Nginx 应用技术指南

【前言】：

编写此技术指南在于推广普及 NGINX 在国内的使用，更方便的帮助大家了解和掌握 NGINX 的一些使用技巧。本指南很多技巧来自于网络在此对网络上愿意分享的朋友们表示感谢和致意！欢迎大家和我一起丰富本技术指南并提出更好的建议！

一、Nginx 基础知识

1、 简介

Nginx ("engine x") 是一个高性能的 HTTP 和 反向代理 服务器，也是一个 IMAP/POP3/SMTP 代理服务器。 Nginx 是由 Igor Sysoev 为俄罗斯访问量第二的 Rambler.ru 站点开发的，它已经在该站点运行超过两年半了。Igor 将源代码以类BSD许可证的形式发布。尽管还是测试版，但是，Nginx 已经因为它的稳定性、丰富的功能集、示例配置文件和低系统资源的消耗而闻名了。更多的请见官方 wiki: http://wiki.codemongers.com/NginxChs

2、Nginx 的优点

Nginx 做为 HTTP 服务器，有以下几项基本特性：

1.处理静态文件，索引文件以及自动索引；打开文件描述符缓冲.

2.无缓存的反向代理加速，简单的负载均衡和容错.

3、 FastCGI，简单的负载均衡和容错.

4、 模块化的结构。

包括 gzipping, byte ranges, chunked responses, 以及 SSI-filter 等 filter。如果由 FastCGI 或其它代理服务器处理单页中存在的多个 SSI，则这项处理可以并行运行，而不需要相互等待。

5、 支持 SSL 和 TLS SNI.

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

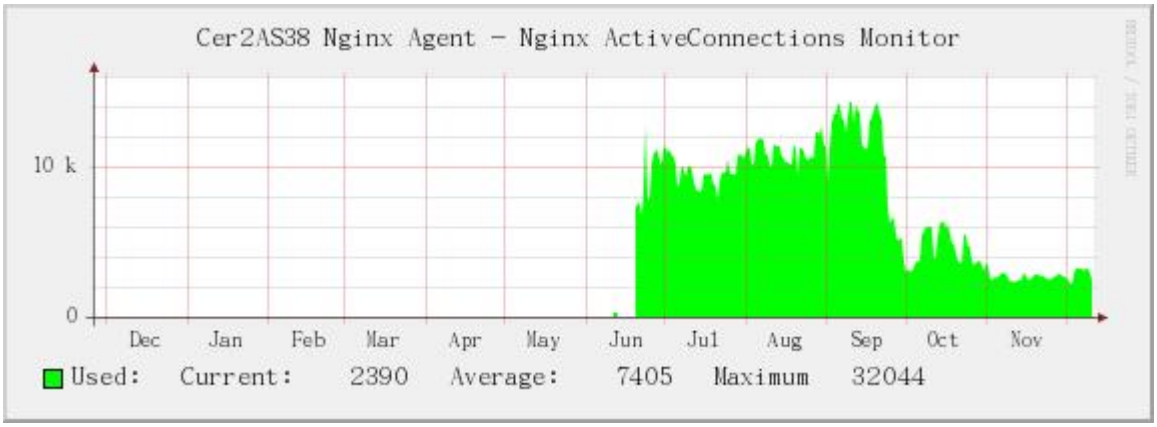
Nginx 具有很高的稳定性。其它 HTTP 服务器，当遇到访问的峰值，或者有人恶意发起慢速连接时，也很可能会导致服务器物理内存耗尽频繁交换，失去响应，只能重启服务器。例如当前 apache 一旦上到 200 个以上进程，web 响应速度就明显非常缓慢了。而 Nginx 采取了分阶段资源分配技术，使得它的 CPU 与内存占用率非常低。Nginx 官方表示保持 10,000 个没有活动的连接，它只占 2.5M 内存，所以类似 DOS 这样的攻击对 Nginx 来说基本上是毫无用处的。就稳定性而言, Nginx 比 lighthttpd 更胜一筹。

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

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

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

收完后端服务器的整个请求之前把压缩流转向客户端。
Nginx 采用了一些 os 提供的最新特性如对 sendfile (Linux 2.2+)，accept-filter (FreeBSD 4.1+)，TCP_DEFER_ACCEPT (Linux 2.4+) 的支持，从而大大提高了性能。



二、 Nginx 安装配置

1、安装 pcre

```
./configure

 make && make install

 cd ../
```

2、Nginx 编译安装

```
./configure --user=www --group=www --prefix=/usr/local/Nginx/ --with-http_stub_status_module
--with-openssl=/usr/local/openssl

make && make install
```

更详细的模块定制与安装请参照官方 wiki.

