid = fork)   or die "Can't fork: $!";
    exit if $pid;
    setsid                    or die "Can't start a new session: $!";
    umask 0;
}
sub main {
        $socket = FCGI::OpenSocket( "127.0.0.1:8999", 10 );#use IP sockets
        $request = FCGI::Request( \*STDIN, \*STDOUT, \*STDERR, \%req_params, $socket );
        if ($request) { request_loop()};
            FCGI::CloseSocket( $socket );
}
sub request_loop {
        while( $request->Accept() >= 0 ) {
           #processing any STDIN input from WebServer (for CGI-POST actions)
           $stdin_passthrough ='';
           $req_len = 0 + $req_params{'CONTENT_LENGTH'};
           if (($req_params{'REQUEST_METHOD'} eq 'POST') && ($req_len != 0) ){
                my $bytes_read = 0;
                while ($bytes_read < $req_len) {
                        my $data = '';
                        my $bytes = read(STDIN, $data, ($req_len - $bytes_read));
                        last if ($bytes == 0 || !defined($bytes));
                        $stdin_passthrough .= $data;
                        $bytes_read += $bytes;
                }
            }
            #running the cgi app
            if ( (-x $req_params{SCRIPT_FILENAME}) && #can I execute this?
                 (-s $req_params{SCRIPT_FILENAME}) && #Is this file empty?
                 (-r $req_params{SCRIPT_FILENAME})    #can I read this file?
            ){
        pipe(CHILD_RD, PARENT_WR);
        my $pid = open(KID_TO_READ, "-|");
        unless(defined($pid)) {
            print("Content-type: text/plain\r\n\r\n");
                        print "Error: CGI app returned no output - ";
                        print "Executing $req_params{SCRIPT_FILENAME} failed !\n";
            next;
        }
        if ($pid > 0) {
```

```
            close(CHILD_RD);
            print PARENT_WR $stdin_passthrough;
            close(PARENT_WR);
            while(my $s = <KID_TO_READ>) { print $s; }
            close KID_TO_READ;
            waitpid($pid, 0);
        } else {
                    foreach $key ( keys %req_params){
                       $ENV{$key} = $req_params{$key};
                    }
                    # cd to the script's local directory
                    if ($req_params{SCRIPT_FILENAME} =~ /^(.*)\/[^\/]+$/) {
                            chdir $1;
                    }
            close(PARENT_WR);
            close(STDIN);
            #fcntl(CHILD_RD, F_DUPFD, 0);
            syscall(&SYS_dup2, fileno(CHILD_RD), 0);
            #open(STDIN, "<&CHILD_RD");
            exec($req_params{SCRIPT_FILENAME});
            die("exec failed");
        }
            }
            else {
                print("Content-type: text/plain\r\n\r\n");
                print "Error: No such CGI app - $req_params{SCRIPT_FILENAME} may not ";
                print "exist or is not executable by this process.\n";
            }
        }
}
```

3.2 fastcgi 自启动服务脚本：

文件路径：/etc/rc.d/init.d/perl-fastcgi

```
#!/bin/sh
#
# nginx - this script starts and stops the nginx daemon
#
# chkconfig: - 85 15
# description: Nginx is an HTTP(S) server, HTTP(S) reverse \
# proxy and IMAP/POP3 proxy server
# processname: nginx
# config: /opt/nginx/conf/nginx.conf
# pidfile: /opt/nginx/logs/nginx.pid

# Source function library.
. /etc/rc.d/init.d/functions
```

```
# Source networking configuration.
. /etc/sysconfig/network

# Check that networking is up.
[ "$NETWORKING" = "no" ] && exit 0

perlfastcgi="/usr/bin/fastcgi-wrapper.pl"
prog=$(basename perl)

lockfile=/var/lock/subsys/perl-fastcgi

start() {
    [ -x $perlfastcgi ] || exit 5
    echo -n $"Starting $prog: "
    daemon $perlfastcgi
    retval=$?
    echo
    [ $retval -eq 0 ] && touch $lockfile
    return $retval
}

stop() {
    echo -n $"Stopping $prog: "
    killproc $prog -QUIT
    retval=$?
    echo
    [ $retval -eq 0 ] && rm -f $lockfile
    return $retval
}

restart() {
    stop
    start
}

reload() {
    echo -n $" Reloading $prog: "
    killproc $nginx -HUP
    RETVAL=$?
    echo
}

force_reload() {
    restart
}
rh_status() {
    status $prog
}
```

```
rh_status_q() {
    rh_status >/dev/null 2>&1
}

case "$1" in
    start)
        rh_status_q && exit 0
        $1
        ;;
    stop)
        rh_status_q || exit 0
        $1
        ;;
    restart)
        $1
        ;;
    reload)
        rh_status_q || exit 7
        $1
        ;;
    force-reload)
        force_reload
        ;;
    status)
        rh_status
        ;;
    condrestart|try-restart)
        rh_status_q || exit 0
        ;;
    *)
        echo $"Usage: $0 {start|stop|status|restart|condrestart|try-restart|reload|force-reload}"
        exit 2
    esac
```

3.3 设置脚本权限

```
# chmod a+x /usr/bin/fastcgi-wrapper.pl
# chmod a+x /etc/rc.d/init.d/perl-fastcgi
```

4. FastCGI 测试

4.1 启动 nginx 与 fastcgi

```
# /usr/local/nginx-1.4.2/sbin/nginx
# /etc/init.d/perl-fastcgi start
```

4.2 perl 测试文件：

文件路径/data/site/test.ttlsa.com/test.pl

```
#!/usr/bin/perl
print "Content-type:text/html\n\n";
print <<EndOfHTML;
<html><head><title>Perl Environment Variables</title></head>
<body>
<h1>Perl Environment Variables</h1>
```

```
EndOfHTML
foreach $key (sort(keys %ENV)) {
    print "$key = $ENV{$key}<br>\n";
}
print "</body></html>";
```

5. 访问测试

5.1　访问

http://http:test.ttlsa.com/test.pl,出现内容表示 OK.

6. 简单压力测试：

6.1 使用 tcp/ip 方式

ab -n 1000 -c 10 http://test.ttlsa.com/test.pl

他是在是太慢了，只好用 10 个并发，共计 100 个请求来测试.

```
Document Path:          /test.pl
Document Length:        129 bytes

Concurrency Level:      10
Time taken for tests:   7.182 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      244000 bytes
HTML transferred:       129000 bytes
Requests per second:    139.24 [#/sec] (mean)
Time per request:       71.818 [ms] (mean)
Time per request:       7.182 [ms] (mean, across all concurrent requests)
Transfer rate:          33.18 [Kbytes/sec] received
```

perl + fastcgi + tcp-ip

6.2 使用 socket 方式：

```
ab -n 100000 -c 500 http://test.ttlsa.com/test.pl
```

```
Document Path:          /test.pl
Document Length:        166 bytes

Concurrency Level:      100
Time taken for tests:   1.729 seconds
Complete requests:      10000
Failed requests:        195
   (Connect: 0, Receive: 0, Length: 195, Exceptions: 0)
Write errors:           0
Non-2xx responses:      9817
Total transferred:      3100667 bytes
HTML transferred:       1654777 bytes
Requests per second:    5782.23 [#/sec] (mean)
Time per request:       17.294 [ms] (mean)
Time per request:       0.173 [ms] (mean, across all concurrent requests)
Transfer rate:          1750.86 [Kbytes/sec] received
```

perl + fastcgi + socket

很奇怪，使用 tcp/ip 方式，每秒就 140 多个请求，而使用 socket 方式却有 5800 个请求/秒。差距不是一般的大。顺便测试了一下 php 的 fastcgi，大概请求在 3000（tcp/ip 方式），4800（socket 方式）。

7. 文件下载

perl 脚本下载：perl-fastcgi,fastcgi-wrapper.pl,test.perl 三个文件

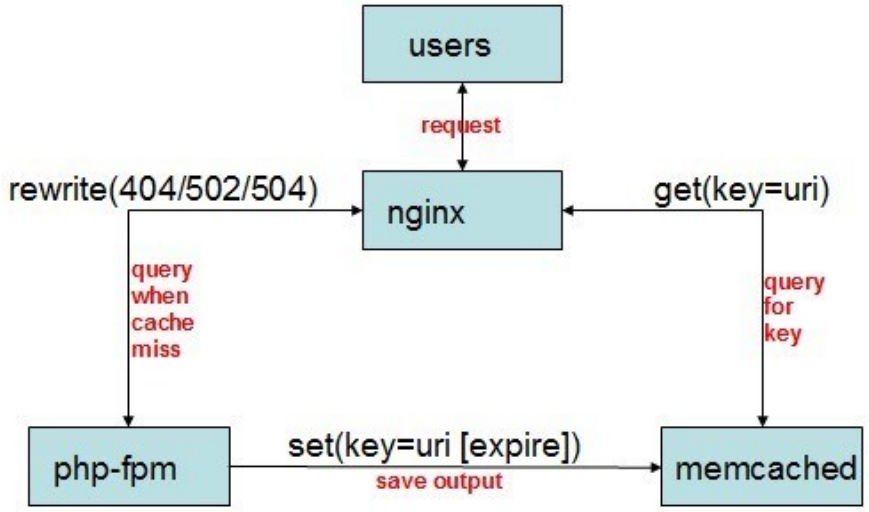
原文：http://www.ttlsa.com/nginx/perl-fastcgi-nginx/

附件：perl-fastcgi文件.rar

22.nginx+memcached 构建页面缓存应用

nginx 的 memcached_module 模块可以直接从 memcached 服务器中读取内容后输出，后续的请求不再经过应用程序处理，如 php-fpm、django，大大的提升动态页面的速度。nginx 只负责从 memcached 服务器中读取数据，要往 memcached 写入数据还得需要后台的应用程序来完成，主动的将要缓存的页面缓存到 memcached 中，可以通过 404 重定向到后端去处理的。
ngx_http_memcached_module 可以操作任何兼用 memcached 协议的软件。如 ttserver、membase 等。

结构图如下：



memcached 的 key 可以通过 memcached_key 变量来设置，如以$uri。如果命中，那么直接输出内容，没有命中就意味着 nginx 需要从应用程序请求页面。同时，我们还希望该应用程序将键值对写入到 memcached，以便下一个请求可以直接从 memcached 获取。
如果键值不存在，nginx 将报告 not found 错误。最好的方法是使用 error_page 指定和 location 请求处理。同时包含”Bad Gateway”错误和”Gateway Timeout”错误，如：error_page 404 502 504 = @app;。
注意：需要设置 default_type，否则可能会显示不正常。

1. 模块指令说明：
memcached_bind
语法：memcached_bind address | off;
默认值：none
配置段：http, server, location
指定从哪个 IP 来连接 memcached 服务器

memcached_buffer_size
语法：memcached_buffer_size size;
默认值：4k|8k;
配置段：http, server, location
读取从 memcached 服务器接收到响应的缓冲大小。尽快的将响应同步传给客户端。

memcached_connect_timeout
语法：memcached_connect_timeout time;
默认值：60s;
配置段：http, server, location
与 memcached 服务器建立连接的超时时间。通常不超过 75s。

memcached_gzip_flag
语法：memcached_gzip_flag flag;
默认值：none

配置段：http, server, location
测试 memcached 服务器响应标志。如果设置了，将在响应头部添加了 Content-Encoding：gzip。

memcached_next_upstream
语法：memcached_next_upstream error | timeout | invalid_response | not_found | off …;
默认值： error timeout;
配置段：http, server, location
指定在哪些状态下请求将转发到另外的负载均衡服务器上，仅当 memcached_pass 有两个或两个以上时使用。

memcached_pass
语法：memcached_pass address:port or socket;
默认值：none
配置段：location, if in location
指定 memcached 服务器地址。使用变量$memcached_key 为 key 查询值，如果没有相应的值则返回 error_page 404。

memcached_read_timeout
语法：memcached_read_timeout time;
默认值：60s;
配置段：http, server, location
定义从 memcached 服务器读取响应超时时间。

memcached_send_timeout
语法：memcached_send_timeout
默认值：60s
配置段：http, server, location
设置发送请求到 memcached 服务器的超时时间。

$memcached_key 变量：
memcached key 的值。

2. nginx memcached 的增强版 ngx_http_enhanced_memcached_module
基于 nginx memcached 模块的，添加的新特性有：
1. 自定义 HTTP 头，如 Content-Type, Last-Modified。
2. hash 键可超过 250 个字符，memcached 受限。
3. 通过 HTTP 请求将数据存储到 memcached。
4. 通过 HTTP 请求从 memcached 删除数据。
5. 通过 HTTP 请求清除所有 memcached 缓存数据。
6. 通过 HTTP 请求获取 memcached 状态数据。
7. 键名空间管理，来部分刷新缓存。
8. 缓存通过 If-Modified-Since 头和内容 Last-Modified 来回复 304Not Modified 请求。

3. 应用实例
nginx 配置实例：

```
upstream memcacheds {
        server 10.1.240.166:22222;
}
server  {
        listen       8080;
        server_name  nm.ttlsa.com;
        index index.html index.htm index.php;
        root  /data/wwwroot/test.ttlsa.com/webroot;
        location /images/ {
```

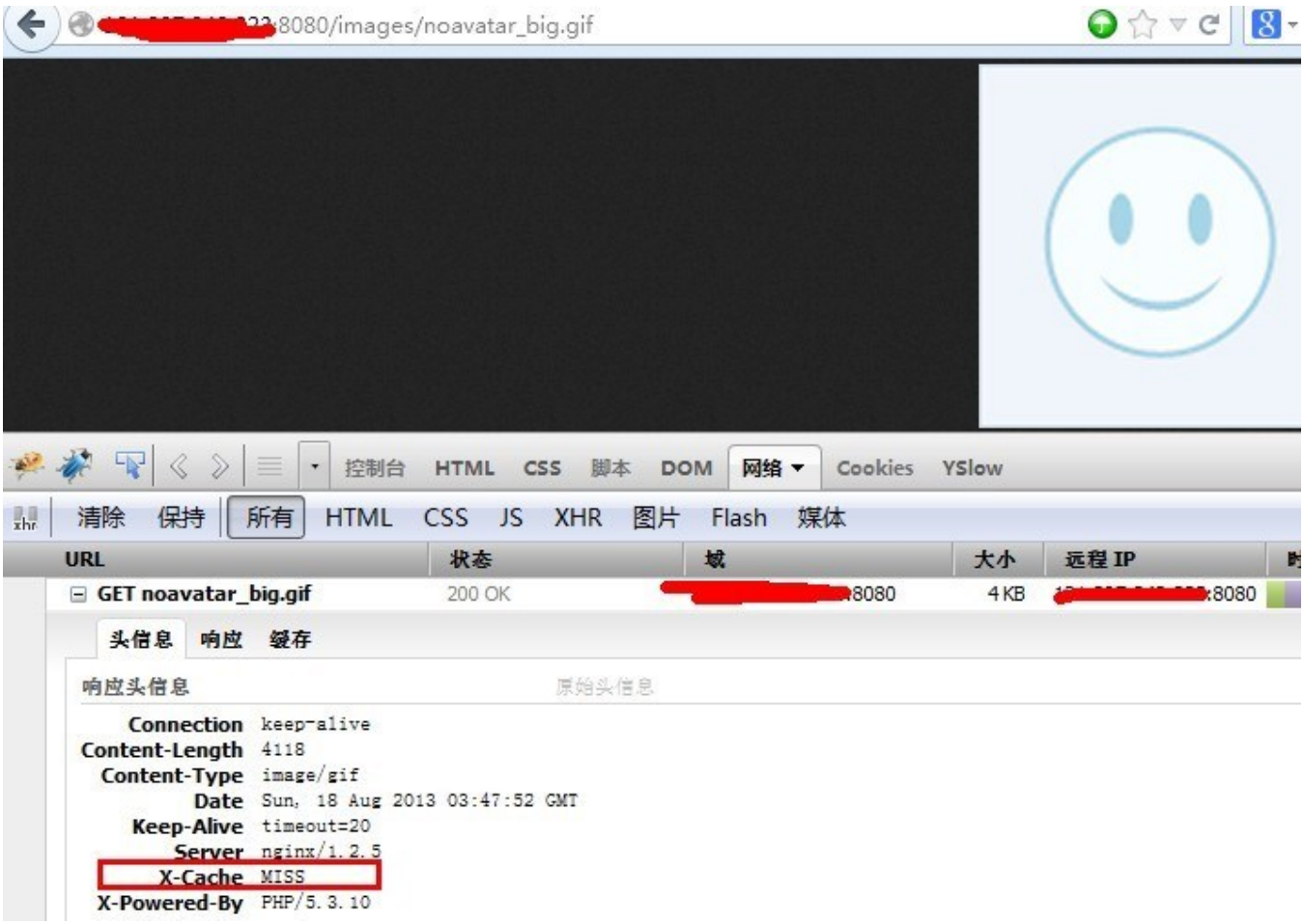
```
                set $memcached_key $request_uri;
                add_header X-mem-key  $memcached_key;
                memcached_pass  memcacheds;
                default_type text/html;
                error_page 404 502 504 = @app;
        }
        location @app {
                rewrite ^/.* /nm_ttlsa.php?key=$request_uri;
        }
        location ~ .*\.php?$
        {
                include fastcgi_params;
                fastcgi_pass  127.0.0.1:10081;
                fastcgi_index index.php;
                fastcgi_connect_timeout 60;
                fastcgi_send_timeout 180;
                fastcgi_read_timeout 180;
                fastcgi_buffer_size 128k;
                fastcgi_buffers 4 256k;
                fastcgi_busy_buffers_size 256k;
                fastcgi_temp_file_write_size 256k;
                fastcgi_intercept_errors on;
                fastcgi_param  SCRIPT_FILENAME  $document_root$fastcgi_script_name;
        }
}
```

nm_ttlsa.php 实例：

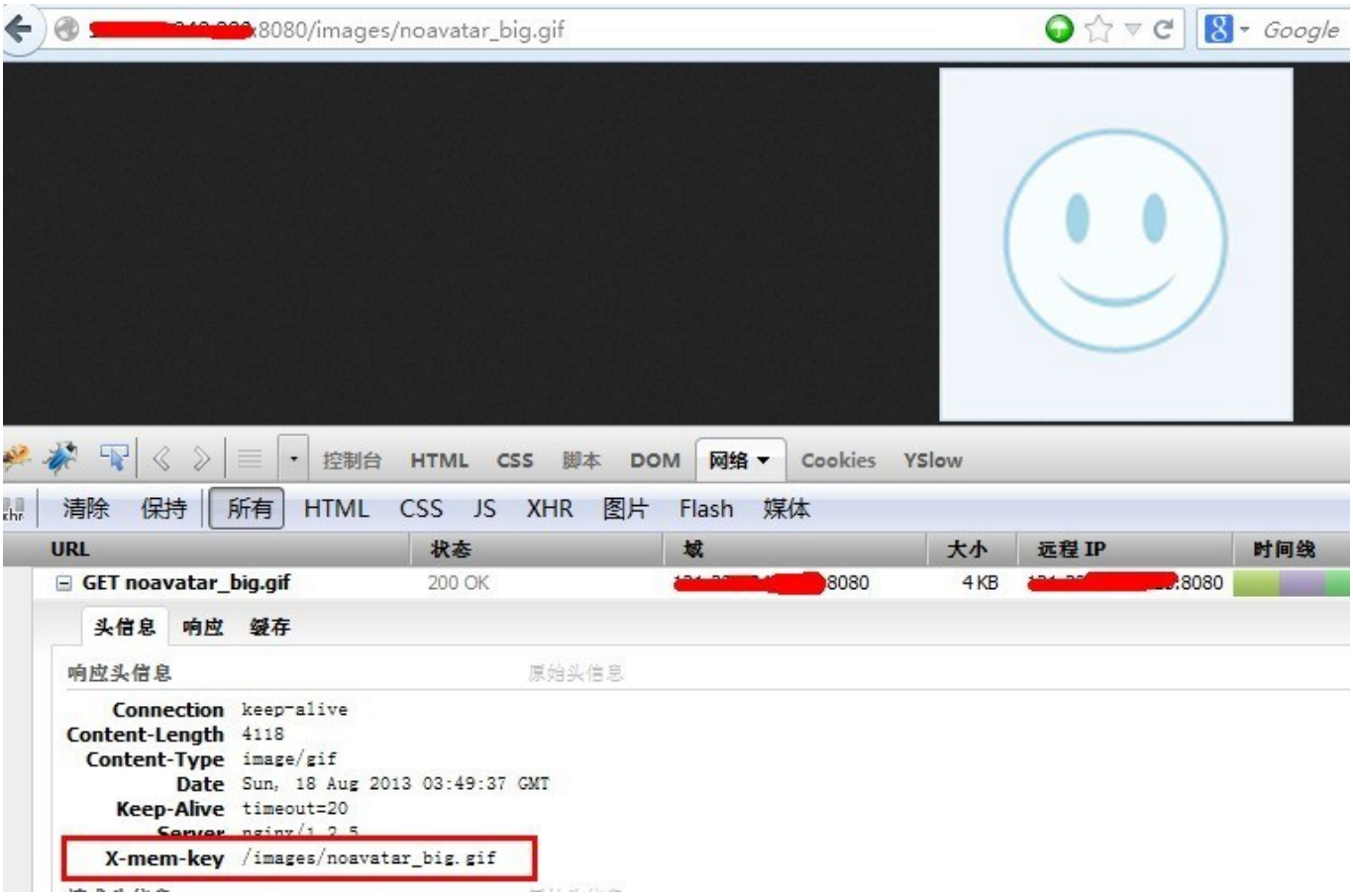
```
<?php
$fn = dirname(__FILE__)  . $_SERVER['REQUEST_URI'];
if(file_exists($fn)) {
    $data = file_get_contents($fn);
    $m = new Memcached();
    $servers = array(
                    array('10.1.240.166', 22222)
                );
    $m->addServers($servers);