3、Nginx 配置文件测试：

```
[root@Chinarenservice ~]# /usr/local/nginx/sbin/nginx -t
2008/12/16 09:08:35 [info] 28412#0: the configuration file /usr/local/nginx/conf/nginx.conf syntax is ok
2008/12/16 09:08:35 [info] 28412#0: the configuration file /usr/local/nginx/conf/nginx.conf was tested successfully
```

4、Nginx 启动：

```
[root@Chinarenservice ~]# /usr/local/nginx/sbin/nginx
```

5、Nginx 配置文件修改重新加载：

```
[root@Chinarenservice ~]# kill -HUP `cat /usr/local/nginx/logs/nginx.pid`
```

三、Nginx 编译优化

1、 GCC 参数：

默认 Nginx 使用的 GCC 编译参数是-O，需要更加优化可以使用以下两个参数
–with-cc-opt=’-O3′ \
–with-cpu-opt=opteron \
使得编译针对特定 CPU 以及增加 GCC 的优化，针对优化后的结果.我们进行测试结果表明使用-O2 以及以上的参数,可以微量增加性能 1%左右，而 O2 和 O3 基本可以认为是相同的：
./http_load -parallel 100 -seconds 10 urls
10811 fetches, 100 max parallel, 5.23252e+06 bytes, in 10 seconds

a、默认参数 -O

1087.2 fetches/sec, 526204 bytes/sec
msecs/connect: 45.5374 mean, 63.984 max, 1.008 min
msecs/first-response: 45.7679 mean, 64.201 max, 2.216 min
1088.9 fetches/sec, 527027 bytes/sec
msecs/connect: 45.0159 mean, 65.291 max, 0.562 min
msecs/first-response: 46.1236 mean, 67.397 max, 9.169 min
1102.2 fetches/sec, 533465 bytes/sec
msecs/connect: 44.5593 mean, 67.649 max, 0.547 min
msecs/first-response: 45.499 mean, 67.849 max, 2.495 min

B、优化编译后 -O2

1081.1 fetches/sec, 523252 bytes/sec
msecs/connect: 45.7144 mean, 63.324 max, 0.823 min
msecs/first-response: 46.1008 mean, 61.814 max, 4.487 min
1110.2 fetches/sec, 537337 bytes/sec
msecs/connect: 43.4943 mean, 60.066 max, 0.715 min
msecs/first-response: 45.756 mean, 62.076 max, 3.536 min
1107 fetches/sec, 535788 bytes/sec
msecs/connect: 44.872 mean, 3036.51 max, 0.609 min
msecs/first-response: 44.8625 mean, 59.831 max, 3.178 min

C、优化编译后 -O3

1097.5 fetches/sec, 531189 bytes/sec
msecs/connect: 45.1355 mean, 3040.24 max, 0.583 min
msecs/first-response: 45.3036 mean, 68.371 max, 4.416 min
1111.6 fetches/sec, 538014 bytes/sec
msecs/connect: 44.2514 mean, 64.831 max, 0.662 min
msecs/first-response: 44.8366 mean, 69.904 max, 3.928 min
1099.4 fetches/sec, 532109 bytes/sec
msecs/connect: 44.7226 mean, 61.445 max, 0.596 min
msecs/first-response: 45.4883 mean, 287.113 max, 3.336 min

2、 修改 Nginx 的 header 伪装服务器

```
# cd Nginx-0.6.31

# vi src/core/Nginx.h

#ifndef _NGINX_H_INCLUDED_
```

```
#define _NGINX_H_INCLUDED_
#define NGINX_VERSION      "7.2"
#define NGINX_VER          "Freeke/" NGINX_VERSION
#define NGINX_VAR          "NGINX"
#define NGX_OLDPID_EXT     ".oldbin"
#endif /* _NGINX_H_INCLUDED_ */


# curl -I www.chinarenservice.com

HTTP/1.1 200 OK
Server: Freeke/7.2
Date: Mon, 24 Nov 2008 02:42:51 GMT
Content-Type: text/html; charset=gbk
Transfer-Encoding: chunked
Connection: keep-alive
```

3、 Tcmalloc 优化 Nginx 性能

从 Nginx 0.6.29 添加 Feature: the **ngx_google_perftools_module**，那 Nginx 也可以利用 Tcmalloc 来提长性能。

```
[root@chinarenservice ~]# wget http://download.savannah.gnu.org/releases/libunwind/libunwind-0.99-alpha.tar.gz
[root@chinarenservice ~]# tar zxvf libunwind-0.99-alpha.tar.gz
[root@chinarenservice ~]# cd libunwind-0.99-alpha/
[root@chinarenservice ~]# CFLAGS=-fPIC ./configure
[root@chinarenservice ~]# make CFLAGS=-fPIC
[root@chinarenservice ~]# make CFLAGS=-fPIC install
[root@chinarenservice ~]# wget http://google-perftools.googlecode.com/files/google-perftools-0.98.tar.gz
[root@chinarenservice ~]# tar zxvf google-perftools-0.98.tar.gz
[root@chinarenservice ~]# cd google-perftools-0.98/
[root@chinarenservice ~]# ./configure
[root@chinarenservice ~]# make && make install
[root@chinarenservice ~]# echo "/usr/local/lib" > /etc/ld.so.conf.d/usr_local_lib.conf
[root@chinarenservice ~]# ldconfig
[root@ chinarenservice local]# lsof -n | grep tcmalloc
Nginx      7323     root    mem       REG        8,2     1412859     440730
/usr/local/lib/libtcmalloc.so.0.0.0
Nginx      7324      www    mem       REG        8,2     1412859     440730
/usr/local/lib/libtcmalloc.so.0.0.0
Nginx      7325      www    mem       REG        8,2     1412859     440730
/usr/local/lib/libtcmalloc.so.0.0.0
Nginx      7326      www    mem       REG        8,2     1412859     440730
/usr/local/lib/libtcmalloc.so.0.0.0
Nginx      7327      www    mem       REG        8,2     1412859     440730
/usr/local/lib/libtcmalloc.so.0.0.0
```