    $r=$m->set($_GET['key'],$data);
    header('Content-Length: '.filesize($fn)."\r\n");
    header('Content-Type: image/gif'."\r\n");
    header('X-cache: MISS'."\r\n");
    print $data;
}else{
    header('Location: http://www.ttlsa.com'."\r\n");
}
```

4. 测试

第一次访问：（需要经过 php 处理）



再次访问：（直接从 memcached 读取）



哈，这个实例并不好。
1. 地球人都知道 memcached 不是持久化的，如果是永久性的图片应用，选用可以持久化存储方案合适，如 riak、membase、ttserver、mongodb　GridFS 等等。
2. 如果是用户头像的应用，用 memcached 来做缓存也不合适。因为用户更改头像又得刷新缓存，鉴于此，一步到位的用 ttserver 或 mongodb GridFS 来做用户头像的存储岂不是更好么。
ttserver+nginx 构建高并发高可用性应用参见：http://www.ttlsa.com/html/1429.html
这个实例改改或许可以用来在线迁移图片到 key-value 存储的过渡方案。

nginx 的 memc-nginx 和 srcache-nginx 模块可以主动的向 memcached 添加缓存。后续整理后再发布出来。参见《memc_nginx+srcache_nginx+memcached 构建透明的动态页面缓存》http://www.ttlsa.com/html/2460.html

23.memc_nginx+srcache_nginx+memcached 构建透明的动态页面缓存

在上一节《nginx+memcached 构建页面缓存应用》http://www.ttlsa.com/html/2418.html 中，说道 nginx 只负责从 memcached 服务器中读取数据，要往 memcached 写入数据还得需要后台的应用程序来完成。使用 memc-nginx 和 srcache-nginx 模块就可以主动的向 memcached 添加缓存，对应用程序来说是透明的。大大的提高动态页面访问速度。第一次访问创建缓存，后续访问在缓存过期时间内，直接从 memcached 返回，不需要再次经过 php-fpm 处理。

nginx_memc 模块与 nginx_srcache 模块配合使用，来提供缓存服务器后端的操作，在技术上，任何提供 REST 接口的模块都可以与 nginx_srcache 配合使用来获取和存储缓存子请求。

使用 memcached 作为后端缓存，需要注意 memcached 存储大小的限制，不得超过 1M。为了使用更宽松的后端存储服务，建议使用 redis 等，参见《srcache_nginx+redis 构建缓存系统》。

1. memc-nginx-module 模块指令说明：
memc_pass
语法：memc_pass address:port or socket;
默认值：none
配置段：http, server, location, if
指定 memcached 服务器地址。

memc_cmds_allowed
语法：memc_cmds_allowed <cmd>…
默认值： none
配置段：http, server, location, if
列出允许访问的 memcached 命令。默认情况下，所有的 memcached 命令都可以访问。

memc_flags_to_last_modified
语法：memc_flags_to_last_modified on|off
默认值：off
配置段：http, server, location, if
读取 memcached 标识，并将其设置为 Last-Modified 头部值。对于有条件的 get，nginx 返回 304 未修改响应，以便节省带宽。

memc_connect_timeout
语法：memc_connect_timeout <time>
默认值：60s
配置段：http, server, location
与 memcached 服务器建立连接的超时时间。不得超过 597 hours。

memc_send_timeout
语法：memc_send_timeout <time>
默认值：60s
配置段：http, server, location
设置发送请求到 memcached 服务器的超时时间。不得超过 597 hours。

memc_read_timeout
语法：memc_read_timeout <time>
默认值：60s
配置段：http, server, location
定义从 memcached 服务器读取响应超时时间。不得超过 597 hours。

memc_buffer_size
语法：memc_buffer_size <size>
默认值：4k/8k
配置段：http, server, location
读取从 memcached 服务器接收到响应的缓冲大小。

2. memcached 支持的命令
memcached 存储命令 set、add、replace、prepend、append，以$memc_key 作为键。$memc_exptime 定义过期时间，默认值为 0。$memc_flags 作为标识，默认值为 0，来建立相应的 memcached 查询。
如果$memc_value 没有定义，那么请求的请求体将作为该值，除了 incr 和 decr 命令外。注意：如果$memc_value 定义为空的字符串，那么该空字符串仍然被当做该值。
2.1 get $memc_key
使用键来检索值。

```
location /foo {
set $memc_cmd 'get';
set $memc_key 'my_key';
memc_pass 127.0.0.1:11211;
add_header X-Memc-Flags $memc_flags;
}
```

如果该键被找到，响应体为该键值，返回 200。否则范围 404 Not Found。如果发生错误或客户端错误或服务端错误，则返回 502。标识码设置到$memc_flags 变量。通常使用 add_header 指令来添加到响应头部。

2.2 set $memc_key $memc_flags $memc_exptime $memc_value
将请求体作为 memcached 值。如果另外指定值可以通过$memc_value 变量来指定。

```
location /foo {
    set $memc_cmd 'set';
    set $memc_key 'my_key';
    set $memc_flags 12345;
    set $memc_exptime 24;
    memc_pass 127.0.0.1:11211;
}
```

或

```
location /foo {
    set $memc_cmd 'set';
    set $memc_key 'my_key';
    set $memc_flags 12345;
    set $memc_exptime 24;
    set $memc_value 'my_value';
```

```
    memc_pass 127.0.0.1:11211;
}
```

返回 201，说明创建 memcached 缓存存储成功。返回 200 说明 NOT_STORED。返回 404 说明 NOT_FOUND。返回 502 说明发生错误或客户端错误或服务端错误。
memcached 原始响应是响应体，404 NOT FOUND 除外。

2.3 add $memc_key $memc_flags $memc_exptime $memc_value
和 set 命令相似

2.4 prepend $memc_key $memc_flags $memc_exptime $memc_value
和 set 命令相似

2.5 delete $memc_key
删除该键值

```
location /foo
set $memc_cmd delete;
set $memc_key my_key;
memc_pass 127.0.0.1:11211;
}
```

返回 200 说明删除成功。返回 404 说明 NOT_FOUND。返回 502 说明发生错误或客户端错误或服务端错误。

2.6 delete $memc_key $memc_exptime
和 delete 命令相似。

2.7 incr $memc_key $memc_value
给指定的$memc_key 对应的$memc_value 增量。

```
location /foo {
set $memc_key my_key;
set $memc_value 2;
memc_pass 127.0.0.1:11211;
}
```

每次访问/foo 将导致 my_key 的值加 2。
返回 200 说明成功。返回 404 说明键 Not Found。返回 502 说明发生错误或客户端错误或服务端错误。

2.8 decr $memc_key $memc_value
与 incr 相似。

2.9 flush_all
刷新 memcached 上所有的键。

```
location /foo {
set $memc_cmd flush_all;
memc_pass 127.0.0.1:11211;
}
```

2.10 flush_all $memc_exptime
与 flush_all 相似。

2.11 stats
输出 memcached 统计信息。

```
location /foo {
set $memc_cmd stats;
memc_pass 127.0.0.1:11211;
}
```

2.12 version
返回 memcached 版本信息。

```
location /foo {
set $memc_cmd version;
memc_pass 127.0.0.1:11211;
}
```

3. srcache-nginx-module 模块指令说明：
srcache_fetch
语法：srcache_fetch <method> <uri> <args>?
默认值：no
配置段：http, server, location, location if
查询缓存。返回 200 说明缓存命中，直接从缓存响应客户端请求。非 200 需要后端程序处理。

srcache_fetch_skip
语法：srcache_fetch_skip <flag>
默认值：srcache_fetch_skip 0
配置段：http, server, location, location if
<flag>支持 nginx 变量。当这个参数值不为空和不等于 0，则从缓存取数据过程被无条件跳过。

srcache_store
语法：srcache_store <method> <uri> <args>?
默认值：no
配置段：http, server, location, location if
将当前请求的响应存入缓存。可以使用 srcache_store_skip 和 srcache_store_max_size 指令禁用缓存。不管是响应状态行，响应头，响应体都会被缓存。默认情况下，下列特殊响应头不会被缓存：
Connection
Keep-Alive
Proxy-Authenticate
Proxy-Authorization
TE
Trailers
Transfer-Encoding
Upgrade
Set-Cookie
可以使用 srcache_store_pass_header、srcache_store_hide_header 指令来控制哪些头要缓存哪些不要。
注意：即使所有的响应数据被立即发送，当前的 nginx 请求生命周期未必完成，直到 srcache_store 子请求完成。这意味着服务器端延迟关闭 TCP 连接，或下一个请求服务发送同一个 TCP 连接。

srcache_store_max_size
语法：srcache_store_max_size <size>
默认值：srcache_store_max_size 0
配置段：http, server, location, location if
当响应体超过该值，将不会缓存。
当后端缓存存储有对缓存数据做硬限制，这个指令非常有用。比如 memcached 服务器，上限是 1M。
默认值 0，不限制。

srcache_store_skip
语法：srcache_store_skip <flag>
默认值：srcache_store_skip 0
配置段：http, server, location, location if
<flag>支持 nginx 变量。当这个参数值不为空和不等于 0，则从存入缓存过程被无条件跳过。

srcache_store_statuses
语法：srcache_store_statuses <status1> <status2> ..
默认值：srcache_store_statuses 200 301 302
配置段：http, server, location, location if
该指令控制那些状态码响应被缓存。

srcache_header_buffer_size
语法：srcache_header_buffer_size <size>
默认值：srcache_header_buffer_size 4k/8k
配置段：http, server, location, location if
在序列化响应头时控制头缓冲大小。默认大小为页面大小，通常为 4k 或 8k，取决于具体平台。
注意：该大小是以每个头的，因此，需要足够大来容纳最大响应头。

srcache_store_hide_header
语法：srcache_store_hide_header <header>
默认值：no
配置段：http, server, location, location if
默认情况下，除了以下头缓存所有响应头：
Connection
Keep-Alive
Proxy-Authenticate
Proxy-Authorization
TE
Trailers
Transfer-Encoding
Upgrade
Set-Cookie
可以隐藏多个响应头，不区分大小写。如
srcache_store_hide_header X-Foo;
srcache_store_hide_header Last-Modified;

srcache_store_pass_header
语法：srcache_store_pass_header <header>
默认值：no
配置段：http, server, location, location if
默认情况下，除了以下头缓存所有响应头：
Connection
Keep-Alive
Proxy-Authenticate
Proxy-Authorization
TE
Trailers
Transfer-Encoding
Upgrade
Set-Cookie
可以缓存多个响应头，不区分大小写。如
srcache_store_pass_header Set-Cookie;
srcache_store_pass_header Proxy-Autenticate;

srcache_methods
语法：srcache_methods <method>…

默认值：srcache_methods GET HEAD
配置段：http, server, location

srcache_ignore_content_encoding
语法：srcache_ignore_content_encoding on|off
默认值：srcache_ignore_content_encoding off
配置段：http, server, location, location if
内容是否编码。
建议后端服务器禁用 gzip/deflate 压缩。在 nginx.conf 配置：

```
proxy_set_header Accept-Encoding  "";
```

srcache_request_cache_control
语法：srcache_request_cache_control on|off
默认值：srcache_request_cache_control off
配置段：http, server, location
当该指令为 on 时，请求头 Cache-Control 和 Pragma 按照下面的方法处理：
1. srcache_fetch 查询缓存操作时，当请求头 Cache-Control: no-cache 、 Pragma: no-cache 将跳过。
2. srcache_store 存入缓存操作时，当请求头 Cache-Control: no-store 将跳过。
当该指令为 off 时，将禁用此功能，对于繁忙的站点依赖缓存加速被认为是最安全的。

srcache_response_cache_control
语法：srcache_response_cache_control on|off
默认值：srcache_response_cache_control on
配置段：http, server, location
当该指令为 on 时，响应头 Cache-Control 和 Expires 按照下面的方法处理：
Cache-Control: private skips srcache_store,
Cache-Control: no-store skips srcache_store,
Cache-Control: no-cache skips srcache_store,
Cache-Control: max-age=0 skips srcache_store,
Expires: <date-no-more-recently-than-now> skips srcache_store.
该指令优先级比 srcache_store_no_store，srcache_store_no_cache，srcache_store_private 高。

srcache_store_no_store
语法：srcache_store_no_store on|off
默认值：srcache_store_no_store off
配置段：http, server, location
开启该指令，将强制响应头 Cache-Control: no-store。默认为关闭。

srcache_store_no_cache
语法：srcache_store_no_cache on|off
默认值：srcache_store_no_cache off
配置段：http, server, location
开启该指令，将强制响应头 Cache-Control: no-cache。默认为关闭。

srcache_store_private
语法：srcache_store_private on|off
默认值：srcache_store_private off
配置段：http, server, location
开启该指令，将强制响应头 Cache-Control: private。默认为关闭。

srcache_default_expire
语法：srcache_default_expire <time>
默认值：srcache_default_expire 60s

配置段：http, server, location, location if
控制默认过期时间。当响应头既没有 Cache-Control: max-age=N 也没有指定 Expires 时，允许的 $srcache_expire 变量值。
该值必须小于 597hours。

srcache_max_expire
语法：srcache_max_expire <time>
默认值：srcache_max_expire 0
配置段：http, server, location, location if
控制最大缓存时间，此设置优先级高于其他计算方法。
该值必须小于 597hours。
默认为 0，不限制。

4. srcache-nginx-module 变量
$srcache_expire
当前的响应存入缓存的过期时间。按照下面的方法计算：
1. 当响应头 Cache-Control: max-age=N 被指定，那么 N 将作为过期时间。
2. 如果响应头 Expires 被指定，那么该值与当前时间差作为过期时间。
3. 当既没有指定 Cache-Control: max-age=N 也没有指定 Expires，那么使用 srcache_default_expire 指定的值。
如果超过 srcache_max_expire 指令值，那么此变量最终值为 srcache_max_expire。

$srcache_fetch_status
获取缓存的三种状态值：HIT, MISS, BYPASS。

$srcache_store_status
存入缓存的两种状态值：STORE , BYPASS。

5. 安装 nginx_memc 和 nginx_srcache 模块

```
# wget https://github.com/agentzh/memc-nginx-module/archive/master.zip
# wget https://github.com/agentzh/srcache-nginx-module/archive/master.zip
# ./configure --prefix=/usr/local/nginx-1.2.5 \
--add-module=../srcache-nginx-module
--add-module=../memc-nginx-module
# make
# make install
```

6. 配置

```
upstream memcacheds {
        server 10.1.240.166:22222;
}
server   {
        listen          8090;
        server_name    test.ttlsa.com;
        index index.html index.htm index.php;
        root    /data/wwwroot/www.ttlsa.com/webroot;

        location /memc {
                internal;
                memc_connect_timeout 100ms;
                memc_send_timeout 100ms;
                memc_read_timeout 100ms;
```

```
                set $memc_key $query_string;
                set $memc_exptime 120;
                memc_pass memcacheds;
                }

        location ~ .*\.php?$
        {
                if ($uri ~ /ttlsa/){
                        set $ttlsa_key $request_uri;
                        srcache_fetch GET /memc $ttlsa_key;
                        srcache_store PUT /memc $ttlsa_key;
                        add_header X-Cached-From $srcache_fetch_status;
                        add_header X-Cached-Store $srcache_store_status;
                }
                include fastcgi_params;
                fastcgi_pass    127.0.0.1:10081;
                fastcgi_index index.php;
                fastcgi_connect_timeout 60;
                fastcgi_send_timeout 180;
                fastcgi_read_timeout 180;
                fastcgi_buffer_size 128k;
                fastcgi_buffers 4 256k;
                fastcgi_busy_buffers_size 256k;
                fastcgi_temp_file_write_size 256k;
                fastcgi_intercept_errors on;
                fastcgi_param   SCRIPT_FILENAME   $document_root$fastcgi_script_name;
        }
}
```

7. 测试

7.1 第一次访问（404 Not found 创建缓存）

7.2 后续访问（直接从缓存中响应）

第一次访问创建缓存，后续访问在缓存过期时间内，直接从 memcached 返回，不需要再次经过 php-fpm 处理。大大提升动态页面访问速度。

《memc_nginx+srcache_nginx+memcached 遇到的问题》

24.nginx 同一个 IP 上配置多个 HTTPS 主机

最近公司域名更变，同时，又要新旧域名同时运行。 那么，对于 https 的域名在同一个 IP 上如何同时存在多个虚拟主机呢？遂，查看了下 nginx 手册，有这么一段内容，如下：

如果在同一个 IP 上配置多个 HTTPS 主机，会出现一个很普遍的问题：

```
server {
    listen              443;
    server_name         www.example.com;
    ssl                 on;
    ssl_certificate www.example.com.crt;
    ...
}

server {
    listen              443;
    server_name         www.example.org;
    ssl                 on;
    ssl_certificate www.example.org.crt;
    ...
}
```

使用上面的配置，不论浏览器请求哪个主机，都只会收到默认主机 www.example.com 的证书。这是由 SSL 协议本身的行为引起的——先建立 SSL 连接，再发送 HTTP 请求，所以 nginx 建立 SSL 连接时不知道所请求主机的名字，因此，它只会返回默认主机的证书。

最古老的也是最稳定的解决方法就是每个 HTTPS 主机使用不同的 IP 地址：

```
server {
    listen              192.168.1.1:443;
    server_name         www.example.com;
    ssl                 on;
    ssl_certificate www.example.com.crt;
    ...
}
```

```
server {
    listen              192.168.1.2:443;
    server_name         www.example.org;
    ssl                 on;
    ssl_certificate www.example.org.crt;
    ...
}
```

那么，在同一个 IP 上，如何配置多个 HTTPS 主机呢？

nginx 支持 TLS 协议的 SNI 扩展（Server Name Indication，简单地说这个扩展使得在同一个 IP 上可以以不同的证书 serv 不同的域名）。不过，SNI 扩展还必须有客户端的支持，另外本地的 OpenSSL 必须支持它。

如果启用了 SSL 支持，nginx 便会自动识别 OpenSSL 并启用 SNI。是否启用 SNI 支持，是在编译时由当时的 ssl.h 决定的（SSL_CTRL_SET_TLSEXT_HOSTNAME），如果编译时使用的 OpenSSL 库支持 SNI，则目标系统的 OpenSSL 库只要支持它就可以正常使用 SNI 了。

nginx 在默认情况下是 TLS SNI support disabled。

启用方法：

需要重新编译 nginx 并启用 TLS。步骤如下：

```
# wget http://www.openssl.org/source/openssl-1.0.1e.tar.gz
# tar zxvf openssl-1.0.1e.tar.gz
# ./configure --prefix=/usr/local/nginx --with-http_ssl_module \
--with-openssl=./openssl-1.0.1e \
--with-openssl-opt="enable-tlsext"
# make
# make install
```

查看是否启用：

```
# /usr/local/nginx/sbin/nginx -V
TLS SNI support enabled
```

这样就可以在　同一个 IP 上配置多个 HTTPS 主机了。

实例如下：

```
server   {
        listen 443;
        server_name     www.ttlsa.com;
        index index.html index.htm index.php;
        root   /data/wwwroot/www.ttlsa.com/webroot;
        ssl on;
        ssl_certificate "/usr/local/nginx/conf/ssl/www.ttlsa.com.public.cer";
        ssl_certificate_key "/usr/local/nginx/conf/ssl/www.ttlsa.com.private.key";
        ......
}

server   {
        listen 443;
        server_name     www.heytool.com;
        index index.html index.htm index.php;
        root   /data/wwwroot/www.heytool.com/webroot;
```

```
        ssl on;
        ssl_certificate "/usr/local/nginx/conf/ssl/www.heytool.com.public.cer";
        ssl_certificate_key "/usr/local/nginx/conf/ssl/www.heytool.com.private.key";
        ......
}
```

这样访问每个虚拟主机都正常。

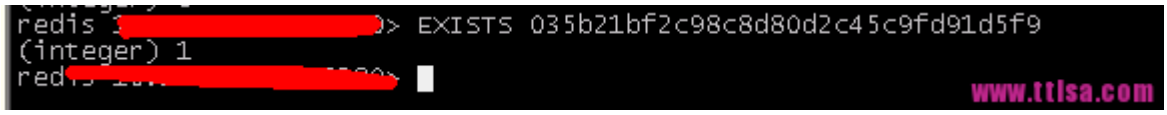
25.srcache_nginx redis 清除缓存

srcache_nginx + redis 缓存方案，我公司业务上用到的比较多。srcache_nginx 模块相关参数介绍，可以参见《memc_nginx+srcache_nginx+memcached 构建透明的动态页面缓存》。 redis 是一种高效的 key-value 存储。nginx 更是被广泛使用的 web 服务器。srcache_nginx redis 构建缓存系统应用一例可以参见：http://www.ttlsa.com/html/3952.html 。有时，又需要清除缓存。那么缓存该如何清除呢？ 缓存的清除操作与 nginx 缓存清除大同小异。关于 nginx 清缓存遇到的问题可以参考下：《nginx purge 更新缓存 404 错误》。看配置：

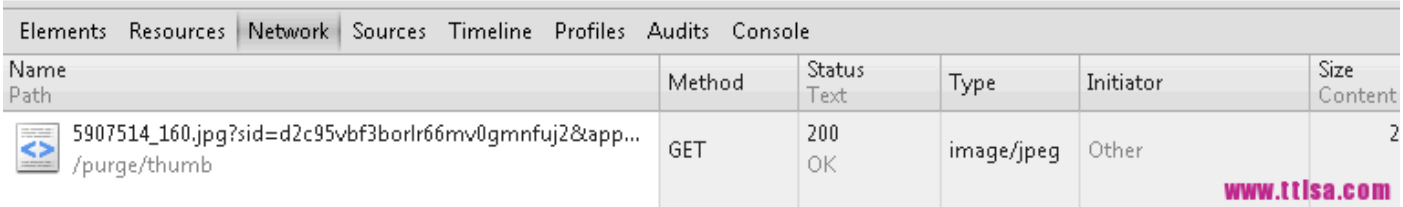
```
location ~ /purge(/.*) {
                set $key $1?$args;
                set_md5 $redis_key $key;
                redis2_query del $redis_key;
                redis2_pass redis;
        }
```

测试：

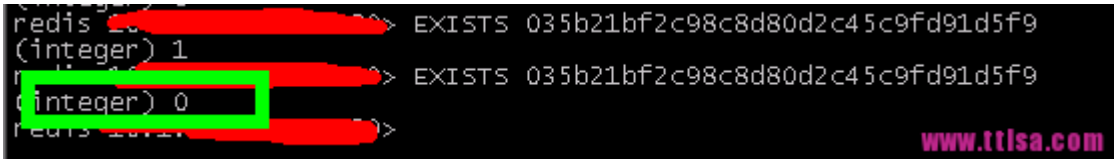
在删除前，缓存已经存在于 redis 中



清缓存操作：



验证是否还存在于 redis 中



26.nginx 动态 IP 黑白名单构建 web 防火墙(ngx_white_black_list)

功能描述：
处在黑名单中的 ip 与网络，将无法访问 web 服务。
处在白名单中的 ip，访问 web 服务时，将不受 nginx 所有安全模块的限制。
支持动态黑名单（需要与 ngx_http_limit_req 配合）
具体详见下面的说明
文件配置方法说明

一、定义黑名单或白名单方法：

1. 配置格式

配置关键字 黑名单或白名单文件 存储空间

```
white_black_list_conf conf/white.list zone=white:2m;
    | | | |
    | | | ————————————————- 存储空间大小 这里是 2m. 空间大小决定黑白名单的容量
    | | ————————————————————————————————————存储空间名
    | ————————————————————————————黑名单或白名单配置文件路径
    ————————————————————————配置命令
```

2. 配置关键字 white_black_list_conf。
3. 只能在 http{} 中使用
4. white_black_list_conf 可以配置多个 只需 zone=value 其中的 value 不同就可
5. 配置示例：

```
http{
    ……
    white_black_list_conf conf/white.list zone=white:4m;
    white_black_list_conf conf/black.list zone=black:4m;
    ……
    server{
    …….
    }
    …….
}
```

二、黑白名单作用范围

1. 配置格式

配置关键字 on/off

配置关键字有：white_list 与 black_list 分别用来表示白名单与黑名单

2. 能在 http{}、server{}、location{}下使用，功能默认是关闭
3. 配置示例：

```
http{
    ……
    white_black_list_conf conf/white.list zone=white1:4m;
    white_black_list_conf conf/black.list zone=black1:4m;
    white_list white1 on; #白名单 white1 在整个 http{} 中都开启
    black_list black1 on; #黑名单 black1 在整个 http{} 中都开启
    server{
        …….
    }
    …….
}
http{
    ……
```