Nginx	7328	www	mem	REG	8,2	1412859	440730 /usr/local/lib/libtcmalloc.so.0.0.0
Nginx	7329	www	mem	REG	8,2	1412859	440730 /usr/local/lib/libtcmalloc.so.0.0.0
Nginx	7330	www	mem	REG	8,2	1412859	440730 /usr/local/lib/libtcmalloc.so.0.0.0
Nginx	7331	www	mem	REG	8,2	1412859	440730 /usr/local/lib/libtcmalloc.so.0.0.0

在编译 Nginx 时添加参数--with-google_perftools_module

4、 减小编译后文件大小：

默认的 Nginx 编译选项里居然是用 debug 模式(-g)的（debug 模式会插入很多跟踪和 ASSERT 之类），编译以后一个 Nginx 有好几兆。去掉 Nginx 的 debug 模式编译，编译以后只有 375K（Nginx-0.5.33, gcc4）。在 auto/cc/gcc，最后几行有：

```
# debug
CFLAGS=”$CFLAGS -g”
```

注释掉或删掉这几行，重新编译即可。-g

四、Nginx 根据 URL 分发

1、第一种方法：

使用 NginxHttpUpstreamRequestHashModule 的方式，增加或减少机器时所引起的 hash 全部错乱的问题还是很令人担心，所以经过一段时间细致思考，觉得由自己手工制定并实现 url hash 规则，然后利用 Nginx 的 location 标签或 if 语法来实现来得更为灵活，可操作性和可用性会大大加强。不过配置就稍显复杂了，也需要程序方面的支持。

使用这种环境，主要需要考虑链接形式，链接形式不能够是/xxx.jsp?id=1 这样的带有?的，否则处理起来会很复杂，需要使用 rewrite 将这种形式的 url 变化成/freeke/1.html，其中加一级目录的目的是可以利用到 location 标签。如果是纯静态页或图片，一般都会有自成的目录规则。

首先我们制定一个链接的划分规则，这个规则有点区别于文件目录的划分规则，它本身并不需要考虑文件夹内文件数目的多少，制定这个规则的目的是容纳足够多的服务器！一般来说，如果 id 是字符型的，只需要分出 26 个字母 10 个数字，能够容纳 36 台 cache 服务器，这已经很足够了。如果是数字型的 id，那就拿数字 id╪0，就可以支持 100 台 cache，已经足够夸张。一般说来，╪，支持 10 台服务器应该就足够了，配置也容易一点。

有了这个目录规则，就可以通过 Nginx 的语法来书写配置了。

首先制定一堆 upstream，如果是偶数，理论能够分得更均衡。

```
upstream freeke1 {
server 10.0.0.1;
}
upstream freeke2 {
server 10.0.0.2;
}
```

```
#配置一个 all 的目的是兼容不进行 hash 的剩余的页面，比如首页
upstream chinarenservice {
server 10.0.0.1;
server 10.0.0.2;
}
```

1、已经按/a/划好的，使用 location 划分

```
location ~* /a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r/ {
proxy_pass http://freeke1;
}
location ~* /s|t|u|v|w|x|y|z|0|1|2|3|4|5|6|7|8|9|0/ {
proxy_pass http://freeke2;
}
location / {
proxy_pass http://chinarenservice;
}
```

2、已经有一定链接规则，不想变化，可以使用 if 语句判断

```
location / {
proxy_pass http://chinarenservice;
if  ( $request_uri ~* /page_(a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r)/ ) {
proxy_pass http://freeke1;
}
if  ( $request_uri ~* /page_(s|t|u|v|w|x|y|z|0|1|2|3|4|5|6|7|8|9|0)/ ) {
proxy_pass http://freeke2;
}
}
```

在新增服务器后，需要改动配置，手工将一些目录规则的文件分出去，剩下的仍然访问原先的服务器，不会造成太严重影响。在这种配置下，像首页这样的访问量大的单页，它访问量大，容量小，所以不可能产生容灾问题。使用轮循的方式工作，会比原先分到死定一台 cache，在可用性上要好得多。

2、第二种方法：

使用第三方的插件来实现：

五、Nginx Rewrite

1.Nginx Rewrite 基本标记(flags)

last - 基本上都用这个 Flag。
break - 中止 Rewirte，不在继续匹配
redirect - 返回临时重定向的 HTTP 状态 302
permanent - 返回永久重定向的 HTTP 状态 301

2. 正则表达式匹配，

其中：

* ~ 为区分大小写匹配

* ~* 为不区分大小写匹配

* !~和!~*分别为区分大小写不匹配及不区分大小写不匹配

3. 文件及目录匹配，

* -f 和!-f 用来判断是否存在文件

* -d 和!-d 用来判断是否存在目录

* -e 和!-e 用来判断是否存在文件或目录

* -x 和!-x 用来判断文件是否可执行

4.Nginx 的一些可用的全局变量，可用做条件判断：

$args

$content_length

$content_type

$document_root

$document_uri

$host

$http_user_agent

$http_cookie

$limit_rate

$request_body_file

$request_method

$remote_addr

$remote_port

$remote_user

$request_filename

```
$request_uri

$query_string

$scheme

$server_protocol

$server_addr

$server_name

$server_port

$uri
```