```
    white_black_list_conf conf/white.list zone=white2:4m;
    white_black_list_conf conf/black.list zone=black2:4m;
    server{
        …….
        white_list white2 on; #白名单 white1 在整个 server{} 中都开启
        black_list black2 on; #黑名单 black1 在整个 server{} 中都开启
        …….
    }
    …….
}
http{
    ……
    white_black_list_conf conf/white.list zone=white3:4m;
    white_black_list_conf conf/black.list zone=black3:4m;
    white_black_list_conf conf/black.list zone=black2:4m;
    white_black_list_conf conf/white.list zone=white2:4m;
    server{
        …….
        location /do {
            ……..
            white_list white3 on; #白名单 white3 在 location /do{} 中开启
            black_list black3 on; #黑名单 black3 在 location /do{} 中开启
            ……..
        }
        location /do1{
            white_list white2 on; #白名单 white2 在整个 server{} 中都开启
            black_list black2 on; #黑名单 black2 在整个 server{} 中都开启
        }
        …….
    }
    …….
}
```

http 配置接口说明：

一、配置配置接口

```
http{
    …….
    server{
        ……
        location /sec_config{
            sec_config on;
        }
        ……
    }
    …….
}
```

二、配置方法：

1. http://xxx/sec_config 查看黑白名单定义情况

返回结果如下

{

```
    “version”：    “nginx/1.3.0″，
    “code”：    “0”，
    “item”：    {
        “conf_type”：    “white_black_list_conf”，
        “zone_name”：    “white”，
        “list_path”：    “/home/john/nginx/conf/white.list”
    },
    “item”：    {
        “conf_type”：    “white_black_list_conf”，
        “zone_name”：    “black”，
        “list_path”：    “/home/john/nginx/conf/black.list”
    },
    “item”：    {
        “conf_type”：    “white_black_list_conf”，
        “zone_name”：    “ex”，
        “list_path”：    “/home/john/nginx/conf/status_ex”
    }
}
```

2. http://xxx/sec_config?zone_name=white 查看 zone_name 为 white 的 list_path 中的具体内容

3. http://xxx/sec_config?zone_name=white&add_item=192.168.141.23 向 zone_name 为 white 中增加 192.168.141.23

4. http://xxx/sec_config?zone_name=white&delete_item=192.168.141.23 在 zone_name 为 white 中删除 192.168.141.23

查看配置方法 2:

http://xxx/sec_config?for_each

三、黑白名单文件内容

conf/black.list 文件内容如下

2.2.2.2

192.168.141.1

3.3.3.3

4.4.4.5

2.3.4.4

四、动态黑名单

要使用该功能必须对 ngx_http_limit_req_module.c 进行 patch

在 ngx_http_limit_req_module.c 中

增加#include <white_black_list.h>

并修改代码找到：

```
“
if (rc == NGX_BUSY) {
    ngx_log_error(lrcf->limit_log_level, r->connection->log, 0,
                  “limiting requests, excess: %ui.%03ui by zone \”%V\””,
                  excess / 1000, excess % 1000,
                  &limit->shm_zone->shm.name);
”
```

在其下面增加：

```
ngx_black_add_item_interface(r, 1);
```

配备关键字：

dyn_black

格式：

```
    dyn_black $zone_name time;
```

比如：

```
    dyn_black black 60; //禁止访问 60 秒，60 秒后自动解除
```

注意：

必须要配置 black_list

配置示例：

```
    http{
        ….
        white_black_list_conf conf/black.list zone=black:4m;
        limit_req_zone $binary_remote_addr zone=one:8m rate=4r/s;
        …
        server {
            location / {
      black_list black on;
      limit_req zone=one burst=6;
      dyn_black black 60; //禁止访问 60 秒，60 秒后自动解除
      …
      }
      location /xxx {
      sec_config on;
      }
      …
        }
        …
    }
```

参考文章

项目地址：https://github.com/codehunte/ngx_white_black_list

项目文档：https://github.com/codehunte/ngx_white_black_list/blob/master/white_black_list.txt

27.srcache_nginx+redis 构建缓存系统

在《memc_nginx+srcache_nginx+memcached 构建透明的动态页面缓存》一文中，我们使用到 memcached 来作为缓存载体。想必大家都知道 memcached 有存储大小的限制，不得超过 1M。 本文将使用 redis 来作为缓存载体。nginx 的 srcache_nginx 模块指令参数解释参见《memc_nginx+srcache_nginx+memcached 构建透明的动态页面缓存》。

1. nginx 模块

```
--add-module=../modules/ngx_devel_kit-0.2.18
--add-module=../modules/set-misc-nginx-module-0.22rc8
--add-module=../modules/srcache-nginx-module-0.22
--add-module=../modules/redis-nginx-module-0.3.6
--add-module=../modules/redis2-nginx-module-0.10
```

nginx 模块安装参见 ttlsa.com 中相关文档。

2. redis 安装配置

安装步骤参见：http://www.ttlsa.com/html/1646.html

配置参数解释参见：http://www.ttlsa.com/html/1226.html

配置实例：

```
# vim redis.conf
daemonize yes
pidfile /var/run/redis-6379.pid
port 6379
bind 127.0.0.1
timeout 0
tcp-keepalive 0
loglevel notice
logfile stdout
databases 16
stop-writes-on-bgsave-error yes
rdbcompression yes
rdbchecksum yes
dbfilename dump.rdb
slave-serve-stale-data yes
slave-read-only yes
repl-disable-tcp-nodelay no
slave-priority 100
maxmemory 8096mb
maxmemory-policy volatile-ttl
appendonly no
appendfsync everysec
no-appendfsync-on-rewrite no
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
lua-time-limit 5000
slowlog-log-slower-than 10000
slowlog-max-len 128
hash-max-ziplist-entries 512
hash-max-ziplist-value 64
list-max-ziplist-entries 512
list-max-ziplist-value 64
set-max-intset-entries 512
zset-max-ziplist-entries 128
zset-max-ziplist-value 64
activerehashing yes
client-output-buffer-limit normal 0 0 0
client-output-buffer-limit slave 256mb 64mb 60
client-output-buffer-limit pubsub 32mb 8mb 60
hz 10
aof-rewrite-incremental-fsync yes
```

由于只把 redis 当做缓存使用，因此没有启用持久化。

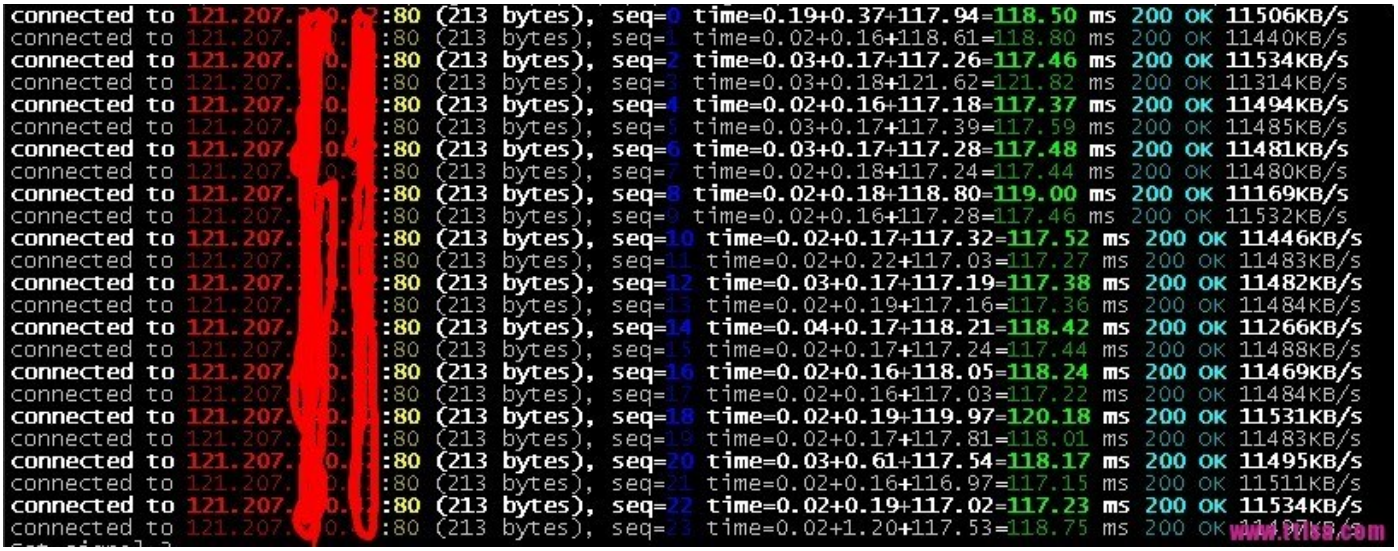
3. nginx 配置

```
# vim nginx.conf

http
{
        include         mime.types;
        default_type    application/octet-stream;
        log_format    main    '$remote_addr - $remote_user [$time_local] "$request" '
                                        '"$status" $body_bytes_sent "$http_referer" '
                                        '"$http_user_agent" "$http_x_forwarded_for" '
                                        '"$gzip_ratio" $request_time $bytes_sent $request_length';
        log_format srcache_log '$remote_addr - $remote_user [$time_local] "$request" '
                                '"$status" $body_bytes_sent $request_time $bytes_sent $request_length '
                                '[$upstream_response_time]     [$srcache_fetch_status]     [$srcache_store_status]
[$srcache_expire]';

        set_real_ip_from 10.0.0.0/8;
        real_ip_header X-Forwarded-For;
        include         vhosts/test.ttlsa.com.conf;
}
# vim vhosts/test.ttlsa.com.conf

upstream redis {
        server 127.0.0.1:6379;
        keepalive 512;
}
server
       {
        listen          80;
        server_name    test.ttlsa.com;
        index index.html index.htm index.php;
        root    /data/test.ttlsa.com/webroot;

        location ~ .*\.php {
                srcache_store_private on;
                srcache_methods GET;
                srcache_response_cache_control off;

                if ($uri ~ /ttlsa.com/pp.php$){
                        set $key $request_uri;
                        set_escape_uri $escaped_key $key;
                        srcache_fetch GET /redis $key;
                        srcache_default_expire 172800;
                        srcache_store PUT /redis2 key=$escaped_key&exptime=$srcache_expire;

                        #add_header X-Cached-From $srcache_fetch_status;
                        #set_md5 $md5key $key;
                        #add_header X-md5-key $md5key;
                        #add_header X-Cached-Store $srcache_store_status;
                        #add_header X-Key $key;
```

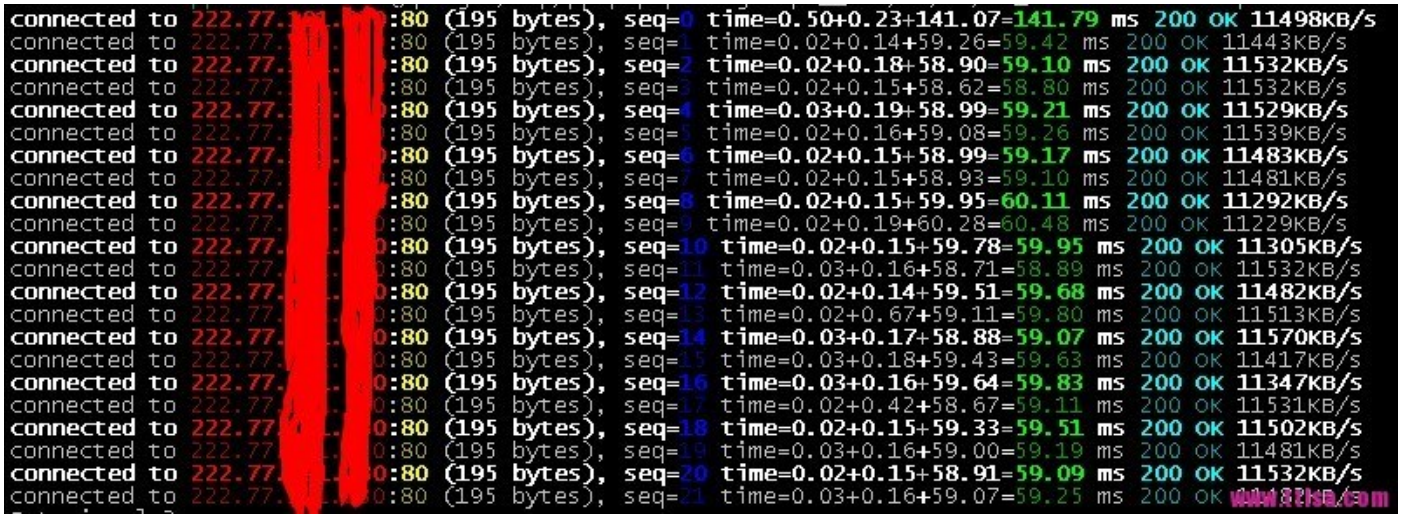
```
                        #add_header X-Query_String $query_string;
                        #add_header X-expire $srcache_expire;

                        access_log /data/httplogs/test.ttlsa.com-photo-access.log srcache_log;
                }

                include fastcgi_params;
                fastcgi_pass    127.0.0.1:9000;
                fastcgi_index index.php;
                fastcgi_connect_timeout 60;
                fastcgi_send_timeout 180;
                fastcgi_read_timeout 180;
                fastcgi_buffer_size 128k;
                fastcgi_buffers 4 256k;
                fastcgi_busy_buffers_size 256k;
                fastcgi_temp_file_write_size 256k;
                fastcgi_intercept_errors on;
                fastcgi_param    SCRIPT_FILENAME    $document_root$fastcgi_script_name;
                fastcgi_split_path_info ^(.+\.php)(.*)$;
                fastcgi_param PATH_INFO $fastcgi_path_info;
         }

        location = /redis {
                internal;
                set_md5 $redis_key $args;
                redis_pass redis;
        }

        location = /redis2 {
                internal;

                set_unescape_uri $exptime $arg_exptime;
                set_unescape_uri $key $arg_key;
                set_md5 $key;

                redis2_query set $key $echo_request_body;
                redis2_query expire $key $exptime;
                redis2_pass redis;
        }

        error_log    /data/httplogs/test.ttlsa.com-error.log;
        access_log    /data/httplogs/test.ttlsa.com-aceess.log main;

}
```

4. 测试

没有做缓存状态：

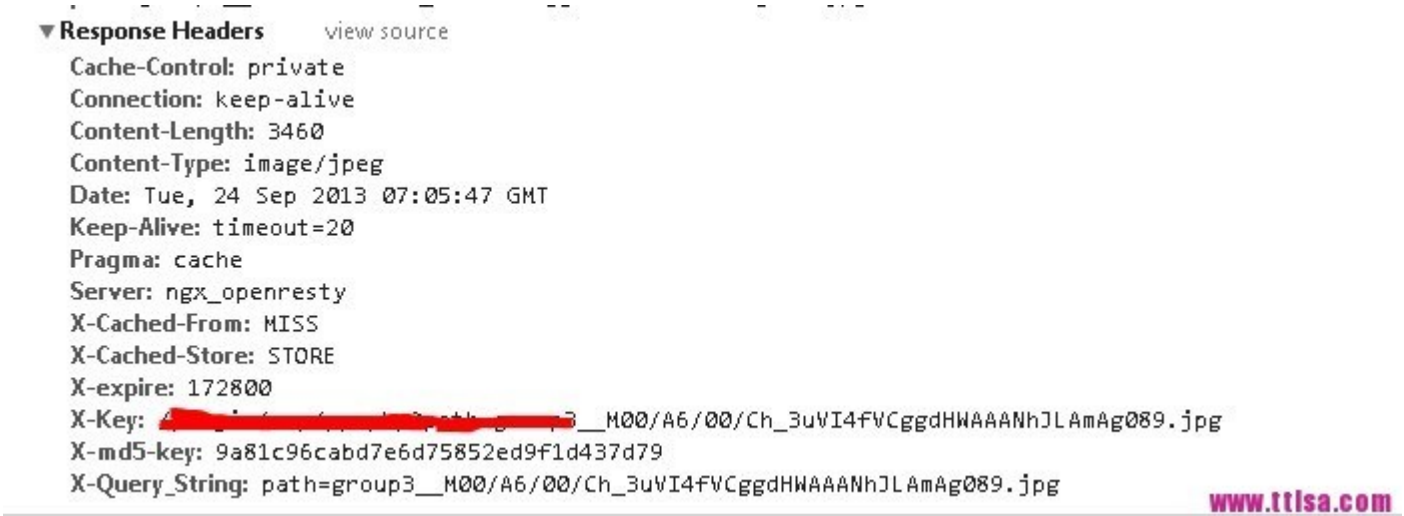


有做缓存状态：

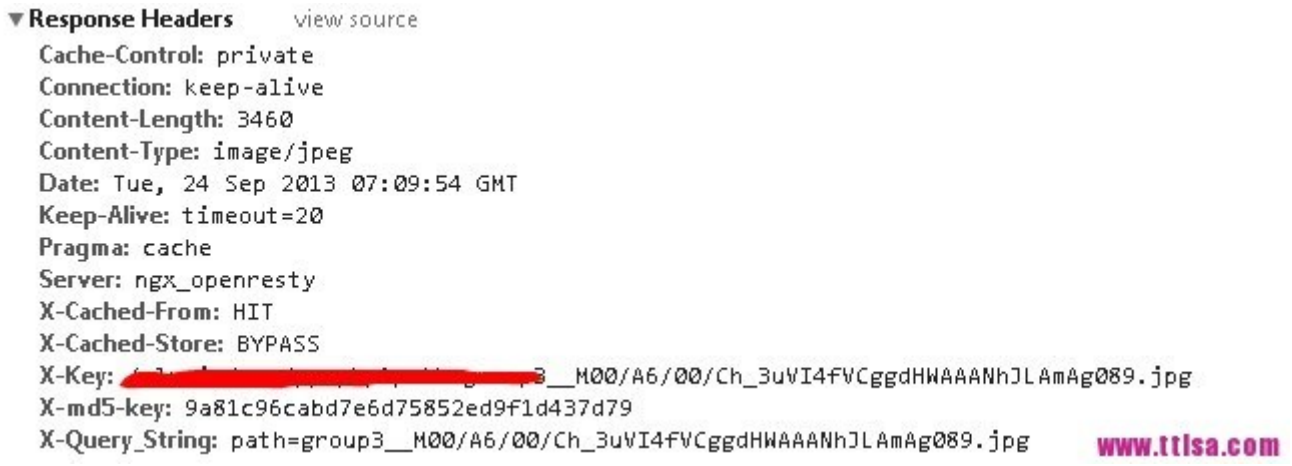


5. 响应头状态

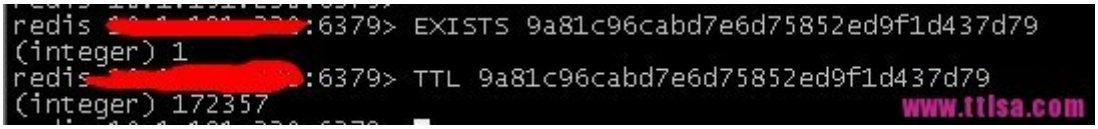
第一次请求：



再次请求：



6. 查看 redis 是否缓存以及过期时间



28.nginx 模块 nginx-http-footer-filter 研究使用

nginx-http-footer-filter 想必大家都觉得很陌生,那我们就来认识一下它吧,这是淘宝开发的 nginx 模块. 它用于 nginx 在响应请求文件底部追加内容. 今天抽空研究下这个插件，希望对大家有所帮助。为什么发现了这个插件，因为这几天公司需要在所有 shtml 文件后面追加一个 js 代码用来做统计（之前统计代码没加齐全），在寻求解决方法的过程中找到了它认识了它最后喜欢上了它，你可能以为我用这个插件去实现了我要的功能，其实在认识他之前我用 shell 脚本替换齐全了. 不过我还是决定研究测试一下 nginx-http-footer-filter，或许以后的需求上能有帮助,更或许能帮上其他需要帮助的人.进入正题吧.

1. nginx-http-footer-filter 到底是做什么的？

说白了，就是在请求的页面底部插入你要插入的代码。

2. 我们能用 nginx-http-footer-filter 来做什么？

1、统一追加 js 代码用于统计(我是这么想的)
2、底部追加响应这个请求的 realsver（后端真实服务器）信息,便于系统管理员排查故障.
3、你管理着数量庞大的虚拟主机，在所有 web 后面追加你的广告代码，黑链什么的（很无耻）
4、举一反三吧,自己想想能用来做什么吧.

淘宝用它来做什么？

打开淘宝首页，查看他源代码，拖到最下面,内容如下：

```
<!--city: fuzhou-->
<!--province: unknown-->
<!--hostname: -->
<!--hostname: home1.cn199-->
```

我们可以很清晰的看到，这边有省和地区还有主机名,也就是淘宝真实服务器的主机名,处理我这个请求的主机名为 home1.cn199, city 取到了 fuzhou,provinece 省份没取到，估计是它 Geo 的问题
或者随便打开一个商品页面, 查看源代码,如下：

```
</html>
<script type="text/javascript">TShop.initFoot({});</script>
```

可以看到他这边给这页面追加了一个 js 代码,淘宝开发这个模块的用意想必大家都明白了，集思广益，或许大家还有更好的用处.

3. 怎么安 nginx-http-footer-filter

3.1 下载地址：

https://github.com/alibaba/nginx-http-footer-filter/tree/1.2.2

3.2 安装 nginx-footer 模块

之前已经安装过 nginx，所以我选择覆盖 nginx 文件。

```
# cd /usr/local/src/
# wget https://codeload.github.com/alibaba/nginx-http-footer-filter/zip/1.2.2
# unzip 1.2.2
# http://nginx.org/download/nginx-1.4.1.tar.gz
# tar -xzvf nginx-1.4.1.tar.gz
# cd nginx-1.4.1
# ./configure --prefix=/usr/local/nginx-1.4.1 \
--with-http_stub_status_module --with-http_realip_module \
--add-module=../nginx-http-footer-filter-1.2.2
# make
# mv /usr/local/nginx-1.4.1/sbin/nginx /usr/local/nginx-1.4.1/sbin/old_nginx
# mv objs/nginx /usr/local/nginx-1.4.1/sbin/
# /usr/local/nginx-1.4.1/sbin/nginx -s stop
# /usr/local/nginx-1.4.1/sbin/nginx
```

3.3 验证模块是否安装成功

```
# /usr/local/nginx-1.4.1/sbin/nginx -V
nginx version: nginx/1.4.1
built by gcc 4.4.7 20120313 (Red Hat 4.4.7-3) (GCC)
TLS SNI support enabled
configure arguments: --prefix=/usr/local/nginx-1.4.1
--with-http_stub_status_module
--with-http_realip_module
--add-module=../nginx-http-footer-filter-1.2.2
```

4. 怎么使用 nginx-http-footer-filter 模块

4.1 配置 location

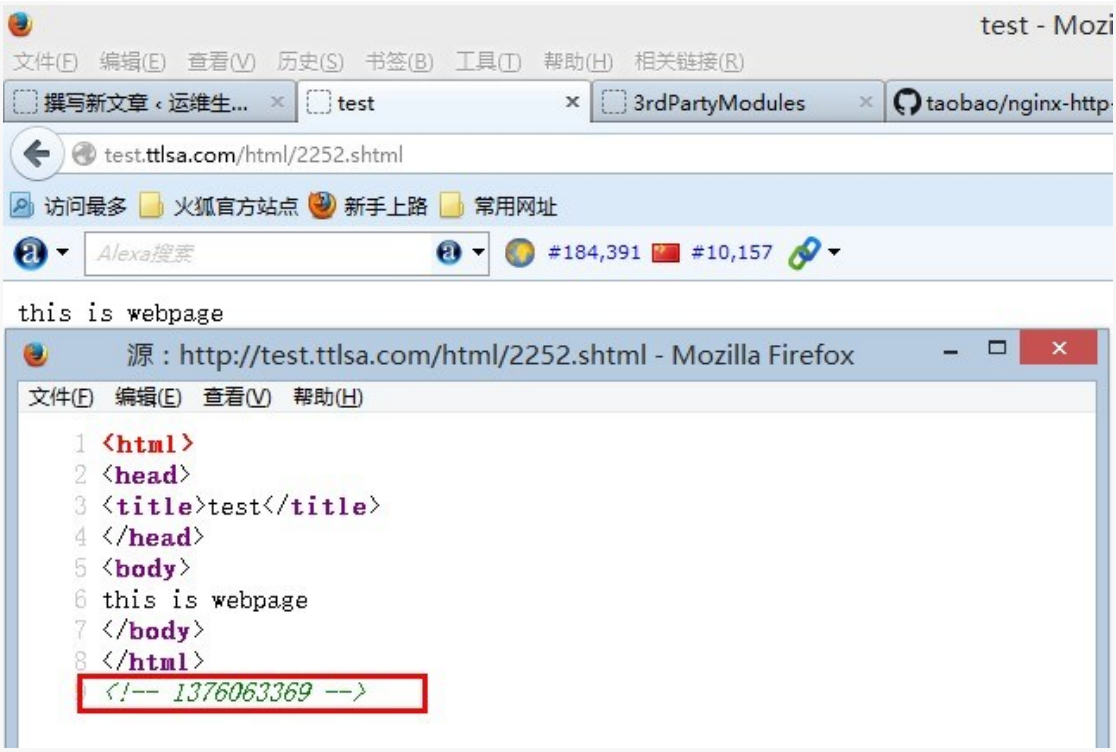
在 location 中使用 footer “你的内容” 即可.看如下配置

```
server {
        listen          173.255.219.122:80;
        server_name    test.ttlsa.com;
        access_log    /data/logs/nginx/test.ttlsa.com.access.log    main;
        index index.html index.php index.html;
        root /data/site/test.ttlsa.com;
        location / {
            footer "<!-- $date_gmt -->";
            index index.html;
        }
        location =/html/2252.css {
            footer_types text/css;
            footer "/* host: $server_name - $date_local */";
}
```

4.2 测试 nginx-footer 效果

```
# cat 2252.shtml
<html>
    <head>
      <title>test</title>
    </head>
    <body>
        this is webpage
    </body>
</html>
```

访问站点 test.ttlsa.com/html/2252.shtml



nginx 追加底部代码

如图，我们可以看到文件最底部加上了<!- 1376063369 ->,怎么变成了时间撮了,因为我这边是 ssi 的语法，如果你不知道什么是 ssi，那么请参考文章什么是 ssi.

[warning]他仅仅是在文件的最后一行追加,而不是<body>里面.这点大家要注意了.[/warning]

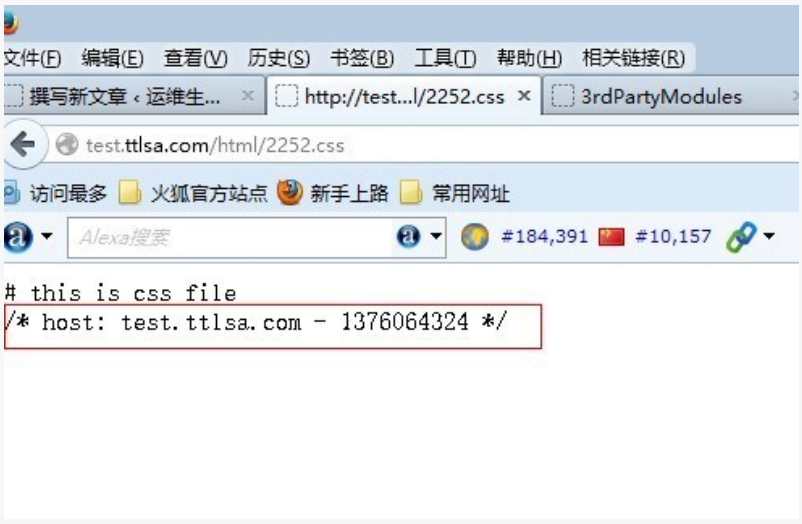
4.3 再来测试一下 css 文件

```
# cat 2242.css
# this is css file
```

如下是访问结果：

```
# this is css file
/* host: test.ttlsa.com - 1376064324 */
```

看图：



nginx footer 追加底部代码 css

5. 我能写多个 footer 指令吗？

不行，以下我写了两个 footer

```
location / {
    footer "12312321321";
    footer "<!-- $date_gmt -->";
    index index.html;
}
```

如下测试，提示 footer 指令重复了

```
# /usr/local/nginx-1.4.1/sbin/nginx -t
nginx: [emerg] "footer" directive is duplicate in /usr/local/nginx-1.4.1/conf/vhost/test.ttlsa.com.conf:13
nginx: configuration file /usr/local/nginx-1.4.1/conf/nginx.conf test failed
```

6. 只能用 ssi 变量吗？

当然不是,随便你写，可以是 ssi 指令，也可以是 nginx 变量，也可以是任何无意义的字符串

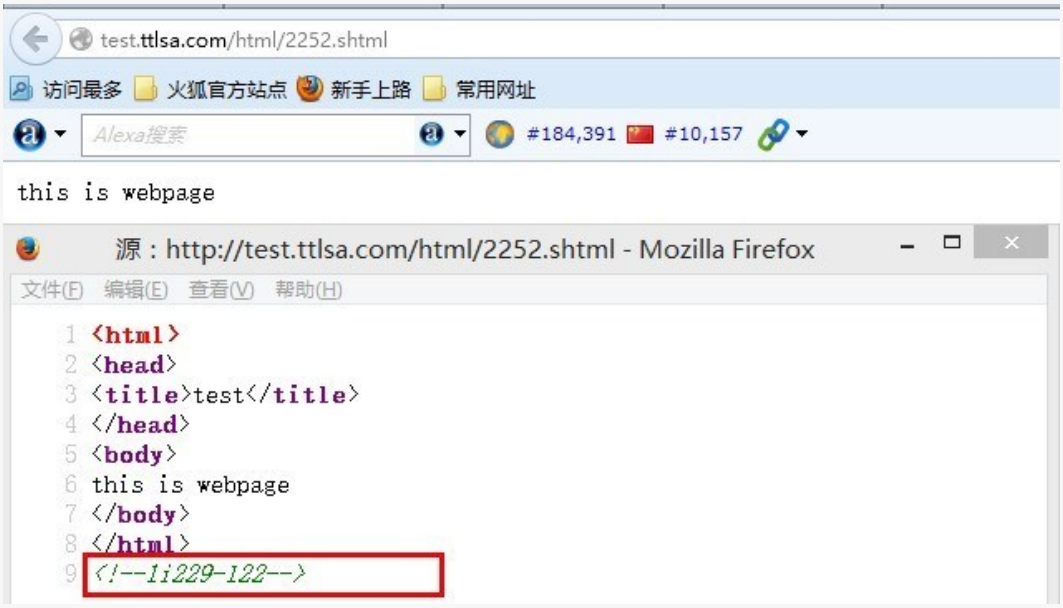
如下：

```
footer “12312321321”;
footer “<!-12312321321->”;
footer “<!-$remote_addr->”;
```

比如我想知道这个页面是哪台 web 服务器处理的,那么我在底部插入主机名即可.这样，有 500 错误，我便可以马上定位到具体的服务器了

```
footer "<!--$hostname-->";
```

返回结果如下：



nginx footer 追加底部代码 nginx 变量 hostname

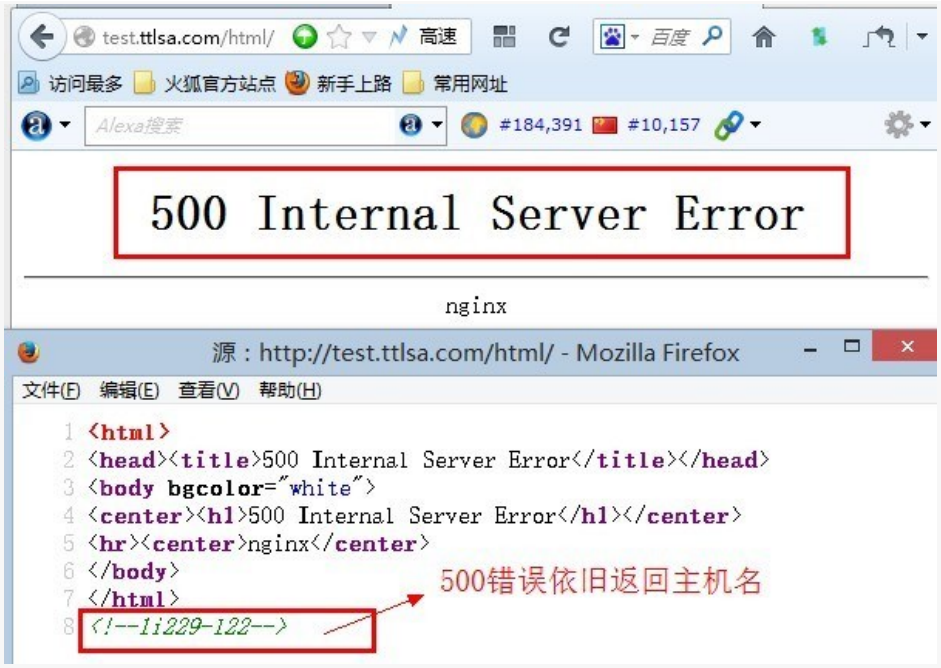
7. 服务器返回 500, 404, 403 等错误， 是否还会追加内容到底部

会, 如果不追加, 就无法通过返回的页面得知哪台 web 出现故障, 这明显就不符合作者的初衷了

配置如下：

```
location / {
    return 500;
    footer "<!--$hostname-->";
}
```

结果如下：



nginx footer 追加底部代码 500 错误

8. 模块指令说明：

footer 模块非常简单，就只有两个指令，具体说明如下

footer 字符串

默认值：“

配置段: http, server, location
这个定义了将什么内容追加到文件内容的底部

footer_types MIME 类型
默认值: footer_types: text/html
配置段: http, server, location
定义被追加底部文件的 MIME 返回类型,默认值是 text/html

29.nginx 本地缓存模块 ngx_slowfs_cache

nginx proxy 反向代理本身就支持缓存的，但是如果没有使用到 nginx 反向代理的话，就需要使用 ngx_slowfs_cache 模块来实现本地站点静态文件缓存，同时还为低速的存储设备创建快速缓存。

1. 安装 ngx_slowfs_cache 和 ngx_cache_purge 模块

```
# wget http://labs.frickle.com/files/ngx_slowfs_cache-1.10.tar.gz
# wget http://labs.frickle.com/files/ngx_cache_purge-2.1.tar.gz
# tar zxvf ngx_slowfs_cache-1.10.tar.gz
# tar zxvf ngx_cache_purge-2.1.tar.gz
# cd nginx-1.2.5
# ./configure --prefix=/usr/local/nginx-1.2.5 \
--with-http_stub_status_module --with-http_realip_module \
--add-module=../ngx_slowfs_cache-1.10 \
--add-module=../ngx_cache_purge-2.1
# make
# make install
```

2. 使用

```
http {
    slowfs_cache_path   /data/cache/proxy_cache_dir      levels=1:2      keys_zone=fastcache:4096m   inactive=1d
max_size=20g;
    slowfs_temp_path   /data/cache/proxy_temp_dir 1 2;
    server {
        location ~ /purge(/.*) {
            allow                   127.0.0.1;
            allow                   10.0.0.0/8;
            deny                     all;
            slowfs_cache_purge   fastcache $1;
        }

        location ~* \.(gif|jpg|jpeg|css|js|bmp|png)$ {
            slowfs_cache            fastcache;
            slowfs_cache_key      $uri;
            slowfs_cache_valid   1d;
            add_header X-Cache '$slowfs_cache_status from $host';
            expires   max;
        }
    }
}
```

说明：slowfs_cache_path 和 slowfs_temp_path 需要在同一分区。 slowfs_cache_path 指定缓存文件的目录级数，缓存区名称为 fastcache，内存缓存空间为 4096m，1 天没有被访问的内容自动清除，硬盘缓存空间为 20g。slowfs_cache_purge 为清除缓存。
要注意 location 执行顺序。nginx purge 更新缓存 404 错误　一例参见 http://www.ttlsa.com/html/1084.html

3. 模块指令说明
slowfs_cache
语法：slowfs_cache zone_name
默认值：none
配置段：http, server, location
定义使用的缓存区。要与 slowfs_cache_path 指令定义的一致。

slowfs_cache_key
语法: slowfs_cache_key key
默认值: none
配置段: http, server, location
设置缓存的键

slowfs_cache_purge
语法: slowfs_cache_purge zone_name key
默认值: none
配置段: location
根据指定的 key 从缓存中清除也存在的缓存

slowfs_cache_path
语法: slowfs_cache_path path [levels] keys_zone=zone_name:zone_size [inactive] [max_size]
默认值: none
配置段: http
设置缓存区和缓存结构

slowfs_temp_path
语法: slowfs_temp_path path [level1] [level2] [level3]
默认值: /tmp 1 2
配置段: http
设置临时区。文件在移到缓存区时的临时存储地。

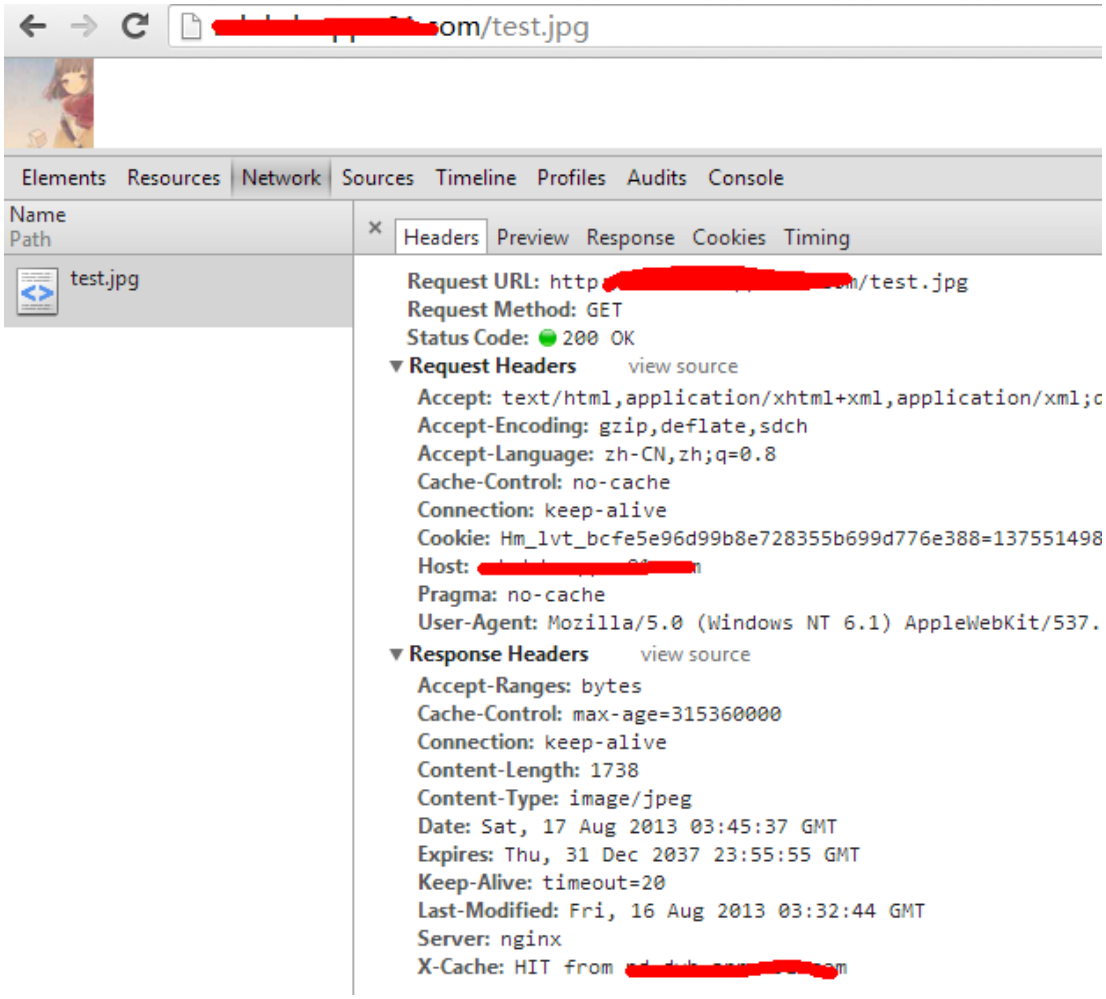
slowfs_cache_min_uses
语法: slowfs_cache_min_uses number
默认值: 1
配置段: http, server, location
设置文件被访问多少次后复制到缓存

slowfs_cache_valid
语法: slowfs_cache_valid [reply_code] time
默认值: none
配置段: http, server, location
设置缓存时间

slowfs_big_file_size
语法: slowfs_big_file_size size
默认值: 128k
配置段: http, server, location
设置大文件阀值，避免服务中断

$slowfs_cache_status 变量：
表示缓存文件的可用性，值有 MISS, HIT, EXPIRED

4. 测试



5. 清缓存

对于文件可以这么来做缓存。那么对于要经过 php 处理过的该怎么缓存呢？

30.nginx+fancy 实现漂亮的索引目录

nginx 不仅仅作为 web 站点使用，也可以当做一个文件共享的使用，索引目录列表提供用户下载文件，apache 的 index 功能很强大也很漂亮.咱们使用的 nginx，说下 nginx 索引目录（nginx 内置索引目录点我），他自带的功能很简单，而且不好看，如何做一个漂亮的索引列表.接下来看.

安装环境

系统：centos 6.3nginx:1.4.2
fancy: http://wiki.nginx.org/NgxFancyIndex

下载安装 fancy

```
# wget http://gitorious.org/ngx-fancyindex/ngx-fancyindex/archive-tarball/master
# tar -xzvf master
# wget http://nginx.org/download/nginx-1.4.2.tar.gz
# tar -xzvf nginx-1.4.2.tar.gz
# cd nginx-1.4.2
# ./configure --prefix=/usr/local/nginx-1.4.2    --add-module=../ngx-fancyindex-ngx-fancyindex
# make
# make install
```

fancy 索引配置

```
server {
listen            80;
 server_name    test.ttlsa.com;
 access_log    /data/logs/nginx/test.ttlsa.com.access.log    main;
index index.html index.php index.html;
 root /data/site/test.ttlsa.com;
location / {
}
 location ~ ^/2589(/.*)
 {
 fancyindex on;
 fancyindex_exact_size off;
 fancyindex_localtime on;
 fancyindex_footer "myfooter.shtml";
 }
 }
```

看看 nginx 加了 fancy 的效果, 如下图.

fancy+fancy

对比一下 nginx 内置的 index 效果（上篇文章贴过来的图），如下

nginx auto_index 索引效果图

参数解释：

fancyindex on：开启 fancy 索引

fancyindex_exact_size off:不使用精确的大小,使用四舍五入,1.9M 会显示为 2M 这样.如果开启的话，单位为字节

fancyindex_localtime on：使用本地时间

fancyindex_footer “myfooter.shtml”：把当前路径下的 myfooter.shtml 内容作为底部.文件不存在底部会出现 404

myfooter.shtml 内容如下：

```
<!-- footer START -->
 <div id="footer">
 <a id="gotop" href="#" onclick="MGJS.goTop();return false;">回到顶部</a>
 <a id="powered" href="http://wordpress.org/">WordPress</a>
 <div id="copyright">
 版权所有 &copy; 2011-2013 </div>
 <div id="themeinfo">
 <a href="http://www.ttlsa.com/about/">关于我们</a> | <a href="http://www.ttlsa.com/sitemap.html">网站导航</a> |
<ahref="http://www.ttlsa.com/sitemap.xml">网站地图</a> |<a rel="nofollow" href="http://www.miibeian.gov.cn/">闽ICP 备 11007147 号-1</a>
 </div>
 </div>
 <!-- footer END -->
```

fancy 指令使用：

fancyindex

语法：*fancyindex* [*on* | *off*]

默认值：fancyindex off

配置块：http, server, location

描述：开启/关闭目录索引功能

fancyindex_css_href

语法：*fancyindex_css_href uri*

默认值：fancyindex_css_href “”

配置块：http, server, location

描述：外置 css 路径，这个 css 将会替代掉现有的 css 样式。如果你会 css，那你可以把索引列表做得更加漂亮.咱们 ttlsa 没有网页设计师,所以只能用自带的了^^

fancyindex_exact_size

语法: *fancyindex_exact_size* [*on* | *off*]
默认值: fancyindex_exact_size on
配置块: http, server, location
描述: 定义如何显示文件的大小，默认是 on，on：文件大小使用精确值，单位为字节.off:单位为 KB,MB,GB，如果含有小数点，将会四舍五入。例如 1.9MB，将会显示为 2MB。

fancyindex_footer

语法: *fancyindex_footer path*
默认值: fancyindex_footer “”
配置块: http, server, location
描述: 指定哪个文件嵌入到索引页面的底部，效果请看本文的第一张图片

fancyindex_header

语法: *fancyindex_header path*
默认值: fancyindex_header “”
配置块: http, server, location
描述: 指定哪个文件嵌入到索引页面的头部.用法和 fancyindex_footer 类似

fancyindex_ignore

语法: *fancyindex_ignore string1 [string2 [… stringN]]*
默认值: No default.
配置块: http, server, location
描述: 哪些文件/目录隐藏掉，如果你的 nginx 支持正则，那么可以使用正则表达式来过滤

例如我想隐藏 dir 打头的文件或目录以及文件 filea.txt，配置如下：
fancyindex_ignore “dir*” “filea.txt”
效果如下：

对比图：

fancyindex_localtime

语法: *fancyindex_localtime* [*on* | *off*]
默认值: fancyindex_localtime off
配置块: http, server, location
Description: 使用当地时间显示文件的创建时间，默认是 off（GMT 时间）

31.nginx secure_link 下载防盗链

下载服务器上有众多的软件资源，可是很多来源不是本站，是迅雷、flashget，源源不断的带宽,防盗链绝对是当务之急. 使用来源判断根本不靠谱，只能防止一些小白站点的盗链，迅雷之类的下载工具完全无效，如果你是 nginx 的话，使用 secure link 完美解决这个问题，远离迅雷.本文仅用于下载服务器，不适用于图片防盗链.

1. 安装 nginx

默认情况下 nginx 不会安装 secure_link 模块,需要手动指定,配置参数如下