六、Nginx Redirect

将所有chinarenservice.com与abc.chinarenservice.com域名全部自跳转到http://www.chinarenservice.com

```
server {
        listen          80;
        server_name  chinarenservice.com abc.chinarenservice.com;
        index index.html index.php;
        root  /var/InfiNET/web/;
        if ($http_host !~ "^www\.linxtone\.org$") {
                rewrite  ^(.*)      [url]http://www.chinarenservice.com$1 redirect;
          }

        .....................
    }
```

七、Nginx 目录自动加斜线:

```
  if (-d $request_filename){
        rewrite ^/(.*)([^/])$ http://$host/$1$2/ permanent;
    }
```

八、Nginx 防盗链

```
#Preventing hot linking of images and other file types
location ~* ^.+\.(gif|jpg|png|swf|flv|rar|zip)$ {
      valid_referers none blocked server_names *.chinarenservice.com http://localhost baidu.com;
if ($invalid_referer) {
     rewrite    ^/    [img]http://www.chinarenservice.com/images/default/logo.gif[/img];
```

```
    # return    403;
     }
}
```

九、Nginx expires

1、根据文件类型 expires

```
# Add expires header for static content
location ~* \.(js|css|jpg|jpeg|gif|png|swf)$ {

    if (-f $request_filename) {
       root /var/InfiNET/web/bbs;
       expires       1d;
       break;
    }
}
```

2、根据判断某个目录

```
# serve static files
location ~ ^/(images|javascript|js|css|flash|media|static)/  {
root      /var/InfiNET/web/down;
       expires 30d;
 }
```

十、Nginx 访问控制

1、Nginx 身份证验证

```
#cd /usr/local/Nginx/conf
#mkdir htpasswd
/usr/local/apache2/bin/htpasswd -c /usr/local/Nginx/conf/htpasswd/freeke chinarenservice #添加用户名为
chinarenservice
New password:    (此处输入你的密码)
Re-type new password:    (再次输入你的密码)
Adding password for user

http://count.chinarenservice.com/freeke/data/index.html(目录存在/var/InfiNET/web/freeke/data/目录下)

将下段配置放到虚拟主机目录，当访问 http://count.chinarenservice/freeke/即提示要密验证:

location ~ ^/(freeke)/  {
                root      /var/InfiNET/web/count;
                       auth_basic                "LT-COUNT-Freeke";
```

```
                auth_basic_user_file  /usr/local/Nginx/conf/htpasswd/freeke;
        }
```

2、Nginx 禁止访问某类型的文件

如，Nginx 下禁止访问*.txt 文件，配置方法如下.代码:

```
location ~* \.(txt|doc)$ {
   if (-f $request_filename) {
   root /var/InfiNET/web/chinarenservice/test;
   break;
   }
}
```

方法 2:

```
location ~* \.(txt|doc)${
        root /var/InfiNET/web/chinarenservice/test;
        deny all;
}
```

禁止下载以点开头的文件：如 .freeke；.dat；.exe

```
location ~ /\..+ {
deny all;
}
```

禁止访问某个目录

```
location ~ ^/(WEB-INF)/ {
            deny all;
}
```

3、使用 ngx_http_access_module 限制 ip 访问

```
location / {
    deny     192.168.1.1;
    allow    192.168.1.0/24;
    allow    10.1.1.0/16;
    deny     all;
}
```

详细参见wiki: http://wiki.codemongers.com/NginxHttpAccessModule#allow

4、Nginx 下载限制并发和速率

```
limit_zone    one  $binary_remote_addr  10m;
server
```

```
    {
        listen       80;
        server_name  down.chinarenservice.com;
        index index.html index.htm index.php;
        root   /var/InfiNET/web/down;
        #Zone limit
        location / {
            limit_conn   one  1;
            limit_rate  20k;
        }
.....
    }
```

5、大文件上传限制

上传碰到“413 Request Entity Too Large”错误。只想让 Nginx 可以处理 1M 以上的文件上传，可以更改设置（例如允许 30M 的文件上传）：在 Nginx.conf 的 http{}中增加

client_max_body_size 30m

同时要修改 php.ini 中相关参数

［Resource Limits］

max_execution_time = 800 ; Maximum execution time of each script, in seconds 由于上传大文件比较费时，所以 max_execution_time 设为 800,默认是 30

［Data Handling］

; Maximum size of POST data that PHP will accept.修改下面的数值，以增大上传文件大小,默认是 8M

post_max_size = 20M

［File Uploads］

; Maximum allowed size for uploaded files.修改下面的数值，以增大上传文件大小,默认是 2M

upload_max_filesize = 20M

6、Nginx 实现 Apache 一样目录列表

```
location / {
    autoindex  on;
}
```

7、 http_accesskey_module 模块应用：

问题情况：

相册或者文章等资源，需要设置多种访问的控制，比如：好友可见、好友中的单个分组可见等。

解决办法：

利用 nginx 的几个 mod 结合来实现，相关的 mod 有：http_accesskey_module、mod_parsed_vars。

http_accesskey_module 估名思意是用来做限制访问的，通过传递的 key 检测来实现访问控制。mod_parsed_vars 模块是用于得到生成 key 相关参数的模块，比如说 http_accesskey_module 中的 key 是依据 COOKIE/GET/POST