```
# ./configure --with-http_secure_link_module \
--prefix=/usr/local/nginx-1.4.2 --with-http_stub_status_module
# make
# make install
```

2. 配置 nginx：

```
server {
        listen          80;
        server_name     s1.down.ttlsa.com;
        access_log      /data/logs/nginx/s1.down.ttlsa.com.access.log    main;
        index index.html index.php index.html;
        root /data/site/s1.down.ttlsa.com;
        location / {
                secure_link $arg_st,$arg_e;
                secure_link_md5 ttlsa.com$uri$arg_e;
                if ($secure_link = "") {
                        return 403;
                }
                if ($secure_link = "0") {
                        return 403;
                }
        }
}
```

3. php 下载页面

```
<?php
 # 作用：生成 nginx secure link 链接
 # 站点：www.ttlsa.com
 # 作者：凉白开
 # 时间：2013-09-11
$secret = 'ttlsa.com'; # 密钥
 $path = '/web/nginx-1.4.2.tar.gz'; # 下载文件
 # 下载到期时间,time 是当前时间,300 表示 300 秒,也就是说从现在到 300 秒之内文件不过期
 $expire = time()+300;
# 用文件路径、密钥、过期时间生成加密串
 $md5 = base64_encode(md5($secret . $path . $expire, true));
 $md5 = strtr($md5, '+/', '-_');
 $md5 = str_replace('=', '', $md5);
# 加密后的下载地址
 echo '<a href=http://s1.down.ttlsa.com/web/nginx-1.4.2.tar.gz?st='.$md5.'&e='.$expire.'>nginx-1.4.2</a>';
 echo '<br>http://s1.down.ttlsa.com/web/nginx-1.4.2.tar.gz?st='.$md5.'&e='.$expire;
 ?>
```

4. 测试 nginx 防盗链

打开 http://test.ttlsa.com/down.php 点击上面的连接下载
下载地址如下：

http://s1.down.ttlsa.com/web/nginx-1.4.2.tar.gz?st=LSVzmZllg68AJaBmeK3E8Q&e=1378881984

页面不要刷新，等到 5 分钟后在下载一次，你会发现点击下载会跳转到 403 页面。

5. secure link 防盗链原理

- 用户访问 down.php
- down.php 根据 secret 密钥、过期时间、文件 uri 生成加密串
- 将加密串与过期时间作为参数跟到文件下载地址的后面
- nginx 下载服务器接收到了过期时间，也使用过期时间、配置里密钥、文件 uri 生成加密串

- 将用户传进来的加密串与自己生成的加密串进行对比，一致允许下载，不一致 403
整个过程实际上很简单，类似于用户密码验证. 尤为注意的一点是大家一定不要泄露了自己的密钥，否则别人就可以盗链了，除了泄露之外最好能经常更新密钥.

5. secure link 指令

secure_link
语法: secure_link md5_hash[,expiration_time]
默认: none
配置段: location
variables: yes

这个指令由 uri 中的 MD5 哈希值和过期时间组成. md5 哈希必须由 base64 加密的,过期时间为 unix 时间.如果不加过期时间,那么这个连接永远都不会过期.

secure_link_md5
语法: secure_link_md5 secret_token_concatenated_with_protected_uri
默认: none
配置段: location
variables: yes
md5 值对比结果,使用上面提供的 uri、密钥、过期时间生成 md5 哈希值.如果它生成的 md5 哈希值与用户提交过来的哈希值一致，那么这个变量的值为 1，否则为 0

secure_link_secret
语法: secure_link_secret word
默认:
配置段: location
Reference: secure_link_secret
nginx 0.8.50 之后的版本已经使用 secure_link_md5 取代,不在多说.

6.注意事项

- 密钥防止泄露、以及经常更新密钥
- 下载服务器和 php 服务器的时间不能相差太大，否则容易出现文件一直都是过期状态.

7. 最后

secure link 以及内置到了 nginx，不需要额外安装第三方模块，有下载服务器的兄弟,我极力推荐你们使用它，除非你不在乎你的带宽.

32.nginx 显示随机首页模块(Random Index)

前言

一般情况下,一个站点默认首页都是定义好的 index.html、index.shtml、index.php 等等,如果想站点下有很多页面想随机展示给用户浏览,那得程序上实现，显得尤为麻烦,如果你安装了 nginx，那么使用 nginx 的 random index 即可达成这个功能，凡是以/结尾的请求，都会随机展示当前目录下的文件作为首页.

random index 介绍

ngx_http_random_index_module 模块处理以’/’为后缀的请求,并且在当前目录下随机抽取一个页面作为首页.这个模块将在 ngx_http_index_module 模块之前执行. 默认情况下，这个模块没有安装，你需要在安装 nginx 的时候加上配置参数 - with-http_random_index_module.

随机首页配置

```
location / {
  random_index on;
 }
```

random index 指令

语法: random_index on | off;
默认值: random_index off;
配置段: location
启用或者禁用 random index 模块

33.nginx 实现图片防盗链(referer 指令)

前几天讲了《nginx 下载防盗链》，今天继续说下图片防盗链. 他们两个使用的指令不同，前者使用 secure link,并且需要程序配合,但是效果非常好;后者不需要程序配合,根据图片来源来实现,但是只能先限制基本的图片盗用,无法防止图片采集.

nginx referer 指令简介

nginx 模块 ngx_http_referer_module 通常用于阻挡来源非法的域名请求.我们应该牢记,伪装 Referer 头部是非常简单的事情，所以这个模块只能用于阻止大部分非法请求.我们应该记住，有些合法的请求是不会带 referer 来源头部的,所以有时候不要拒绝来源头部（referer）为空的请求.

图片防盗链配置

```
location ~* \.(gif|jpg|png|bmp)$ {
    valid_referers none blocked *.ttlsa.com server_names ~\.google\. ~\.baidu\.;
    if ($invalid_referer) {
        return 403;
        #rewrite ^/ http://www.ttlsa.com/403.jpg;
    }
}
```

以上所有来至 ttlsa.com 和域名中包含 google 和 baidu 的站点都可以访问到当前站点的图片,如果来源域名不在这个列表中，那么$invalid_referer 等于 1，在 if 语句中返回一个 403 给用户，这样用户便会看到一个 403 的页面,如果使用下面的 rewrite，那么盗链的图片都会显示 403.jpg。如果用户直接在浏览器输入你的图片地址,那么图片显示正常，因为它符合 none 这个规则.

nginx 防盗链指令

语法: referer_hash_bucket_size size;
默认值: referer_hash_bucket_size 64;
配置段: server, location
这个指令在 nginx 1.0.5 中开始出现.
Sets the bucket size for the valid referers hash tables. The details of setting up hash tables are provided in a separate document.

语法: referer_hash_max_size size;
默认值: referer_hash_max_size 2048;
配置段: server, location
这个指令在 nginx 1.0.5 中开始出现.
Sets the maximum size of the valid referers hash tables. The details of setting up hash tables are provided in a separate document.

语法: valid_referers none | blocked | server_names | string …;
默认值: —
配置段: server, location
指定合法的来源’referer’, 他决定了内置变量$invalid_referer 的值，如果 referer 头部包含在这个合法网址里面，这个变量被设置为 0，否则设置为 1. 记住，不区分大小写的.

参数说明

none
“Referer” 来源头部为空的情况
blocked
“Referer”来源头部不为空，但是里面的值被代理或者防火墙删除了，这些值都不以 http://或者 https://开头.
server_names
“Referer”来源头部包含当前的 server_names（当前域名）
arbitrary string
任意字符串,定义服务器名或者可选的 URI 前缀.主机名可以使用*开头或者结尾，在检测来源头部这个过程中，来源域名中的主机端口将会被忽略掉
regular expression
正则表达式,~表示排除 https://或 http://开头的字符串.

最后

图片使用来源头部做防盗链是最合理的. 简单、实用。但是没有办法防采集。如果想做文件的防盗链请参考前面章节讲到的使用 secure link 文件防盗链文章.

34.nginx 空白图片(empty_gif 模块)

用过百度统计的兄弟有没有注意到百度使用 1×1 的空白图片传递统计参数，自己做异步统计的兄弟是否使用静态文件来传递参数。为什么使用空白图片呢，而不是自己存放一张小图呢，nginx 里面的空白图片是保存在内存中的，速度绝对比硬盘上读取的快. 看下如何使用 empty_gif 生成响应 1×1 的空白图片吧.

或许哪天 ttlsa 自己要做统计，咱们也可以使用 empty_gif 来传递参数，说归说，肯定性还是比较小，能用第三方的统计就用第三方统计。好了，进入正题吧。

nginx 默认内置 ngx_http_empty_gif_module 模块, 如何安装 nginx 我不在多讲.直接看下 empty_gif 的用法

nginx 配置

nginx 模块 ngx_http_empty_gif_module 会响应 1×1 的 GIF 图片.

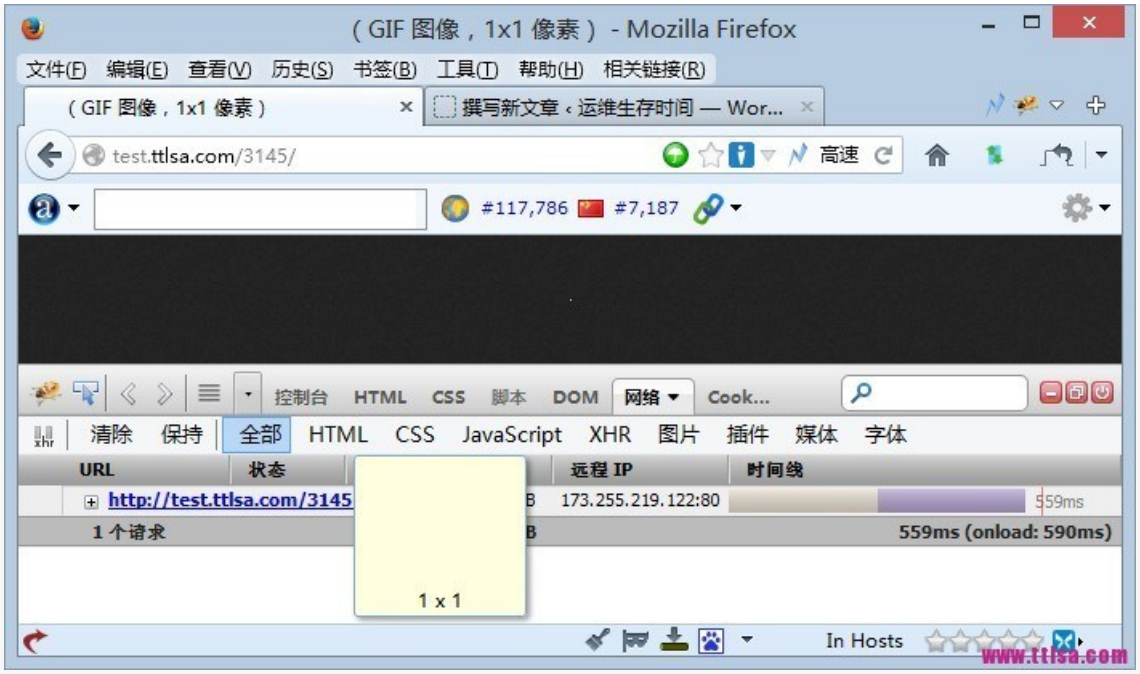
```
location = /_.gif {
        empty_gif;
}
```

如下是我的 nginx 配置

```
server {

        listen          80;
        server_name    test.ttlsa.com;
        access_log    /data/logs/nginx/test.ttlsa.com.access.log    main;
        index index.html index.php index.html;
        root /data/site/test.ttlsa.com;
        location ~* /3145/
        {
            empty_gif;
        }
}
```

测试 empty_gif

访问 test.ttlsa.com/3145/结果如下：



empty_gif nginx 空白图片

empty_gif 指令

语法:　　empty_gif;
默认:　　—
配置段:　　location
开启响应 1×1 空白图片

最后

empty_gif 用得最多的地方还是统计,当然你觉得可以用的地方也是可以用，只要是你用得着，毕竟内存速度比硬盘要快非常多.

35.nginx 记录分析网站响应慢的请求(ngx_http_log_request_speed)

nginx 模块 ngx_http_log_request_speed 可以用来找出网站哪些请求很慢，针对站点很多，文件以及请求很多想找出哪些请求比较慢的话，这个插件非常有效. 作者的初衷是写给自己用的，用来找出站点中处理时间较长的请求，这些请求是造成服务器高负载的很大根源. 日志记录之后，在使用 perl 脚本分析日志，即可知道哪些请求需要修正.

1. 模块安装

nginx 第三方模块安装方法，我们 ttlsa.com 已经说过很多次了，我这边不在重复了。
配置参数

```
./configure --prefix=/usr/local/nginx-1.4.1 --with-http_stub_status_module \
  --add-module=../ngx_http_log_request_speed
```

2. 指令 log_request_speed

log_request_speed_filter
语法: log_request_speed_filter [on|off]
配置段: n/a
context: location, server, http
启动或禁用模块

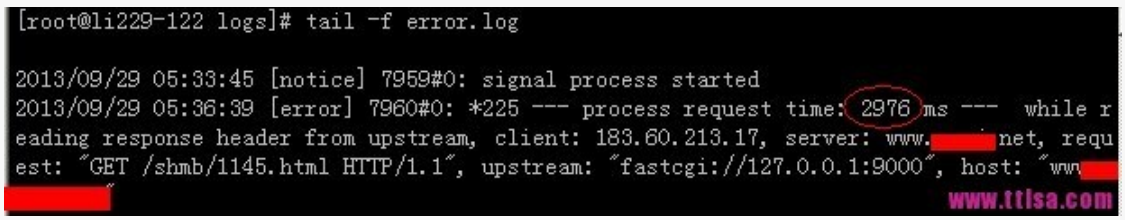
log_request_speed_filter_timeout
语法: log_request_speed_filter_timeout [num sec]
默认: 5 秒
配置段: location, server, http
这边并不是真正意义的超时,而是说当请求超过这边给定的时间，将会记录到 nginx 错误日志中. 默认值是 5000 微秒（5 秒），如果一个请求小于 5 秒,这个请求不会被记录到日志中，但是如果超过 5 秒，那请求将会被记录到 nginx 的错误日志中

3. 使用实例

3.1 nginx 配置

```
http{
        log_request_speed_filter on;
        log_request_speed_filter_timeout 3;
        ...
}
```

错误日志中记录的慢请求如下



nginx 慢请求日志

3.2 日志分析

```
cd /usr/local/nginx-1.4.1/logs
wget http://wiki.nginx.org/images/a/a8/Log_Analyzer.tar.gz
tar -xzvf Log_Analyzer.tar.gz
cd request_speed_log_analyzer
# cat ../error.log | grep 'process request'| ./analyzer.pl -r
POST /wp-admin/admin-ajax.php HTTP/1.1 --- avg ms: 1182, value count: 2
```

GET /shmb/1145.html HTTP/1.1 --- avg ms: 2976, value count: 1 <--- THE WINNER
从日志中，我们发现这边有 2 条请求比较慢，最慢的是/shmb/1145.html ，而且还标示“THE WINNER”，作者你赢了。很幽默。

3.3 分析脚本语法

./analyzer.pl -h
-h : this help message# 显示帮助信息
-u : group by upstream# 按 upstream 分组
-o : group by host# 按主机分组
-r : group by request# 按请求分组，推荐这个

4. nginx 测试版本

目前作者只在 0.6.35 和 0.7.64 下测试，不保证其他环境下可以使用。我当前的测试版本是 1.4.1，目前使用正常，在使用前请大家先测试一下。

5. 结束语

首先很感谢作者写的这个简单实用的 nginx 插件，这个插件的目的不仅仅是记录请求的响应时间，而且是用来找出响应慢的请求。如果你的服务器上有大量的站点，或者大量的程序文件，但是访问量不高，负载却很高，你想找出是哪个请求慢，我想这个插件非常适合你。

参考地址
ngx_http_log_request_speed 下载地址：
http://wiki.nginx.org/images/7/78/Ngx_http_log_request_speed.tar.gz
ngx_http_log_request_speed 脚本地址：http://wiki.nginx.org/images/a/a8/Log_Analyzer.tar.gz

36.nginx map 使用方法

map 指令使用 ngx_http_map_module 模块提供的。默认情况下，nginx 有加载这个模块，除非人为的 --without-http_map_module。
ngx_http_map_module 模块可以创建变量，这些变量的值与另外的变量值相关联。允许分类或者同时映射多个值到多个不同值并储存到一个变量中，map 指令用来创建变量，但是仅在变量被接受的时候执行视图映射操作，对于处理没有引用变量的请求时，这个模块并没有性能上的缺失。

一. ngx_http_map_module 模块指令说明

map
语法：map $var1 $var2 { … }
默认值：—
配置段：http
map 为一个变量设置的映射表。映射表由两列组成，匹配模式和对应的值。
在 map 块里的参数指定了源变量值和结果值的对应关系。
匹配模式可以是一个简单的字符串或者正则表达式，使用正则表达式要用（‘~’）。
一个正则表达式如果以 “~” 开头，表示这个正则表达式对大小写敏感。以 “~*”开头，表示这个正则表达式对大小写不敏感。

```
map $http_user_agent $agent {
        default "";
        ~curl curl;
        ~*apachebench" ab;
}
```

正则表达式里可以包含命名捕获和位置捕获，这些变量可以跟结果变量一起被其它指令使用。

```
map $uri $value {
```

```
    /ttlsa_com                      /index.php;
    ~^/ttlsa_com/(?<suffix>.*)$   /boy/;
    ~/fz(/.*)                       /index.php?;
}
```

[warning]不能在 map 块里面引用命名捕获或位置捕获变量。如~^/ttlsa_com/(.*) /boy/$1; 这样会报错 nginx: [emerg] unknown variable。[/warning]如果源变量值包含特殊字符如“~”，则要以“\”来转义。

```
map $http_referer $value {
    Mozilla      111;
    \~Mozilla   222;
}
```

结果变量可以是一个字符串也可以是另外一个变量。

```
map $num $limit {
            1 $binary_remote_addr;
            0 "";
}
```

map 指令有三个参数：

default ： 指定如果没有匹配结果将使用的默认值。当没有设置 default，将会用一个空的字符串作为默认的结果。

hostnames ： 允许用前缀或者后缀掩码指定域名作为源变量值。这个参数必须写在值映射列表的最前面。

include ： 包含一个或多个含有映射值的文件。

如果匹配到多个特定的变量，如掩码和正则同时匹配，那么会按照下面的顺序进行选择：

1. 没有掩码的字符串
2. 最长的带前缀的字符串，例如：“*.example.com”
3. 最长的带后缀的字符串，例如：“mail.*”
4. 按顺序第一个先匹配的正则表达式 （在配置文件中体现的顺序）
5. 默认值

map_hash_bucket_size

语法: map_hash_bucket_size size;

默认值: map_hash_bucket_size 32|64|128;

配置段: http

指定一个映射表中的变量在哈希表中的最大值，这个值取决于处理器的缓存。

map_hash_max_size

语法: map_hash_max_size size;

默认值: map_hash_max_size 2048;

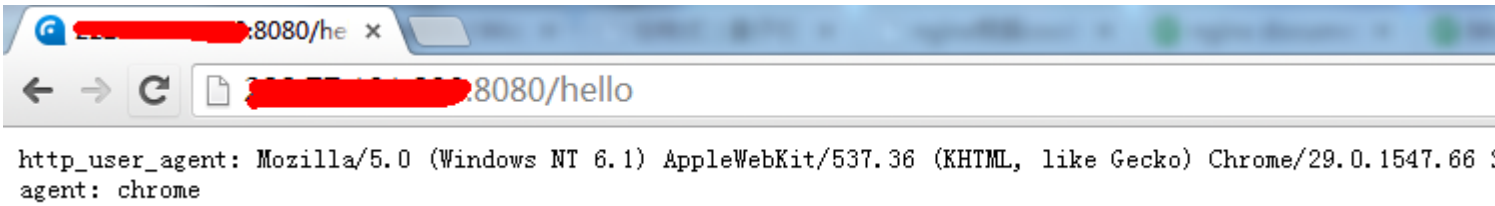
配置段: http

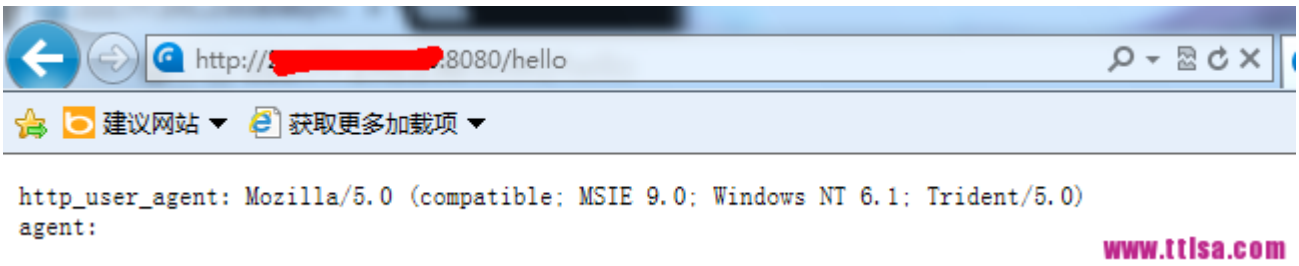
设置映射表对应的哈希表的最大值。

二. 实例

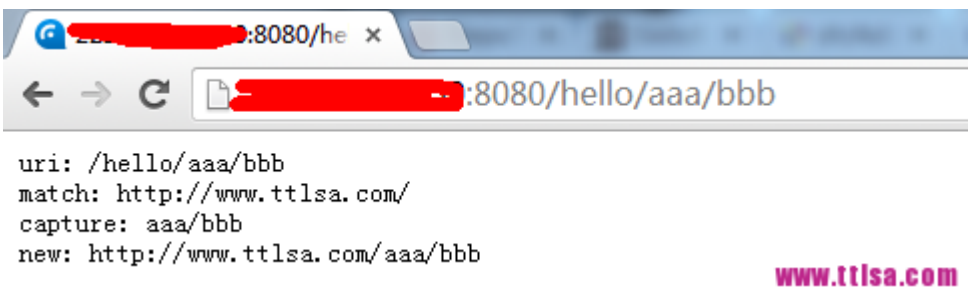
```
http {
    map $http_user_agent $agent {
        ~curl curl;
        ~*chrome chrome;
    }
    server {
        listen          8080;
        server_name    test.ttlsa.com;

        location /hello {
```

```
                default_type text/plain;
                echo http_user_agent: $http_user_agent;
                echo agent: agent:$agent;
        }
    }
}
# curl 127.0.0.1:8080/hello
http_user_agent: curl/7.15.5 (x86_64-redhat-linux-gnu) libcurl/7.15.5 OpenSSL/0.9.8b zlib/1.2.3 libidn/0.6.5
agent: curl
```






```
http {
    map $uri $match {
        ~^/hello/(.*) http://www.ttlsa.com/;
    }
    server {
        listen          8080;
        server_name     test.ttlsa.com;

        location /hello {
                default_type text/plain;
                echo uri: $uri;
                echo match: $match;
                echo capture: $1;
                echo new: $match$1;
        }
    }
}
```



37.nginx 限速白名单配置

在《nginx 限制连接数 ngx_http_limit_conn_module 模块》和《nginx 限制请求数 ngx_http_limit_req_module 模块》中会对所有的 IP 进行限制。在某些情况下，我们不希望对某些 IP 进行限制，如自己的反代服务器 IP，公司 IP 等等。这就需要白名单，将特定的 IP 加入到白名单中。下面来看看 nginx 白名单实现方法，需要结合 geo 和 map 指令来实现。geo 和 map 指令使用方法参见下面文章。《nginx geo 使用方法》和《nginx map 使用方法》。不扯蛋了，看配置。

```
http {
    geo $whiteiplist  {
        default 1;
        127.0.0.1 0;
        10.0.0.0/8 0;
        121.207.242.0/24 0;
    }
    map $whiteiplist  $limit {
        1 $binary_remote_addr;
        0 "";
    }
    limit_conn_zone $limit zone=limit:10m;
    server {
        listen          8080;
        server_name   test.ttlsa.com;
        location ^~ /ttlsa.com/ {
                limit_conn limit 4;
                limit_rate 200k;
                alias /data/www.ttlsa.com/data/download/;
        }
    }
}
```

技术要点：

1. geo 指令定义一个白名单$whiteiplist，默认值为 1，所有都受限制。 如果客户端 IP 与白名单列出的 IP 相匹配，则$whiteiplist 值为 0 也就是不受限制。
2. map 指令是将$whiteiplist 值为 1 的，也就是受限制的 IP，映射为客户端 IP。将$whiteiplist 值为 0 的，也就是白名单 IP，映射为空的字符串。
3. limit_conn_zone 和 limit_req_zone 指令对于键为空值的将会被忽略，从而实现对于列出来的 IP 不做限制。

测试方法：

```
# ab -c 100 -n 300 http://test.ttlsa.com:8080/ttlsa.com/docs/pdf/nginx_guide.pdf
```

38.nginx 修改 upstream 不重启的方法(ngx_http_dyups_module 模块)

nginx 很强大，第三方模块也不少,淘宝在 nginx 上很活跃，特别是章亦春，他参与的模块至少 10+， 好了今天主角不是他，是一款动态配置 upstream 的模块，这个模块使用 rest 接口. 简单，方便，并且可以不需要重启 nginx。但是有个问题比较明显，nginx 重启之后，什么都没了.

1. 安装 ngx_http_dyups_module

首先安装 nginx 动态 upstream 配置模块，如果你已经安装了 nginx，那么轻参考 ttlsa 上的如何安装 nginx 第三方模块，会安装的请跳过.