中的一个或者多个值来生成的，那么我们需要让 nginx 获取到这些具体的值，而 nginx 本身是不具备的，mod_parsed_vars 起的就是这个作用。

我们要来实现访问控制的思路很简单，当用户登录后，检测到对资源的访问权限后，如果具有访问权限，那么依据资源的关键字、COOKIE/GET/POST 中的一个可依赖的值、一个服务器端的密钥来生成一个用于 http_accesskey_module 模块来进行访问限制检查的 KEY。

举例如下：

当用户登录后 COOKIE 中会有 SESSIONID，假如用户要访问图片编号为 PID 的图片，图片地址为:http://image.chinarenservice.com/PID.jpg。服务器端的密钥为 PKEY,这个时候用于 http_accesskey_module 的 KEY 的生成方法则可以简单为:KEY = md5(SESSIONID + PID + PKEY).那么实现访问的 URL 则为：http://image.chinarenservice.com/PID.jpg?key=KEY.同时我们在 nginx 中做相应的配置，来对图片的访问进行控制，就能达到限制访问的目的了。

相关配置参考：

测试用的 nginx 版本为 0.7.13.

1、下载 http_accesskey_module 和 mod_parsed_vars 两个模块。（如果你需要，可以找我。）

2、执行下面的一些命令，不做解释了。

```
cd nginx-0.7.13
bzcat ../nginx-accesskey-2.0.3.diff.bz2 | patch -p1
patch -p0 < /freeke/software/mod_parsed_vars/ngx_http_parsed_variables.patch
./configure –with-http_accesskey_module  –add-module=/freeke/software/mod_parsed_vars
make && make install
```

3、修改 nginx 的配置文件限制访问,注：密钥为&%*&*)(),使用的加密参数项为：GET 中的 pid 的值。

```
server {
listen       81;
root   /var/InfiNET/www/images;
access_log   /usr/local/nginx/logs/host.access.log   main;
location / {
accesskey            on;
accesskey_hashmethod   md5;
accesskey_arg        “key”;
accesskey_signature   “&%*&*)()$args_pid”;
index   index.html index.htm index.php;
}
}
```

4、在你的程序中生成访问的 URL(PHP 代码)。

```
$pkey = ‘&%*&*)()’;
$pid = ‘1314′;
$access_key = md5($pkey . $pid);
echo ‘<a href=”http://www.chinarenservice.com/’ . $pid . ‘.jpg?key=’ . $access_key .’”>查看图片</a>’;
```

到此为此我们基本实现的基于应用的权限控制，且方法灵活，因为 KEY 的生成由应用程序来控制，能进行任意的修改和变动，而架构不受影响。

十一、Nginx Location

1. 基本语法:

[和上面 rewrite 正则匹配语法基本一致]

```
location [=|~|~*|^~] /uri/ { … }
    * ~ 为区分大小写匹配
    * ~* 为不区分大小写匹配
    * !~和!~*分别为区分大小写不匹配及不区分大小写不匹配
```

示例 1:

```
location = / {
# matches the query / only.
# 只匹配 / 查询。
}
```

匹配任何查询，因为所有请求都已 / 开头。但是正则表达式规则和长的块规则将被优先和查询匹配

示例 2:

```
location ^~ /images/ {
# matches any query beginning with /images/ and halts searching,
# so regular expressions will not be checked.# 匹配任何已 /images/ 开头的任何查询并且停止搜索。任何正则表达式将不会被测试。
```

示例 3:

```
location ~* \.(gif|jpg|jpeg)$ {
# matches any request ending in gif, jpg, or jpeg. However, all

# requests to the /images/ directory will be handled by
}# 匹配任何已 gif、jpg 或 jpeg 结尾的请求。
```

十二、Nginx 日志处理

1、Nginx 日志切割

```
#contab -e
59 23 * * * /var/InfiNET/usemon/awstats/logcronNginx.sh /dev/null 2>&1
```

[root@count ~]# cat /var/InfiNET/usemon/awstats/logcronNginx.sh

```
#!/bin/sh
/bin/mv /home/log/bbs.access.log /home/log/bbs.access.`date +%Y%m%d`.log
/bin/mv /home/log/register.access.log /home/log/register.access.`date +%Y%m%d`.log
/bin/mv /home/log/shutter.access.log /home/log/shutter.access.`date +%Y%m%d`.log
/bin/mv /home/log/ikuaimen.access.log /home/log/ikuaimen.access.`date +%Y%m%d`.log
/usr/bin/killall -s USR1 Nginx
```

2、 Nginx logrotate 处理：

```
/home/log/*access.log {
        daily
```

```
missingok
rotate 31
nocompress
prerotate
#/usr/local/awstats/wwwroot/cgi-bin/awstats.pl -update -config=shutter
endscript
postrotate
if [ -f /usr/local/nginx/logs/nginx.pid ]; then
kill -USR1 `cat /usr/local/nginx/logs/nginx.pid`
fi
endscript
}
```

3、 Nginx and Cronolog

I recently setup Nginx for one of our webservers and needed to hook cronolog up to it for all the normal reasons you want cronolog.