```
# cd /usr/local/src/
# wget https://github.com/yzprofile/ngx_http_dyups_module/archive/master.zip \
-O    ngx_http_dyups_module-master.zip
# unzip ngx_http_dyups_module-master.zip
# wget http://nginx.org/download/nginx-1.4.2.tar.gz
# tar -xzvf nginx-1.4.2.tar.gz
# cd nginx-1.4.2
# ./configure --prefix=/usr/local/nginx-1.4.2 --with-http_stub_status_module
\--add-module=../ngx_http_dyups_module-master/
# make
# make install
```

2. 指令(Directives)

语法: dyups_interface
默认: none
配置段: location
启用配置 upstream 的接口

语法: dyups_read_msg_timeout time
默认: 1s
配置段: main
设置从共享内存中读取 commands 的超时时间，默认为 1 秒

语法: dyups_shm_zone_size size
默认: 2MB
配置段: main
设置存储 commands 的共享内存
This directive set the size of share memory which used to store the commands.

语法: dyups_upstream_conf path
默认: none
配置段: main
这个指令用来指定 upstream 配置文件的路径,他会在启动的时候加载

语法: dyups_trylock on | off
默认: off
配置段: main
是否启用锁，如果启用了它，同一时刻有人在修改，那么将会返回 409.

3. restful 接口

GET
/detail 获取所有 upstream 名称以及 upstream 里面的 servers 信息

/list 获取 upstream 列表
/upstream/name 使用 upstream 名称获取 upstream 信息

POST
/upstream/name 更新 upstream
body 配置内容;
body server ip:port;

DELETE
/upstream/name 删除 upstream,name 相应修改

3.1 调用接口响应 http 状态码
500: 需要 reload nginx
409: 重新调用一次接口，上个请求被锁了.
204：调用 list 或者 detail 时出现，表示没有响应内容
其他：你的命令错误，请修改
注意：你需要第三方模块来生成新的配置文件到 nginx 配置目录. 作者也没有说什么第三方模块，这个插件很好，不能生成配置文件，让他显得尤为不足.

4. nginx 配置

备注:以下配置有安装 echo 模块.

```
http {
    # 从 upstream 读取初始 upstream 配置
    dyups_upstream_conf    conf/upstream.conf;
    include conf/upstream.conf;
    # 默认主机
    server {
        listen      80;
        location / {
            proxy_pass http://$host;
        }
    }
    # 动态配置 upstream 的接口站点
    server {
        listen 81;
        location / {
            dyups_interface;# 这个指令表示这边是接口站点
        }
    }
    # upstream 后面的 realserver,2 台 801，,82
    server {
        listen 801;
        location / {
            echo 801;
        }
    }
    server {
        listen 802;
        location / {
            echo 802;
        }
    }
```

```
}
upstream.conf 配置
upstream ttlsa1 {
    server 127.0.0.1:801;
}
upstream ttlsa12 {
    server 127.0.0.1:802;
}
```

5. 使用方法演示

5.1 添加 upstream

```
# curl -d "server 127.0.0.1:801;server 127.0.0.1:802;" 127.0.0.1:81/upstream/ttlsa3
success
```

测试

```
# curl -H "host: ttlsa3" 127.0.0.1
801

# curl -H "host: ttlsa3" 127.0.0.1
802
```

可以看到通过 host 的 ttlsa3 可以访问到 upstream 配置的两台服务器。如果你发现 curl 几次都是一样的，那么轻多试几次。

5.2 查看 upstream 详细信息

```
# curl 127.0.0.1:81/detail
ttlsa1
server 127.0.0.1:801

ttlsa2
server 127.0.0.1:802

ttlsa3
server 127.0.0.1:801
server 127.0.0.1:802
```

5.3 删除 upstream

```
# curl -i -X DELETE 127.0.0.1:81/upstream/ttlsa1
success

# curl 127.0.0.1:81/detail
ttlsa2
server 127.0.0.1:802

ttlsa3
server 127.0.0.1:801
server 127.0.0.1:802
```

5.4 增加带 ip_hash 的 upstream

```
# curl -d "ip_hash;server 127.0.0.1:801;server 127.0.0.1:802;" 127.0.0.1:81/upstream/ttlsa4
success
```

```
# curl 127.0.0.1:81/upstream/ttlsa4
server 127.0.0.1:801
server 127.0.0.1:802
```

为什么没有带 ip_hash 的信息，本身就无法显示，那我们在看看 weight 会不会显示出来

5.5 增加带 weight 的 upstream

```
# curl -d "server 127.0.0.1:801;server 127.0.0.1:802 weight=2;" 127.0.0.1:81/upstream/ttlsa5
success

# curl 127.0.0.1:81/upstream/ttlsa5
server 127.0.0.1:801
server 127.0.0.1:801
```

还是不显示,虽然没显示，但是效果还是有的，大家自己去测试吧.

6. 注意事项

本模块不能和 nginx_upstream_check_module 一起使用，接下来的版本会支持。或者可以使用 tenengine。淘宝真是不遗余力在推广他们的 tenengine.

7. 结束语

ngx_http_dyups_module 带的功能我很喜欢，但是最大的不足就是不能生成配置文件，所有内容都保存在内存中，希望以后的版本能够支持。有这个模块，shell 脚本也可以修改 upstream，不在需要重启 nginx。

39.nginx 实现简体繁体字互转以及中文转拼音(ngx_set_cconv 模块)

谈到中文简体与繁体字互转，以及汉字转拼音，大家的第一反应就是使用程序来实现，比如 php，java。最近一直在 nginx 第三方模块上晃荡，发现 nginx 可以实现简繁互转并且也同时实现了转拼音的功能，特意装上简单的测试一下。

备注：测试之前告知大家目前它只支持 utf8 编码.

1. 安装 nginx 模块

NDK 地址：http://github.com/simpl-it/ngx_devel_kit
cconv 地址：http://cconv.googlecode.com/files/cconv-0.6.2.tar.gz

1.1 安装 cconv

cconv 的 lib 提供给 nginx 模块调用，实现繁体互转以及汉字转拼音的功能.

```
# cd /usr/local/src/
# wget http://cconv.googlecode.com/files/cconv-0.6.2.tar.gz
# tar -xzvf cconv-0.6.2.tar.gz
# cd cconv-0.6.2
# ./configure
# make
# make install
```

lib 库默认安装到 usr/local 下,如果你是 64 系统执行如下命令

```
# ln -s /usr/local/lib/libcconv.so.0.0.0 /lib64/libcconv.so.0
```

32 位执行

```
# ln -s /usr/local/lib/libcconv.so.0.0.0 /lib/libcconv.so.0
```

1.2　安装 nginx

```
# cd /usr/local/src/
# wget https://github.com/simpl/ngx_devel_kit/archive/master.zip -O ngx_devel_kit-master.gzip
# wget https://github.com/liseen/set-cconv-nginx-module/archive/master.zip -O set-cconv-nginx-module-master.zip
# wget http://nginx.org/download/nginx-1.4.2.tar.gz
# unzip ngx_devel_kit-master.gzip
# unzip set-cconv-nginx-module-master.zip
# tar -xzvf nginx-1.4.2.tar.gz
# cd nginx-1.4.2
# ./configure    --prefix=/usr/local/nginx-1.4.2 --with-ld-opt='-lcconv' \
--with-http_stub_status_module --add-module=../ngx_devel_kit-master \
--add-module=../set-cconv-nginx-module-master
# make -j2
# make install
```

2. 指令（Directives）

```
set_cconv_to_simp # 繁体转简体
set_cconv_to_trad # 简体转繁体
set_pinyin_to_normal # 汉字转拼音
```

3. nginx 配置

3.1 配置 location

```
server {
      listen           80;
      server_name    test.ttlsa.com;

      location /ttlsa2jianti {
            set $ttlsa "運維生存時間  - www.ttlsa.com";
            set_cconv_to_simp $ttlsa $ttlsa;
            echo $ttlsa;
      }
      location /ttlsa2fanti {
            set $ttlsa "运维生存时间  - www.ttlsa.com";
            set_cconv_to_trad $ttlsa $ttlsa;
            echo $ttlsa;
      }
      location /ttlsa2pinyin {
            set $ttlsa "运维生存时间  -    w w w.ttlsa.com";
            set_pinyin_to_normal $ttlsa $ttlsa;
            echo $ttlsa;
      }
}
```

3.2 访问测试

```
# curl http://test.ttlsa.com/ttlsa2jianti
运维生存时间  - www.ttlsa.com

# curl http://test.ttlsa.com/ttlsa2fanti
運維生存時間  - www.ttlsa.com
```

```
# curl http://test.ttlsa.com/ttlsa2pinyin
yunweishengcunshijian -  www.ttlsa.com
```

繁体与简体都相互转化了，也可以转成拼音，而且全角的字母也转成了半角.

4. 注意事项

和程序一样，自定义变量不要使用程序内置的变量以及 $arg_XXX or $http_XXX。如下

```
set_cconv_to_simp $arg_user 'foo';
```

这种方法不要使用，会出问题.

5. 兼容性

以下版本通过测试

* 0.8.x (last tested version is 0.8.38)
* 0.7.x >= 0.7.46 (last tested version is 0.7.65)
* 0.5，0.6 这些老版本不兼容
* 1.4.2 没问题,目前我使用的是这个版本.

40.nginx 针对爬虫进行限速配置

网络爬虫一方面可以给网站带来一定的流量，便于搜索引擎收录，利于用户搜素，同时也会给服务器带来一定的压力，在网络爬虫对网站内容进行收录时，会引起服务器负载高涨。有没有什么方法既不阻止网络爬虫对网站内容进行收录，同时对其连接数和请求数进行一定的限制呢？

先来普及下 robots.txt 协议：

robots.txt（也称为爬虫协议、爬虫规则、机器人协议等）是放置在网站根目录中的.TXT 文件，是搜索引擎蜘蛛程序默认访问网站第一要访问的文件，如果 搜索引擎蜘蛛程序找到这个文件，它就会根据这个文件的内容，来确定它访问权限的范围。robots.txt 将告诉搜索引擎蜘蛛程序网站哪些页面时可以访问，哪些不可以。Robots 协议是网站国际互联网界通行的道德规范，其目的是保护网站数据和敏感信息、确保用户个人信息和隐私不被侵犯。因其不是命令，故需要搜索引擎自觉遵守。

[warning]robots.txt 必须放置在一个站点的根目录下，而且文件名必须全部小写,一词不差。[/warning]

robots.txt 写法：

User-agent: * 这里的*代表的所有的搜索引擎种类，*是一个通配符
Disallow: /admin/ 这里定义是禁止爬寻 admin 目录下面的内容
Disallow: /require/ 这里定义是禁止爬寻 require 目录下面的内容

使用 robots.txt 可以来控制某些内容不被爬虫收录，保证网站敏感数据和用户信息不被侵犯。

对爬虫进行限速处理实现方法如下：

相关内容参见：

《nginx 限制连接数 ngx_http_limit_conn_module 模块》
《nginx 限制请求数 ngx_http_limit_req_module 模块》
《nginx map 使用方法》

```
http {
    map $http_user_agent $agent {
        default "";
        ~curl $http_user_agent;
        ~*apachebench $http_user_agent;
        ~*spider $http_user_agent;
        ~*bot  $http_user_agent;
        ~*slurp $http_user_agent;
    }
```

```
    limit_conn_zone $agent zone=conn_ttlsa_com:10m;
    limit_req_zone $agent zone=req_ttlsa_com:10m rate=1r/s;

    server {
        listen          8080;
        server_name    test.ttlsa.com;
        root /data/webroot/www.ttlsa.com/

        location     / {
            limit_req zone=conn_ttlsa_com burst=5;
            limit_conn req_ttlsa_com 1;
            limit_rate 500k;
        }
    }
}
```

测试：

```
# ab -c 10 -n 300 http://test.ttlsa.com:8080/www.ttlsa.com.html
```

41.nginx 替换网站响应内容（ngx_http_sub_module）

ngx_http_sub_module 模块是一个过滤器，它修改网站响应内容中的字符串，比如你想把响应内容中的‘ttlsa’全部替换成‘运维生存时间’，这个模块已经内置在 nginx 中，但是默认未安装，需要安装需要加上配置参数：--with-http_sub_module

1. 安装 nginx

```
# wget http://nginx.org/download/nginx-1.4.2.tar.gz
# tar -xzvf nginx-1.4.2.tar.gz
# cd nginx-1.4.2
#    --prefix=/usr/local/nginx-1.4.2 --with-http_stub_status_module    --with-http_sub_module
# make
# make install
```

如果你已经安装了 nginx，只需要额外追加这个模块，请看如何安装 nginx 第三方模块

2. 指令（Directives）

语法:　　sub_filter string replacement;
默认值:　　—
配置段:　　http, server, location
设置需要使用说明字符串替换说明字符串.string 是要被替换的字符串，replacement 是新的字符串，它里面可以带变量。

语法:　　sub_filter_last_modified on | off;
默认值: sub_filter_last_modified off;
配置段:　　http, server, location
这个指令在 nginx 1.5.1 中添加，我这个版本没有，可以忽略掉.
Allows preserving the “Last-Modified” header field from the original response during replacement to facilitate response caching.
By default, the header field is removed as contents of the response are modified during processing.

语法: sub_filter_once on | off;
默认值: sub_filter_once on;

配置段: http, server, location
字符串替换一次还是多次替换，默认替换一次，例如你要替换响应内容中的 ttlsa 为运维生存时间，如果有多个 ttlsa 出现，那么只会替换第一个，如果 off，那么所有的 ttlsa 都会 被替换

语法: sub_filter_types mime-type …;
默认值: sub_filter_types text/html;
配置段: http, server, location
指定需要被替换的 MIME 类型,默认为“text/html”，如果制定为*，那么所有的

3. nginx 替换字符串实例

3.1 配置

```
server {
    listen          80;
    server_name    www.ttlsa.com;

    root /data/site/www.ttlsa.com;

    location / {
        sub_filter    ttlsa '运维生存时间';
        sub_filter_types text/html;
        sub_filter_once on;
    }
}
```

3.2 测试

内容如下

```
# cat /data/site/www.ttlsa.com/2013/10/20131001_sub1.html
welcome to tTlsa!
TTLSA TEAM!
```

访问结果

```
# curl www.ttlsa.com/2013/10/20131001_sub1.html
welcome to  运维生存时间!
TTLSA TEAM!
```

我们可以看到它替换是不区分大小写的，而且 ttlsa 只被替换了一次。我把 sub_filter_once on 改成 off 试试。

```
location / {
    sub_filter    ttlsa '运维生存时间';
    sub_filter_once off;
}
```

接着测试

```
# curl www.ttlsa.com/2013/10/20131001_sub1.html
welcome to  运维生存时间!
运维生存时间  TEAM!
```

我们可以看到 ttlsa 都被替换掉了.
例如你想在</head>后追加一段 js，配置如下：

```
location / {
    sub_filter          </head> '</head><script language="javascript" src="$script"></script>';
    sub_filter_once on;
}
```

这边我就不再做测试了,大家可以测试一下.

4. 结束语

这个 nginx 替换响应内容的模块安装使用尤为简单，应用的地方相对较少，在 nginx 中也是一个可选模块。假如站点出现什么敏感字，想修改很耗时间，不妨试试这个模块. 或者想临时在站点中加上一个通用 js 或者 css 之类的文件，也可以使用这个模块. 至于要在哪里，大家看看自己的需求.

42.nginx 向响应内容中追加内容（ngx_http_addition_module 模块）

ngx_http_addition_module 在响应之前或者之后追加文本内容，比如想在站点底部追加一个 js 或者 css，可以使用这个模块来实现，这个模块和淘宝开发的 nginx footer 模块有点类似, 但是还是有不同. 这个模块需要依赖子请求，nginx footer 依赖 nginx 写死的配置.

1. 安装 nginx

```
# wget http://nginx.org/download/nginx-1.4.2.tar.gz
# tar -xzvf nginx-1.4.2.tar.gz
# cd nginx-1.4.2
#    --prefix=/usr/local/nginx-1.4.2 --with-http_stub_status_module    --with-http_addition_module
# make
# make install
```

如果你已经安装了 nginx，只想增加模块，请参考 ttlsa 以前的文章如何安装 nginx 第三方模块

2. 指令(Directives)

语法: add_before_body uri;
默认值: —
配置段: http, server, location
发起一个子请求，请求给定的 uri，并且将内容追加到主题响应的内容之前。

语法: add_after_body uri;
默认值: —
配置段: http, server, location
发起一个子请求，请求给定的 uri，并且将内容追加到主题响应的内容之后。

syntax: addition_types mime-type …;
default: addition_types text/html;
context: http, server, location
这个指令在 0.7.9 开始支持，指定需要被追加内容的 MIME 类型, 默认为“text/html”，如果制定为*，那么所有的

3. nginx 配置 addition

3.1 配置 nginx.conf

```
server {
    listen          80;
    server_name    www.ttlsa.com;

    root /data/site/www.ttlsa.com;

    location / {
        add_before_body /2013/10/header.html;
        add_after_body    /2013/10/footer.html;
    }
}
```

3.2 测试

以下三个文件，对应请求的主体文件和 add_before_body、add_after_body 对应的内容

```
# cat /data/site/test.ttlsa.com/2013/10/20131001_add.html
<html>
<head>
      <title>I am title</title>
</head>
   <body>
         ngx_http_addition_module
   </body>
</html>

# cat /data/site/test.ttlsa.com/2013/10/header.html
I am header!

# cat /data/site/test.ttlsa.com/2013/10/footer.html
footer – ttlsa
```

访问结果如下，可以看到 20131001_add.html 的顶部和底部分别嵌入了子请求 header.html 和 footer 的内容。

```
# curl test.ttlsa.com/2013/10/20131001_add.html
I am header!
<html>
   <head>
      <title>I am title</title>
   </head>
<body>
      ngx_http_addition_module
</body>
</html>
footer - ttlsa
```

4. 结束语

addition 模块与上节上节 nginx sub 替换响应内容模块应用场景有点相同，具体怎么使用，大家结合实际情况来使用.欢迎大家继续访问运维生存时间.

43.nginx 访问控制 allow、deny（ngx_http_access_module）

单看 nginx 模块名 ngx_http_access_module,很多人一定很陌生，但是 deny 和 allow 相比没一个人不知道的，实际上 deny 和 allow 指令属于 ngx_http_access_module.我们想控制某个 uri 或者一个路径不让人访问，在 nginx 就得靠它了。

nginx 的访问控制模块语法很简单，至少比 apache 好理解，apache 的 allow 和 deny 的顺序让很多初学者抓头.好了具体看下这个插件的使用方法吧。

1、安装模块

这个模块内置在了 nginx 中，除非你安装中使用了 –without-http_access_module。如果你还没安装过 nginx，那么请参考下 ttlsa 之前写的 nginx 安装.

2、指令

allow
语法:　　allow address | CIDR | unix: | all;
默认值:　　—
配置段:　　http, server, location, limit_except

允许某个 ip 或者一个 ip 段访问.如果指定 unix:,那将允许 socket 的访问.注意：unix 在 1.5.1 中新加入的功能，如果你的版本比这个低，请不要使用这个方法。

deny
语法:　　deny address | CIDR | unix: | all;
默认值:　　—
配置段:　　http, server, location, limit_except

禁止某个 ip 或者一个 ip 段访问.如果指定 unix:,那将禁止 socket 的访问.注意：unix 在 1.5.1 中新加入的功能，如果你的版本比这个低，请不要使用这个方法。

3. allow、deny 实例

```
location / {
deny    192.168.1.1;
allow 192.168.1.0/24;
allow 10.1.1.0/16;
allow 2001:0db8::/32;
deny    all;
}
```

从上到下的顺序，类似 iptables。匹配到了便跳出。如上的例子先禁止了 192.16.1.1，接下来允许了 3 个网段，其中包含了一个 ipv6，最后未匹配的 IP 全部禁止访问.　　在实际生产环境中，我们也会使用 nginx 的 geo 模块配合使用，有兴趣的请参考 ttlsa 相关文章 nginx geo 使用方法.

4. 结束语

nginx 访问控制模块要数 nginx 里面最简单的指令，只要记住你想禁止谁访问就 deny 加上 IP，想允许则加上 allow ip，想禁止或者允许所有，那么 allow all 或者 deny all 即可.

44.nginx+perl 模块的使用

咱们上一次测试过了 perl-fastcgi，这节也来讲讲 nginx 内置模块 perl。转这文章之前我也专程做过性能对比，nginx 内置 perl 模块性能比 fastcgi 更强，这个显而易见的事情，不过性能好不了太多，一个并发 5800，一个并发 6000 多点。类似的组合有 nginx + perl + fastcgi 以及 nginx + lua 的性能也非常出色，喜欢的可以跳过去看看，看 nginx 中内置 perl 模块的继续往下走.

如果对于一个绝大部分内容是静态的网站，只有极少数的地方需要动态显示，碰巧你又了解一点 perl 知识，那么 nginx + perl 的结合就能很好解决问题。要想 nginx 支持 perl 脚本，在编译 nginx 时候需要如下参数：

./configure --with-http_perl_module
如果 make 时候出现如下类似错误：
Can’t locate ExtUtils/Embed.pm in @INC (@INC contains: /usr/lib/perl5/5.10.0/i386-linux-thread-multi /usr/lib/perl5/5.10.0 /usr/local/lib/perl5/site_perl/5.10.0/i386-linux-thread-multi /usr/local/lib/perl5/site_perl/5.10.0 /usr/lib/perl5/vendor_perl/5.10.0/i386-linux-thread-multi /usr/lib/perl5/vendor_perl/5.10.0 /usr/lib/perl5/vendor_perl /usr/local/lib/perl5/site_perl .)

你的机器上可能需要安装 perl-devel perl-ExtUtils-Embed，对于 centos 系统，直接使用 yum 搞定，例如：

```
yum -y install perl-devel perl-ExtUtils-Embed
```

nginx 中使用 perl 有两种方法，一种是直接在配置文件写，还有一种是把 perl 脚本写在外部文件中，下面主要介绍一下第二种用法。

假设 nginx 的根目录为/usr/local/nginx，perl 脚本存放的目录为 nginx 的根目录下的 perl/lib 下，脚本名字为 test.pm，nginx 配置为：

```
#位于 http 配置中
    perl_modules    perl/lib;
    perl_require    test.pm;

#位于 server 配置中
        location /user/ {
            perl pkg_name::process;
        }
```

上述配置是把所有来自 http://servername/user/下的请求交由 test.pm 脚本中定义的 process 方法来处理。

test.pm 脚本的内容如下：

```
package pkg_name;
use Time::Local;
use nginx;
sub process {
    my $r = shift;
    $r-&gt;send_http_header('text/html; charset=utf-8');
    my @arr = split('/', $r-&gt;uri);
    my $username = @arr[2];
    if (!$username || ($username eq "")) {
        $username = "Anonymous";
    }
    $r-&gt;print('Hello, You name is : <strong>' . $username . '</strong>');
    $r-&gt;rflush();
    return;
}
1;
__END__
```

当你访问 http://servername/user/netingcn，你应该可以在网页上看到：

```
Hello, You name is : netingcn
```

另外：当使用 use nginx 时,会有如下的对象可以调用，可以看到上面 shift 一个对象到 $r 上,然后就可以用 $r 调用那些对象了:

$r->args - 请求的参数 .

$r->discard_request_body - 这个参数是让 Nginx 放弃 request 的 body 的内容.

$r->filename - 返回合适的请求文件的名字

$r->has_request_body(function) - 如果没有请求主体,返回 0,但是如果请求主体存在,那么建立传递的函数并返回 1,在程序的最后,nginx 将调用指定的处理器.

$r->header_in(header) - 查找请求头的信息

$r->header_only - 如果我们只要返回一个响应的头

$r->header_out(header, value) - 设置响应的头

$r->internal_redirect(uri) - 使内部重定向到指定的 URI,重定向仅在完成 perl 脚本后发生.可以使用

header_out(Location….的方法来让浏览器自己重定向
$r->print(args, …) - 发送数据给客户端
$r->request_body - 得到客户端提交过来的内容 （body 的参数,可能需要修改 nginx 的 client_body_buffer_size. ）
$r->request_body_file —给客户的 body 存成文件,并返回文件名
$r->request_method — 得到请求 HTTP method.
$r->remote_addr - 得到客户端的 IP 地址.
$r->rflush - 立即传送数据给客户端
$r->sendfile(file [, displacement [, length]) - 传送给客户端指定文件的内容,可选的参数表明只传送数据的偏移量与长度,精确的传递仅在 perl 脚本执行完毕后生效.这可是所谓的高级功能啊
$r->send_http_header(type) - 添加一个回应的 http 头的信息
$r->sleep(milliseconds, handler) - 设置为请求在指定的时间使用指定的处理方法和停止处理,在此期间 nginx 将继续处理其他的请求,超过指定的时间后,nginx 将运行安装的处理方法,注意你需要为处理方法通过一个 reference,在处理器间转发数据你可以使用$r->variable().
$r->status(code) - 设置 http 的响应码
$r->unescape(text) - 使用 http 方法加密内容如 %XX
$r->uri - 得到请求的 URL.
$r->variable(name[, value]) - 设置变量的值

45.nginx 索引目录配置

为了简单共享文件，有些人使用 svn，有些人使用 ftp，但是更多得人使用索引(index)功能。apache 得索引功能强大，并且也是最常见得，nginx 得 auto_index 实现得目录索引偏少,而且功能非常简单。先来看看我们得效果图。

nginx auto_index 索引效果图

nginx 配置

```
location ~ ^/2589(/.*)
{
        autoindex on; //开启
        autoindex_localtime on;//开启显示功能
}
```

auto_index 指令

语法:	autoindex on | off;
配置段:	autoindex off;
配置段:	http, server, location

启用/仅用 nginx 目录索引功能.

语法:	autoindex_exact_size on | off;
配置段	autoindex_exact_size on;
配置段:	http, server, location

制定是否额外得显示文件得大小，单位为字节，mb,gb 等等. 默认是打开得

syntax:	autoindex_localtime on | off;
配置段:	autoindex_localtime off;
配置段:	http, server, location

指定是否显示目录或者文件得时间，默认是不显示

可以发现 nginx 的索引功能非常单一，比 apache 的差远了。有很多 nginx 的爱好者就看不过去了，开发了一个漂亮的索引插件,名叫 fancy index.

46.nginx+video-thumbextractor 生成视频缩略图

前言

这年头都看网络视频吧,优酷,搜狐,土豆,爱奇艺.打开页面都能看到视频的一个截图，这些图片怎么来的，难道是用暴风影音截图弄出来的?不是吧，一般是用服务器上的程序截图之后传到图片服务器上了。可以用 php、java 等等来生成视频缩略图，nginx 也有这方面的功能，一起来探索一下。

系统环境

Linux:centos 5/6
ImageMagick：yum 安装
LibJpeg： v8
地址：http://www.ijg.org/files/
nginx：1.4.2
地址：http://nginx.org/en/download.html
nginx-video-thumbextractor：v0.1.0 .
地址：https://github.com/wandenberg/nginx-video-thumbextractor-module

支持格式

mp4, mov and flv.

最小图片

最小能生成 16×16 的图片

软件安装

安装 ImageMagick
yum install ImageMagick* ImageMagick-*
安装 libjpeg
wget http://www.ijg.org/files/jpegsrc.v8.tar.gz

```
# tar -xzvf jpegsrc.v8.tar.gz
# cd jpeg-8/
# ./configure --enable-static --enable-shared
# make
#make install
```

安装 yasm

```
# wget https://github.com/yasm/yasm/archive/master.zip
# unzip master.zip
#cd yasm-master
# ./configure
# make
# make install
```

安装 ffmpeg

```
# wget http://ffmpeg.org/releases/ffmpeg-2.0.1.tar.gz
# tar -xzvf ffmpeg-2.0.1.tar.gz
# cd ffmpeg
# ./configure --prefix=/usr --disable-ffserver --disable-ffplay --enable-shared
# make
# make install
```

安装 nginx

```
# wget https://codeload.github.com/wandenberg/nginx-video-thumbextractor-module/zip/master -O nginx-video-thumbextractor-module-master.zip
# unzip nginx-video-thumbextractor-module-master.zip
# wget http://nginx.org/download/nginx-1.4.2.tar.gz
# tar -xzvf nginx-1.4.2.tar.gz
＃cd nginx-1.4.2
# ./configure --prefix=/usr/local/nginx-1.4.2 --add-module=../nginx-video-thumbextractor-module-master --with-cc-opt='-I /usr/include/ImageMagick'
# make
# make install
```

nginx 配置

```
server {
        listen          80;
        server_name    test.ttlsa.com;
        access_log    /data/logs/nginx/test.ttlsa.com.access.log    main;
        index index.html index.php index.html;
        root /data/site/test.ttlsa.com;
        location / {
        }
        location ~ /2687/(.*)
        {
                alias /data/site/test.ttlsa.com/2687/;
                video_thumbextractor;
                video_thumbextractor_video_filename      $1;
                video_thumbextractor_video_second        $arg_second;
                video_thumbextractor_image_width         $arg_width;
                video_thumbextractor_image_height        $arg_height;
        }
}
```

测试

创建站点目录

```
# mkdir /data/site/test.ttlsa.com/2687/
```

上传文件到这个目录下，我传的是 v_baofeng.mp4

访问

下面三种访问方式都是正确的.second 这个才是是必填的,否则会出现 400 错误.

http://test.ttlsa.com/2687/v_baofeng.mp4?second=77&width=400&height=200

nginx 视频缩略图-1

http://test.ttlsa.com/2687/v_baofeng.mp4?second=77&width=300

nginx 视频缩略图-2

http://test.ttlsa.com/2687/v_baofeng.mp4?second=77&height=100

nginx 视频缩略图-3

说明：second 视频中的时间点，秒为单位。width 生成的图片宽度，height 生成图片高度。两个参数都设置会裁切图片，如果只设置一个那么会根据那个参数等了比例生成图片.

指令

video_thumbextractor

语法: video_thumbextractor
配置段: location
发行版本: 0.1.0
开启缩略图功能

video_thumbextractor_video_filename
语法: video_thumbextractor_video_filename filename
默认值: none
配置段: http
发行版本: 0.1.0
文件名,如果文件不存在会返回 404

video_thumbextractor_video_second
语法: video_thumbextractor_video_second second
默认值: none
配置段: http
发行版本: 0.1.0
取某一秒的关键帧,如果指定的时间超过了视频的长度，将会返回 404.如果未指定这个值将会返回 400 错误.

video_thumbextractor_image_width
语法: video_thumbextractor_image_width width
默认值: 0
配置段: http

发行版本：0.1.0
生成图片的宽度，这是一个可选项.如果仅仅指定了宽度，那么图片高度会按原比例缩放

video_thumbextractor_image_height
语法：video_thumbextractor_image_height height
默认值：0
配置段：http
发行版本：0.1.0

生成图片的高度，这是一个可选项.如果仅仅指定了高度，那么图片宽度会按原比例缩放
如果这两项都指定了，那么会裁切图片.到时候大家看到的图片可能就是残缺的.

常见错误

1、Wrong JPEG library version: library is 80, caller expects 62
JPEG 版本不匹配，需要安装 v8 版本。

安装过程麻烦了一点，但是配置和使用非常简单。用户上传视频到视频服务器上之后，使用这个方法就可以去到视频的缩略图了.是不是很简单

47.Nginx 国人开发缩略图模块(ngx_image_thumb)

关于

ngx_image_thumb 是 nginx 中用来生成缩略图的模块，生存缩略图的方法很多，之前也写过一篇 《nginx 生成缩略图配置 - ttlsa 教程系列之 nginx》，在 github 上发现国人开发的一款模块，作者的文档写的很详细，我便照搬过来了。以后将做一个测试.

特性

本 nginx 模块主要功能是对请求的图片进行缩略/水印处理，支持文字水印和图片水印。支持自定义字体，文字大小，水印透明度，水印位置，判断原图是否是否大于指定尺寸才处理等等

1. 编译方法

编译前请确认您的系统已经安装了 libcurl-dev libgd2-dev libpcre-dev 依赖库
1.1 Debian / Ubuntu 系统举例

```
# 如果你没有安装 GCC 相关环境才需要执行
$ sudo apt-get install build-essential m4 autoconf automake make
$ sudo apt-get install libgd2-noxpm-dev libcurl4-openssl-dev libpcre3-dev
```

1.2 CentOS /RedHat / Fedora

```
# 请确保已经安装了 gcc automake autoconf m4
$ sudo yum install gd-devel pcre-devel libcurl-devel
```

1.3 FreeBSD / NetBSD / OpenBSD

```
# 不多说了，自己用 port 把 libcurl-dev libgd2-dev libpcre-dev 装上吧
# 编译前请确保已经安装 gcc automake autoconf m4
```

1.4 Windows

```
# 也支持的，不过要修改的代码太多了，包括 Nginx 本身，用 VC++来编译
# 嫌麻烦可以用 cygwin 来编译。还是不建议你这么做了，用 Unix/Linux 操作系统吧。
```

2. nginx / tengine 安装

选 Nginx 还是 Tengine,您自己看,两者选其一

2.1 下载 Tengine

```
# wget http://tengine.taobao.org/download/tengine-1.4.5.tar.gz
# tar -zxvf tengine-1.4.5.tar.gz
# cd tengine-1.4.5
```

2.2 下载 Nginx

```
# wget http://nginx.org/download/nginx-1.4.0.tar.gz
# tar -zxvf nginx-1.4.0.tar.gz
# cd nginx-1.4.0
```

2.3 安装模块

```
# wget https://github.com/3078825/nginx-image/archive/master.zip
# unzip master.zip
# ./configure --add-module=./nginx-image-master
# make
# make install
```

3. 配置

```
location / {
   root html;
   #添加以下配置
   image on;
   image_output on;
 }
```

4. 参数

image on/off 是否开启缩略图功能,默认关闭
image_backend on/off 是否开启镜像服务，当开启该功能时，请求目录不存在的图片（判断原图），将自动从镜像服务器地址下载原图
image_backend_server 镜像服务器地址
image_output on/off 是否不生成图片而直接处理后输出 默认 off
image_jpeg_quality 75 生成 JPEG 图片的质量 默认值 75
image_water on/off 是否开启水印功能
image_water_type 0/1 水印类型 0:图片水印 1:文字水印
image_water_min 300 300 图片宽度 300 高度 300 的情况才添加水印
image_water_pos 0-9 水印位置 默认值 9 0 为随机位置,1 为顶端居左,2 为顶端居中,3 为顶端居右,4 为中部居左,5 为中部居中,6 为中部居右,7 为底端居左,8 为底端居中,9 为底端居右
image_water_file 水印文件(jpg/png/gif),绝对路径或者相对路径的水印图片
image_water_transparent 水印透明度,默认 20
image_water_text 水印文字 “Power By Vampire”
image_water_font_size 水印大小 默认 5
image_water_font 文字水印字体文件路径
image_water_color 水印文字颜色,默认 #000000

调用说明

这里假设你的 nginx 访问地址为 http://127.0.0.1/
并在 nginx 网站根目录存在一个 test.jpg 的图片

通过访问

http://127.0.0.1/test.jpg!c300x200.jpg 将会 生成/输出 test.jpg 300×200 的缩略图
其中 c 是生成图片缩略图的参数， 300 是生成缩略图的 宽度 200 是生成缩略图的 高度

一共可以生成四种不同类型的缩略图。
支持 jpeg / png / gif (Gif 生成后变成静态图片)
C 参数按请求宽高比例从图片高度 10% 处开始截取图片，然后缩放/放大到指定尺寸（ 图片缩略图大小等于请求的宽高 ）
M 参数按请求宽高比例居中截图图片，然后缩放/放大到指定尺寸（ 图片缩略图大小等于请求的宽高 ）
T 参数按请求宽高比例按比例缩放/放大到指定尺寸（ 图片缩略图大小可能小于请求的宽高 ）
W 参数按请求宽高比例缩放/放大到指定尺寸，空白处填充白色背景颜色（ 图片缩略图大小等于请求的宽高 ）

5. 调用举例

http://127.0.0.1/test.jpg!c300x300.jpg
http://127.0.0.1/test.jpg!t300x300.jpg
http://127.0.0.1/test.jpg!m300x300.jpg
http://127.0.0.1/test.jpg!w300x300.jpg
http://127.0.0.1/test.c300x300.jpg
http://127.0.0.1/test.t300x300.jpg
http://127.0.0.1/test.m300x300.jpg
http://127.0.0.1/test.w300x300.jpg

6. 最后

这款模块的缩略图是实时生成的，如果你的网站流量比较大，势必会造成 nginx 服务器负载过高，针对这个问题，你可以参考我们运维生存时间之前写的几篇文章，分别为存硬盘和 redis. 《nginx 实时生成缩略图到硬盘上》《srcache_nginx+redis 构建缓存系统》

参考文章

项目地址：https://github.com/3078825/nginx-image/

48.nginx+set-misc-nginx-module 模块说明

set-misc-nginx-module 模块是标准的 HttpRewriteModule 指令的扩展，提供更多的功能，如 URI 转义与非转义、JSON 引述、Hexadecimal/MD5/SHA1/Base32/Base64 编码与解码、随机数等等。在后面的应用中，都将会接触使用到这个模块的。该模块是由章亦春先生开发的，他开发的其他模块应用也会使用到这个模块的。充分使用 nginx 非阻塞模式，对性能上有极大的提高，我个人认为很有必要去弄懂弄透 nginx，通过自己去动手动脑用实例来加以验证，不断的对模块参数命令加以理解，在实际环境中应用自如。某朋友说“招运维人员，不懂 nginx 的一律不要。”呵呵，有点道理的。

1. set-misc-nginx-module 模块指令说明：
set_if_empty
语法：set_if_empty $dst <src>;
默认值：no
配置段：location, location if
如果参数$dst 是空的，则赋值为<src>。
set $a 32;
set_if_empty $a 56;
$a 的值为 32.
set $a ” ;
set $value “hello, world”
set_if_empty $a $value;
$a 的值为”hello, world”。

set_quote_sql_str
语法：set_quote_sql_str $dst <src> / set_quote_sql_str $dst
默认值：no
配置段：location, location if
当两个参数时，该指令将引用第二个参数<src>值。该指令通常用于防止 SQL 注入。mysql 字符串值引用规则和分配第一个参数结果。

```
location /test {
 set $value “hello\n\r’ \” \\” ;
 set_quote_sql_str $quoted $value;
 echo $quoted;
}
```

结果为’ hello\n\r\’ \” \\’ 。
当是当个参数时，该指令将修改参数变量。如：

```
location /test {
 set $value “hello\n\r’ \” \\” ;
 set_quote_sql_str $value;
 echo $value;
}
```

结果为’ hello\n\r\’ \” \\’ 。

set_quote_pgsql_str
语法：set_quote_pgsql_str $dst <src> / set_quote_pgsql_str $dst
默认值：no
配置段： location, location if
与 set_quote_sql_str 相似，但是要符合 PostgreSQL 的 SQL 字符串常量的引用规则。

set_quote_json_str
语法：set_quote_json_str $dst <src> / set_quote_json_str $dst
默认值：no
配置段：location, location if
当两个参数时，该指令将引用第二个参数<src>。JSON 字符串值引用规则和分配第一个参数结果。

```
location /test {
 set $value “hello\n\r’ \” \\” ;
 set_quote_json_str $quoted $value;

 echo $quoted;
}
```

结果为：” hello\n\r’ \” \\” 。
当当个参数时，该指令将修改参数变量。如：

```
location /test {
 set $value “hello\n\r’ \” \\” ;
 set_quote_json_str $value;

 echo $value;
}
```

结果为：” hello\n\r’ \” \\” 。

set_unescape_uri
语法： set_unescape_uri $dst <src> / set_unescape_uri $dst
默认值： no
配置段：location, location if
当两个参数时，该指令将非转义第二个参数<src>的值作为 URI 一部分，并分配第一个参数变量$dst 分配结果。

如：

```
location /test {
 set_unescape_uri $key $arg_key;
 echo $key;
}
```

当请求 GET /test?key=hello+world%21 时，得到：hello world!。

注意： nginx 标准的变量$arg_PARAMETER 保存原始的 URI 参数(转义过的)，因此需要 set_unescape_uri 指令来非转义先。

当单个参数时，该指令将修改参数变量位置，如：

```
location /test {
set $key $arg_key;
set_unescape_uri $key;
echo $key;
}
```

当请求 GET /test?key=hello+world%21 时，得到：hello world!。

set_escape_uri

语法：set_escape_uri $dst <src> /set_escape_uri $dst

默认值：no

配置段：location, location if

与 set_unescape_uri 相似。

set_hashed_upstream

语法：set_hashed_upstream $dst <upstream_list_name> <src>

默认值：no

配置段：location, location if

参数<src> hash 后的值，对应于<upstream_list_name>中的某个 upstream 名称。

```
upstream moon { … }
upstream sun { … }
upstream earth { … }
upstream_list universe moon sun earth;
location /test {
 set_unescape_uri $key $arg_key;
 set $list_name universe;
 set_hashed_upstream $backend $list_name $key;
 echo $backend;
}
```

当请求/test?key=blah 时，得到的值是 “moon”， “sun”， “earth”其中一个。取决于参数 key。

set_encode_base32

语法：set_encode_base32 $dst <src> / set_encode_base32 $dst

默认值：no

配置段： location, location if

当两个参数时，该指令将对第二个参数<src>进行 base32(hex)编码，并将结果赋值给第一个变量参数$dst。 如：

```
location /test {
 set $raw “abcde”;
 set_encode_base32 $digest $raw;
 echo $digest;
}
```

当请求/test 时，得到：c5h66p35。

默认情况下，字符=用来左填充字节对齐。可以通过 set_misc_base32_padding off 来禁止填充。

当单个参数时，该指令将修改参数变量位置。如：

```
location /test {
 set $value “abcde”;
 set_encode_base32 $value;
 echo $value;
}
```

当请求/test 时，得到：c5h66p35。

set_misc_base32_padding

语法：set_misc_base32_padding on|off

默认值：on

配置段：http, server, server if, location, location if

当 set_encode_base32 指令以 base32 进行编码时，该指令控制是否以字符=来填充。

set_decode_base32

语法：set_decode_base32 $dst <src> | set_decode_base32 $dst

默认值：no

配置段： location, location if

与 set_encode_base32 相似，只不过是反过程。

set_encode_base64

语法： set_encode_base64 $dst <src> |set_encode_base64 $dst

默认值：no

配置段：location, location if

当两个参数时，该指令将对第二个参数<src>进行 base64 编码，并将结果赋值给第一个变量参数$dst。 如：

```
location /test {
 set $raw “abcde”;
 set_encode_base64 $digest $raw;
 echo $digest;
}
```

当请求/test 时，得到 YWJjZGU=。

当单个参数时，该指令将修改参数变量位置。如：

```
location /test {
 set $value “abcde”;
 set_encode_base64 $value;
 echo $value;
}
```

当请求/test 时，得到 YWJjZGU=。

set_decode_base64

语法：set_decode_base64 $dst <src> | set_decode_base64 $dst

默认值：no

配置段：location, location if

与 set_encode_base64 相似，只不过是反过程。

set_encode_hex

语法：set_encode_hex $dst <src> | set_encode_hex $dst

默认值：no

配置段： location, location if

当两个参数时，该指令将对第二个参数<src>进行 hexadecimal 编码，并将结果赋值给第一个变量参数$dst。

如：