But Nginx doesn't support piped logging yet :(But we can use fifo's though to accomplish the same thing :)

1. Configure nginx.conf to log to logs/access.log and logs/error.log like normal.
2. Remove those files from the logs directory.
3. Recreate them as fifo's using "mkfifo access.log" and "mkfifo error.log".
4. Tweak the nginx startup script to start cronolog just before nginx.

Something like this:

```
(cat /usr/local/nginx/logs/access.log |\
/usr/local/sbin/cronolog -l /var/log/nginx/access.log \
/var/log/nginx/%Y/%m/%d/%H/access.log) &
(cat /usr/local/nginx/logs/error.log |\
/usr/local/sbin/cronolog -l /var/log/nginx/error.log \
/var/log/nginx/%Y/%m/%d/%H/error.log) &
```

That's it. It seems that you'd need to stop cronolog when shutting down nginx, but at least on CentOS this isn't required. I suspect that when the fifo is closed for writing it gets closed for reading and cat exists which exits cronolog as well. Would love it it someone could confirm that though.

UPDATE

Igor Sysoev made the comment that the above might hinder nginx's performance because of context switching and the blocking between the worker processes. So instead of the above you can simulate it with the following as an hourly cron task:

```
log_dir="/usr/local/nginx/log"
date_dir=`date +%Y/%m/%d/%H`
/bin/mkdir -p ${log_dir}/${date_dir} > /dev/null 2>&1
/bin/mv ${log_dir}/access.log ${log_dir}/${date_dir}/access.log
/bin/mv ${log_dir}/error.log ${log_dir}/${date_dir}/error.log
kill -USR1 `cat /var/run/nginx.pid`
```

```
/bin/gzip ${log_dir}/${date_dir}/access.log &
/bin/gzip ${log_dir}/${date_dir}/error.log &
```

4、 Nginx 如何不记录部分日志

日志太多，每天好几个 G，少记录一些，下面的配置写到 server{}段中就可以了代码:

```
location ~ .*\.(js|jpg|JPG|jpeg|JPEG|css|bmp|gif|GIF)$
{
    access_log off;
}
```

十三、Nginx Cache 服务配置

如果需要将文件缓存到本地，则需要增加如下几个子参数：

```
proxy_store on;
proxy_store_access user:rw group:rw all:rw;
proxy_temp_path  缓存目录;其中，
proxy_store on 用来启用缓存到本地的功能，
proxy_temp_path 用来指定缓存在哪个目录下，如：proxy_temp_path html;
```

在经过上一步配置之后，虽然文件被缓存到了本地磁盘上，但每次请求仍会向远端拉取文件，为了避免去远端拉取文件，必须修改 proxy_pass：代码:

```
if ( !-e $request_filename) {
    proxy_pass  http://freeke;
}
```

即改成有条件地去执行 proxy_pass，这个条件就是当请求的文件在本地的 proxy_temp_path 指定的目录下不存在时，再向后端拉取。

十四、Nginx 负载均衡

1. Nginx 基础知识

Nginx 的 upstream 目前支持 4 种方式的分配

1)、轮询（默认）
每个请求按时间顺序逐一分配到不同的后端服务器，如果后端服务器 down 掉，能自动剔除。

2)、weight
指定轮询几率，weight 和访问比率成正比，用于后端服务器性能不均的情况。

3)、ip_hash
每个请求按访问 ip 的 hash 结果分配，这样每个访客固定访问一个后端服务器，可以解决 session 的问题。

可以针对同一个 C 类地址段中的客户端选择同一个后端服务器，除非那个后端服务器宕了才会换一个。

4)、fair（第三方）
按后端服务器的响应时间来分配请求，响应时间短的优先分配。

5)、url_hash（第三方）
按访问 url 的 hash 结果来分配请求，使每个 url 定向到同一个后端服务器，后端服务器为缓存时比较有效。

2. Nginx 负载均衡实例 1

```
upstream bbs.chinarenservice.com {#定义负载均衡设备的 Ip 及设备状态
    server 10.0.0.101 down;
    server 10.0.0.102 weight=2;
    server 10.0.0.103;
    server 10.0.0.104 backup;
    server 10.0.0.105 weight=5;
    server 10.0.0.106 weight=8;
}
```

在需要使用负载均衡的 server 中增加代码:

```
proxy_pass [url]http://bbs.chinarenservice.com/;
```

每个设备的状态设置为:代码:

```
1.down 表示单前的 server 暂时不参与负载
2.weight 默认为 1.weight 越大，负载的权重就越大。
3.max_fails ：允许请求失败的次数默认为 1.当超过最大次数时，返回 proxy_next_upstream 模块定义的错误
4.fail_timeout:max_fails 次失败后，暂停的时间。
5.backup： 其它所有的非 backup 机器 down 或者忙的时候，请求 backup 机器。所以这台机器压力会最轻。
Nginx 支持同时设置多组的负载均衡，用来给不用的 server 来使用。

client_body_in_file_only 设置为 On 可以讲 client post 过来的数据记录到文件中用来做 debug
client_body_temp_path 设置记录文件的目录 可以设置最多 3 层目录
location 对 URL 进行匹配.可以进行重定向或者进行新的代理 负载均衡
```

8、 Nginx 负载均衡实例

按访问 url 的 hash 结果来分配请求，使每个 url 定向到同一个后端服务器，后端服务器为缓存时比较有效,也可以用作提高 Squid 缓存命中率.

简单的负载均等实例:
#vi Nginx.conf //Nginx 主配置文件核心配置代码:

```
#loadblance my.chinarenservice.com
        upstream  my.chinarenservice.com  {
        ip_hash;
        server    172.28.5.71;
```

```
        server    172.28.5.72;
        server    172.28.5.73;
        server    172.28.5.74;
        server    172.28.5.75;
        server    172.28.5.76;
    }
……..
include             vhosts/chinarenservice_lb.conf;
……
#vi proxy.conf
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;
client_max_body_size 50m;
client_body_buffer_size 256k;
proxy_connect_timeout 30;
proxy_send_timeout 30;
proxy_read_timeout 60;

proxy_buffer_size 4k;
proxy_buffers 4 32k;
proxy_busy_buffers_size 64k;
proxy_temp_file_write_size 64k;
proxy_next_upstream error timeout invalid_header http_500 http_503 http_404;
proxy_max_temp_file_size 128m;
proxy_store on;
proxy_store_access    user:rw  group:rw  all:r;

#Nginx cache
client_body_temp_path  /usr/local/Nginx/cache/client_body 1 2;
proxy_temp_path /usr/local/Nginx/cache/proxy_temp 1 2;
#vi chinarenservice_lb.conf
server
    {
        listen  80;
        server_name my.chinarenservice.com;
        index index.php;
        root /var/InfiNET/web/mychinarenservice;
        if (-f $request_filename) {
            break;
          }

        if (-f $request_filename/index.php) {
```

```
        rewrite (.*) $1/index.php break;
    }

    error_page 403 [url]http://my.chinarenservice.com/member.php?m=user&a=login;
    location / {
        if ( !-e $request_filename) {
            proxy_pass [url]http://my.chinarenservice.com;
            break;
        }
        include cconf/proxy.conf;
    }
}
```

十五、Nginx 原理代码分析：

1、剖析 Nginx 等单线程服务器设计原理与性能优势

Nginx 现在正在以光的速度蔓延开来，他以其稳定性和高性能等众多优点迅速扩大市场，大家都知道，Nginx 是以单线程为基础的，那么他怎么能在并发性上取得优势的呢？会不会因为网络阻塞而导致主线程阻塞呢？下面就相关问题作一些概念性的阐述。

问题的根本在于人们对于计算机处理性能还没有足够的认识，以及普通的服务器架构简化的处理，做过大型的成熟服务器的人可能都知道，解决一个系统瓶颈比优化 1000 个算法还重要，这也就是木桶效应，一个桶能盛水的多少决定于最短的那一块板，我们之所以在一般的服务器端应用软件中采用一个连接一个线程甚至阻塞在一个线程上的做法，并不是这个方法是最优秀的，设计者没有更好的方法，而是因为这种套路是最简单的，在概念上以及操作上都比较容易让人理解，并且容错性也强，但是对于性能要求极高的服务器比如 dns 或者负载均衡等等，要求处理速度极快，并且能有较高的并发性，这种简单的线程池加连接池的做法就不能解决问题了，比如一个 index 页面请求，他会包含几十个附属资源文件，如果 cilent 网络比较慢，那么就会较长时间的阻塞这几十个连接，用户稍微一多服务器就受不了，因为线程的开销是很大的，如果不能得到迅速释放，将会给服务器带来灾难性的后果，对于公网服务，这种之后会尤为明显，很显然，让服务器为客户端的网速买单是愚蠢的做法。

那么既然多线程都会存在这样的问题单线程怎么会逃脱的调呢？解决问题的关键在于异步 IO，windows 上有 IOCP（完成端口，对于一部 IO 包装的比较多，内部实现时用 cpu 个数的线程进行事件处理，他会通知你你给定的异步读写已经完成了），linux 上有 epool(一个纯事件通知接口，他会通知你可以读或者可以写了)，如果将所有的请求简化为阻塞操作和非阻塞操作问题就简单了，所有需要阻塞请求的部分全部由 epool 触发相应事件，非阻塞（处理耗时很短）部分用主线程一直执行，直到遇到阻塞部分就停止，交由阻塞部分监听异步完成事件，这样就构成了事件驱动模型。

这里比较容易迷惑人的地方是很多人认为函数的处理会阻塞主线程，其实还是上面说的木桶效应，他是不是那块最短的木板，这是需要由测试和经验来决定的，事实是他的处理时间占用很短，做 100 万次 for 循环说不定比局域网经过一次网络访问的时间还要短，理解了这点就不难理解了，如果说你的服务器每秒钟能处理 1 万个请求，那么在处理功能函数上（比如解析协议，操作、输出等等）顶多也就占用 0.1-0.3 秒，剩下的时间都是耗时在了网络阻塞上，耗时在了事件发生上了，既然如此，把操作部分独立分出来用多线程执行又有什么意义呢？对于公网就更不用说了，网络等 IO 阻塞才是影响服务器的主要因素，是那块短了的板。

对于网络的 IO，IOCP 、epool 等事件通知机制就解决了这个问题，性能上由于是阻塞的，所以还不如直接 accept 等快，但是对于网络延时很严重的情况下性能反而显得更好，因为他们可以处理大量的连接而不使性能下降很厉害，如果值直接阻塞能连接处理 1000 个的话，epool 等就可以同时处理 3-5 万个，所以实际的应用价值要大得多。