```
location /test {
 set $raw “章亦春”;
 set_encode_hex $digest $raw;
 echo $digest;
}
```

当请求/test 时，得到：e7aba0e4baa6e698a5。

当单个参数时，该指令将修改参数变量位置。如：

```
location /test {
 set $value “章亦春”;
 set_encode_hex $value;
 echo $value;
}
```

当请求/test 时，得到：e7aba0e4baa6e698a5。

set_decode_hex

语法：set_decode_hex $dst <src> | set_decode_hex $dst

默认值：no

配置段： location, location if

与 set_encode_hex 相似，只不过是反过程。

set_sha1

语法：set_sha1 $dst <src> | set_sha1 $dst

默认值：no

配置段：location, location if

当两个参数时，该指令将对第二个参数<src>进行 SHA-1 编码，并将结果赋值给第一个变量参数$dst。 如：

```
location /test {
 set $raw “hello”;
 set_sha1 $digest $raw;
 echo $digest;
}
```

当请求/test，得到：aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d。

当单个参数时，该指令将修改参数变量位置。如：

```
location /test {
 set $value “hello”;
 set_sha1 $value;
 echo $value;
}
```

set_md5

语法：set_md5 $dst <src> | set_md5 $dst

默认值：no

配置段：location, location if

当两个参数时，该指令将对第二个参数<src>进行 MD5 编码，并将结果赋值给第一个变量参数$dst。 如：

```
location /test {
 set $raw “hello”;
 set_sha1 $digest $raw;
 echo $digest;
}
```

当请求/test，得到：5d41402abc4b2a76b9719d911017c592。

当单个参数时，该指令将修改参数变量位置。

如：

```
location /test {
 set $value “hello”;
 set_sha1 $value;
 echo $value;
}
```

set_hmac_sha1
语法：set_hmac_sha1 $dst <secret_key> <src> | set_hmac_sha1 $dst
默认值：no
配置段：location, location if
为参数<src>计算 HMAC-SHA1 值，将结果赋值给参数变量$dst 并带上密钥<secret_key>。如：

```
location /test {
 set $secret ‘thisisverysecretstuff’;
 set $string_to_sign ‘some string we want to sign’;
 set_hmac_sha1 $signature $secret $string_to_sign;
 set_encode_base64 $signature $signature;
 echo $signature;
}
```

请求/test，得到：R/pvxzHC4NLtj7S+kXFg/NePTmk=。
注意：该指令依赖于 OpenSSL 库，因此编译 nginx 时，需要 - with-http_ssl_module。

set_random
语法：set_random $res <from> <to>
默认值：no
配置段：location, location if
生成从<from>与<to>之间的非负数的随机数。含<from> <to>。

set_secure_random_alphanum
语法：set_secure_random_alphanum $res <length>
默认值：no
配置段：location, location if
生成长度为<length>的随机字符串。字符有[a-zA-Z0-9]。

set_secure_random_lcalpha
语法：set_secure_random_lcalpha $res <length>
默认值：no
配置段：location, location if
生成长度为<length>的随机字符串。字符有[a-z]。

set_rotate
语法：set_rotate $value <from> <to>
默认值：no
配置段：location, location if

set_local_today
语法：set_local_today $dst
默认值：no
配置段：location, location if
将本地的今天日期以”yyyy-mm-dd”格式赋值给参数变量$dst。

2. set-misc-nginx-module 安装

```
# ./configure --prefix=/usr/local/nginx \
--with-http_ssl_module \
--add-module=../ngx_devel_kit \
--add-module=../set-misc-nginx-module
# make
# make install
```

set-misc-nginx-module 依赖 ngx_devel_kit 模块，且 - add-module=ngx_devel_kit 一定要早于 - add-module=set-misc-nginx-module

49.nginx geo 使用方法

geo 指令使用 ngx_http_geo_module 模块提供的。默认情况下，nginx 有加载这个模块，除非人为的 - without-http_geo_module。

ngx_http_geo_module 模块可以用来创建变量，其值依赖于客户端 IP 地址。

geo 指令

语法: geo [$address] $variable { … }
默认值: —
配置段: http
定义从指定的变量获取客户端的 IP 地址。默认情况下，nginx 从$remote_addr 变量取得客户端 IP 地址，但也可以从其他变量获得。如

```
geo $remote_addr $geo {
        default 0;
        127.0.0.1 1;
}
geo $arg_ttlsa_com $geo {
        default 0;
        127.0.0.1 1;
}
```

如果该变量的值不能代表一个合法的 IP 地址，那么 nginx 将使用地址“255.255.255.255”。
nginx 通过 CIDR 或者地址段来描述地址，支持下面几个参数：
delete：删除指定的网络
default：如果客户端地址不能匹配任意一个定义的地址，nginx 将使用此值。 如果使用 CIDR，可以用“0.0.0.0/0”代替 default。
include： 包含一个定义地址和值的文件，可以包含多个。
proxy：定义可信地址。 如果请求来自可信地址，nginx 将使用其“X-Forwarded-For”头来获得地址。 相对于普通地址，可信地址是顺序检测的。
proxy_recursive：开启递归查找地址。 如果关闭递归查找，在客户端地址与某个可信地址匹配时，nginx 将使用“X-Forwarded-For”中的最后一个地址来代替原始客户端地址。如果开启递归查找，在客户端地址与某个可信地址匹配时，nginx 将使用“X-Forwarded-For”中最后一个与所有可信地址都不匹配的地址来代替原始客户端地址。
ranges：使用以地址段的形式定义地址，这个参数必须放在首位。为了加速装载地址库，地址应按升序定义。

```
geo $country {
    default         ZZ;
    include         conf/geo.conf;
    delete          127.0.0.0/16;
```

```
    proxy           192.168.100.0/24;
    proxy           2001:0db8::/32;

    127.0.0.0/24    US;
    127.0.0.1/32    RU;
    10.1.0.0/16     RU;
    192.168.1.0/24 UK;
}
vim conf/geo.conf
10.2.0.0/16     RU;
192.168.2.0/24 RU;
```

地址段例子：

```
geo $country {
    ranges;
    default                   ZZ;
    127.0.0.0-127.0.0.0       US;
    127.0.0.1-127.0.0.1       RU;
    127.0.0.1-127.0.0.255     US;
    10.1.0.0-10.1.255.255     RU;
    192.168.1.0-192.168.1.255 UK;
}
```

[warning]遵循最精确匹配原则，即 nginx 使用能最精确匹配客户端地址的值。[/warning]

适用实例

上面的例子几乎都是官网说明例子。下面举例说明便于理解该指令的用法。

1. 使用默认变量也就是$remote_addr

```
http {
    #geo $remote_addr $ttlsa_com {
    geo $ttlsa_com {
        default 0;
        127.0.0.1 1;
    }
    server {
        listen       8080;
        server_name  test.ttlsa.com;

        location /hello {
            default_type text/plain;
            echo $ttlsa_com;
            echo $arg_boy;
        }
    }
}
# curl 127.0.0.1:8080/hello?boy=默北
1
默北
```

2. 使用指定变量

```
http {
    geo $arg_boy $ttlsa_com {
        default 0;
        127.0.0.1 1;
        8.8.8.8 2;
}
    server {
        listen       8080;
        server_name  test.ttlsa.com;

        location /hello {
            default_type text/plain;
            echo $ttlsa_com;
            echo $arg_boy;
        }
    }
}
# curl 127.0.0.1:8080/hello?boy=8.8.8.8
2
8.8.8.8
```

3. 匹配原则

```
http {
    geo $arg_boy $ttlsa_com {
        default 0;
        127.0.0.1/24 24;
        127.0.0.1/32 32;
        8.8.8.8 2;
}
    server {
        listen       8080;
        server_name  test.ttlsa.com;

        location /hello {
            default_type text/plain;
            echo $ttlsa_com;
            echo $arg_boy;
        }
    }
}
# curl 127.0.0.1:8080/hello?boy=127.0.0.1
32
127.0.0.1
# curl 127.0.0.1:8080/hello?boy=127.0.0.12
24
127.0.0.12
```

[warning]geo 指令主要是根据 IP 来对变量进行赋值的。因此 geo 块下只能定义 IP 或网络段，否则会报错“nginx: [emerg] invalid network”。[/warning]

50.Nginx 与 Lua

这几个月里，我们逐步把 Lua 集成到 Mixlr 的前端 Nginx 配置中。
Lua 是一个可以嵌入到 Nginx 配置文件中的动态脚本语言，从而可以在 Nginx 请求处理的任何阶段执行各种 Lua 代码。刚开始我们只是用 Lua 把请求路由到后端服务器，但是它对我们架构的作用超出了我们的预期。下面就讲讲我们所做的工作。

强制搜索引擎只索引 mixlr.com

Google 把子域名当作完全独立的网站，我们不希望爬虫抓取子域名的页面，降低我们的 Page rank。

```
location / {
  header_filter_by_lua '
    if ngx.var.query_string and ngx.re.match( ngx.var.query_string, "^([0-9]{10})$" ) then
      ngx.header["Expires"] = ngx.http_time( ngx.time() + 31536000 );
      ngx.header["Cache-Control"] = "max-age=31536000";
    end
  ';
```

如果对 robots.txt 的请求不是 mixlr.com 域名的话，则内部重写到 robots_diallow.txt，虽然标准的重写指令也可以实现这个需求，但是 Lua 的实现更容易理解和维护。

根据程序逻辑设置响应头

Lua 提供了比 Nginx 默认配置规则更加灵活的设置方式。 在下面的例子中，我们要保证正确设置响应头，这样浏览器如果发送了指定请求头后，就可以 无限期缓存静态文件，是的用户只需下载一次即可。
这个重写规则使得任何静态文件，如果请求参数中包含时间戳值，那么就设置相应的 Expires 和 Cache-Control 响应头。

```
location / {
  header_filter_by_lua '
    if ngx.var.query_string and ngx.re.match( ngx.var.query_string, "^([0-9]{10})$" ) then
      ngx.header["Expires"] = ngx.http_time( ngx.time() + 31536000 );
      ngx.header["Cache-Control"] = "max-age=31536000";
    end
  ';


  try_files $uri @dynamic;}
```

删除 jQuery JSONP 请求的时间戳参数

很多外部客户端请求 JSONP 接口时，都会包含一个时间戳类似的参数，从而导致 Nginx proxy 缓存无法命中（因为无法忽略指定的 HTTP 参数）。下面的 规则删除了时间戳参数，使得 Nginx 可以缓存 upstream server 的响应内容，减轻后端服务器的负载。

```
location / {
  rewrite_by_lua '
    if ngx.var.args ~= nil then
      -- /some_request?_=1346491660 becomes /some_request
      local fixed_args, count = ngx.re.sub( ngx.var.args, "&?_=[0-9]+", "" );
      if count > 0 then
        return ngx.exec(ngx.var.uri, fixed_args);
      end
    end
  ';}
```

把后端的慢请求日志记录到 Nginx 的错误日志

如果后端请求响应很慢，可以把它记录到 Nginx 的错误日志，以备后续追查。

```
location / {
  log_by_lua '
    if tonumber(ngx.var.upstream_response_time) >= 1 then
      ngx.log(ngx.WARN, "[SLOW] Ngx upstream response time: " .. ngx.var.upstream_response_time ..
"s from " .. ngx.var.upstream_addr);
    end
  ';}
```

基于 Redis 的实时 IP 封禁

某些情况下，需要阻止流氓爬虫的抓取，这可以通过专门的封禁设备去做，但是通过 Lua，也可以实现简单版本的封禁。

```
lua_shared_dict banned_ips 1m;

location / {
  access_by_lua '
    local banned_ips = ngx.shared.banned_ips;
    local updated_at = banned_ips:get("updated_at");

    -- only update banned_ips from Redis once every ten seconds:
    if updated_at == nil or updated_at < ( ngx.now() - 10 ) then
      local redis = require "resty.redis";
      local red = redis:new();
      red:set_timeout(200);

      local ok, err = red:connect("your-redis-hostname", 6379);
      if not ok then
        ngx.log(ngx.WARN, "Redis connection error retrieving banned_ips: " .. err);
      else
        local updated_banned_ips, err = red:smembers("banned_ips");
        if err then
          ngx.log(ngx.WARN, "Redis read error retrieving banned_ips: " .. err);
        else
          -- replace the locally stored banned_ips with the updated values:
          banned_ips:flush_all();
          for index, banned_ip in ipairs(updated_banned_ips) do
            banned_ips:set(banned_ip, true);
          end
          banned_ips:set("updated_at", ngx.now());
        end
      end
    end

    if banned_ips:get(ngx.var.remote_addr) then
      ngx.log(ngx.WARN, "Banned IP detected and refused access: " .. ngx.var.remote_addr);
      return ngx.exit(ngx.HTTP_FORBIDDEN);
    end
  ';}
```

现在就可以阻止特定 IP 的访问：

```
ruby> $redis.sadd("banned_ips", "200.1.35.4")
```

Nginx 进程每隔 10 秒从 Redis 获取一次最新的禁止 IP 名单。需要注意的是，如果架构中使用了 Haproxy 这样类似的负载均衡服务器时， 需要把$remote_addr 设置为正确的远端 IP 地址。
这个方法还可以用于 HTTP User-Agent 字段的检查，要求满足指定条件。

使用 Nginx 输出 CSRF(form authenticity_token)

Mixlr 大量使用页面缓存，由此引入的一个问题是如何给每个页面输出会话级别的 CSRF token。我们通过 Nginx 的子请求，从 upstream web server 获取 token，然后利用 Nginx 的 SSI(server-side include)功能输出到页面中。这样既解决了 CSRF 攻击问题，也保证了 cache 能被正常利用。

```
location /csrf_token_endpoint {
  internal;

  include /opt/nginx/conf/proxy.conf;
  proxy_pass "http://upstream";}

location @dynamic {
  ssi on;
  set $csrf_token '';

  rewrite_by_lua '
    -- Using a subrequest, we our upstream servers for the CSRF token for this session:
    local csrf_capture = ngx.location.capture("/csrf_token_endpoint");
    if csrf_capture.status == 200 then
      ngx.var.csrf_token = csrf_capture.body;

      -- if this is a new session, ensure it sticks by passing through the new session_id
      -- to both the subsequent upstream request, and the response:
      if not ngx.var.cookie_session then
        local match = ngx.re.match(csrf_capture.header["Set-Cookie"], "session=([a-zA-Z0-
9_+=/+]+);");
        if match then
          ngx.req.set_header("Cookie", "session=" .. match[1]);
          ngx.header["Set-Cookie"] = csrf_capture.header["Set-Cookie"];
        end
      end
    else
      ngx.log(ngx.WARN, "No CSRF token returned from upstream, ignoring.");
    end
  ';

  try_files /maintenance.html /rails_cache$uri @thin;}
CSRF token 生成 app/metal/csrf_token_endpoint.rb:
class CsrfTokenEndpoint
  def self.call(env)
    if env["PATH_INFO"] =~ /^\/csrf_token_endpoint/
      session = env["rack.session"] || {}

      token = session[:_csrf_token]
      if token.nil?
        token = SecureRandom.base64(32)
```

```
        session[:_csrf_token] = token
      end

      [ 200, { "Content-Type" => "text/plain" }, [ token ] ]
    else
      [404, {"Content-Type" => "text/html"}, ["Not Found"]]
    end
  end
end
```

我们的模版文件示例:

```
<meta name=" csrf-param" value=" authenticity_token" />
<meta name=" csrf-token" value="<!-# echo var=" csrf_token" default=" " encoding=" none"
->" />
```

Again you could make use of lua_shared_dict to store in memory the CSRF token for a particular session. This minimises the number of trips made to /csrf_token_endpoint.

原文链接：http://devblog.mixlr.com/2012/09/01/nginx-lua/

51.ngx_http_headers_module 模块 add_header 和 expires 指令

一. 前言

ngx_http_headers_module 模块提供了两个重要的指令 add_header 和 expires，来添加 “Expires” 和 “Cache-Control” 头字段，对响应头添加任何域字段。add_header 可以用来标示请求访问到哪台服务器上，这个也可以通过 nginx 模块 nginx-http-footer-filter 研究使用来实现。expires 指令用来对浏览器本地缓存的控制。

二. add_header 指令

语法: add_header name value;
默认值: —
配置段: http, server, location, if in location
对响应代码为 200，201，204，206，301，302，303，304，或 307 的响应报文头字段添加任意域。如:

```
add_header From ttlsa.com
```

三. expires 指令

语法: expires [modified] time;
expires epoch | max | off;
默认值: expires off;
配置段: http, server, location, if in location
在对响应代码为 200，201，204，206，301，302，303，304，或 307 头部中是否开启对 “Expires” 和 “Cache-Control” 的增加和修改操作。
可以指定一个正或负的时间值，Expires 头中的时间根据目前时间和指令中指定的时间的和来获得。

epoch 表示自 1970 年一月一日 00:00:01 GMT 的绝对时间，max 指定 Expires 的值为 2037 年 12 月 31 日 23:59:59，Cache-Control 的值为 10 years。
Cache-Control 头的内容随预设的时间标识指定:

- 设置为负数的时间值:Cache-Control: no-cache。
- 设置为正数或 0 的时间值：Cache-Control: max-age = #，这里#的单位为秒，在指令中指定。

参数 off 禁止修改应答头中的” Expires” 和” Cache-Control” 。

实例一：对图片，flash 文件在浏览器本地缓存 30 天

```
location ~ .*\.(gif|jpg|jpeg|png|bmp|swf)$
 {
           expires 30d;
 }
```

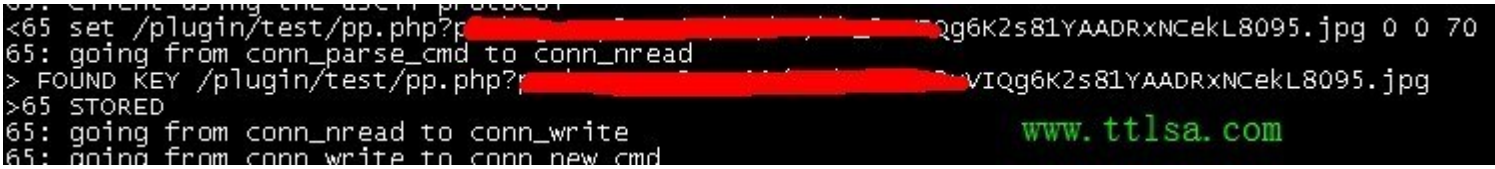
实例二：对 js，css 文件在浏览器本地缓存 1 小时

```
location ~ .*\.(js|css)$
 {
            expires 1h;
 }
```

常见问题

memc_nginx+srcache_nginx+memcached 遇到的问题

在使用 memc_nginx+srcache_nginx+memcached 时，出现了 memcached 只缓存响应头，响应主体却丢失了。具体文章参见：《memc_nginx+srcache_nginx+memcached 构建透明的动态页面缓存》。



缓存的内容被截断了。

查了些文档是由 upstream 响应截断时不报错导致的。

解决办法：

1. 打上下面的补丁

```
https://github.com/agentzh/ngx_openresty/blob/master/patches/nginx-1.4.2-upstream_truncation.patch
```

2. 使用最新版的 nginx

nginx 在 1.5.3 版本修复了这个 bug。

nginx 反向代理 proxy_set_header 自定义 header 头无效

公司使用 nginx 作为负载均衡，有时候需要自定义 header 头发送给后端的真实服务器. 想过去应该是非常的简单的事情.

例子如下：
设置代理服务器 ip 头

```
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
```

然后自己在自定义个 header，remote_header_test，如下：

```
proxy_set_header remote_header_test "123123123";
```

接着后端真实服务器打开 www.ttlsa.com/nginx_header.php
源代码是简单的 phpinfo

```
<?php
 phpinfo();
?>
```

在 phpinfo 结果页面中搜索刚才设置的头部，发现没有找到,网上查找资料,才发现原来 nginx 会忽略掉下划线的头部变量.于是改成如下：

```
proxy_set_header remoteheadertest "123123123";
```

再次打开 www.ttlsa.com/nginx_header.php,搜索 remoteheadertest,有内容. 看来果真不能用下划线. 然后改成’-‘，如下：

```
proxy_set_header remote-header-test "123123123";
```

打开页面,搜索到的头部是 remote_header_test. 自动转换成下划线了.

如果想要支持下划线的话，需要增加如下配置：

```
underscores_in_headers on;
```

可以加到 http 或者 server 中

语法：underscores_in_headers on|off
默认值：off
使用字段：http, server
是否允许在 header 的字段中带下划线

nginx purge 更新缓存 404 错误

nginx 默认安装就会带有反向代理的功能，但想要更好的使用，还得配备 frickle.com 的 ngx_cache_purge 模块，用于清除指定 URL 的缓存。ngx_cache_purge 在安装的 nginx 的时候一起编译进去了，缓存功能一直正常。
文件地址：www.abc.com/includes/templates/zcen/buttons/english/button_in_cart.gif
如下图：

nginx purge 清理缓存失败

但是清理缓存的时候竟然会 404
地址：www.abc.com/purge/includes/templates/zcen/buttons/english/button_in_cart.gif

nginx purge 清理缓存失败

百思不得其解,网上遇到 nginx 清理缓存出现 404 的用户不在少数，网上一共有如下 3 中情况：
1、 ngx_cache_purge 版本与 nginx 版本不匹配
换了一个版本的 purge，发现依旧无效

2、 nginx 启动方法不对
很多人安装完 nginx，仅仅 reload 一次 nginx，实际上应该 stop 之后在 start。这不是我的解决方法。

3、 purge 未编译到 nginx 中
肯定不是这个问题，nginx -V 能查看编译参数

因为有其他事情，这个事情暂且搁置了，一天闲来无事，看着 nginx 的配置文件发呆，突然发现自己犯了一个很大的错误：purge 的 location 放错了位置。

错误配置文件：

```
location /
{
proxy_pass http://xxx.ttlsa.com;
include proxy.conf;
}
location ~ .*\.(png|jpg|gif|GIF|jpeg|JPG|PNG|bmp|BMP|JPEG)?$
{
include proxy.conf;
proxy_pass http://xxx.ttlsa.com;
expires      1h;
access_log off;
}
location ~ /purge(/.*)
{
allow            127.0.0.1;
allow            192.168.12.0/24;
proxy_cache_purge    cache_one   $host$1$is_args$args;
}
```

正确配置文件：

```
location /
{
  proxy_pass http://xxx.ttlsa.com;
  include proxy.conf;
}
location ~ /purge(/.*)
{
  allow            127.0.0.1;
  allow            192.168.12.0/24;
  proxy_cache_purge    cache_one   $host$1$is_args$args;
}
location ~ .*\.(png|jpg|gif|GIF|jpeg|JPG|PNG|bmp|BMP|JPEG)?$
{
  include proxy.conf;
  proxy_pass http://xxx.ttlsa.com;
  expires      1h;
  access_log off;
}
```

细心的兄弟很快能发现我把 purge 的位置放错了，每次更新图片缓存的时候它都只匹配到了图片后缀的 location，接着就返回了 404，根本没有匹配到 purge 这个 location 的机会。把 purge 调到前面就正常了。