剩下的部分就是处理事件发生后的事情上面，我前面的文章已经作了说明，在此不再重复，Nginx 、lighttpd 等都是基于这类模型开发的，有兴趣的可以研究一下他的代码。

2、Nginx 等 web 服务器设计中关于相关注意事项与心得

1、sokcet 和文件 fd 的关闭问题：看起来这是个简单的问题，但是正如内存分配和释放一样，这里也是很容易发生问题的一个地方，在做到反向代理的时候遇到了一个新的问题，一个 fd 会伴随这另外的 socket fd，或者会产生一个文件 fd，这些描述服在一次服务结尾的时候进行释放是理所当然的，但是是程序就会有异常，这些 fd 很容一在异常处理中被忽略，举例来说，我在处理 keep－alive 的长链接的时候就发生了问题，事情是这样的：

一方面不知道用户什么时候关闭连接，第二方面对于一个连接的重用会进行有异于关闭的相关操作，第三方面，如果在接收或者发送的时候数据没有发送接收完全，那么在 socket buff 里面的残余数据就会造成下一次 epool 事件的混淆，但绝大多时候我们无法在 server 层面一次性接收所有的数据，比如一个 post，或者一次文件上传，需要特殊的处理过程产生容器以及对接收到的数据进行存储处理等等，这些操作需要 server 将请求转向给目标处理器，sever 要做的是处理协议头部，所以一般来讲首次只会接收很少两的部分，那么剩余的部分就会对下一次连续请求造成影响。

在这个问题的处理过程上我一开始就对于请求相关的文件 fd 没有处理好，导致在测试的时候看到 fd 直线上升，本来 keep－alive 就是为了减少 fd，这种问题开始检查了半天都没有检查出个所以然，然后 buff 而的数据也是如此，所以我现在的对于 buff 的处理是这样的，如果数据不包含协议，就说明数据有问题，直接断掉连接，新的数据会用新的连接来连接，不然就会为了大量无用的数据进行更多的读操作。

所有我的设计核心也就是尽量减少制程的长度 server 层面能解决的就在 server 层解决。

资源的释放是一个很复杂的综合能力，既需要设计着的细心，也需要一定的技巧，现在感觉一个好的设计模式是很重要的，为了功能而使设计简化是应该做得，但是没有任何设计模式，只是简单的流水执行的方法对于复杂一些的工程是行不通的，随着工程的复杂度增加，前面省下来的时间就全都用在了调试和维护上，工程大到最后肯定是要流产的，对于 c 语言我要说的就是合理利用指针，特别是函数指针对于模式的改进是很有用的。

2、还有一个两好的 log 系统是很有用的，初步用 c 的用户都会直接用 printf 进行调试，倒是简单明了，但是很难对于整个工程进行这样的调试，我以前的做法也是 printf，结果调试完了一块就把他注释起来，防止影响剩下的部分，结果当不知道问题发生在那个模块的时候，printf 调试就显得笨拙了，调试信息多到你自己都没法看懂，但是 log 系统的好处就是方便查阅问题的出处，可以用一个控制函数对所有的调试信息进行屏蔽，这样往往能收到意想不到的效果，另外，建议将 Error 和 Debug 分开两个 log 文件来存储，最好将 Recode/log 也分开存储，因为这三个功能的还是有一定的差别的，一般来讲 Error 是一定要记录的，Debug 用于调试，Log 用于记录正常访问，但是开启 log 系统对于系统的性能影响还是满大的，特别对于单机测试的时候发现 log 影响大到了 50％的将速，对于公网影响小一些，毕竟访问没有那么快，瓶颈还是在网络上。

3、做起来比想象的容易。理解了架构其实写一个 server 的时候难点就在于调试，而不在于实现具体功能，特别是对于 c 的程序，调试是很麻烦的，在 windows 上还好，vs2008 的调试功能还是超级一流的，他能在调试中将链表层层显示出来，感觉非常好，但是对于 linux 下的用户还是没有那么幸运。

4、我现在用的是 netbeans，感觉还凑活，只是比很多人推荐的 c 专业编辑器要好，不要相信那个邪，写 c 代码一定要用文本编辑器或者最基本的代码编辑器，那是骗人的，我们需要的是产品，任何有利于产品迅速成形的方法和工具都是可以利用的，文言文中有篇文章的中心思想是这样的：君子就是善于利用工具的人；把自己折腾的头昏眼花不是一个好习惯，耍酷更是没有必要，我到现在还不会手动写 make 文件，因为我把精力全部放在了架构和模式上了，再有点精力就优化一下功能函数，但我没有精力向大姑娘绣花一样。

5、写好注释。不管你对架构有多熟悉，哪怕你的代码永远也打算对第三方开放，只要你还会维护这段代代码，就尽量做好注释，月详细越好，因为总有一天你会记不住那么多东西的，等你拿到一段没有注释的代码你自己都会对他实现的公能没有信息。

6、做好测试。如果是一个人写代码很容一忽略测试这一块，只要调试通了就可以了，其实对于 c 这样的

语言很容一出问题的，又没有 try catch 可用，所以需要对于每一块的功能都要有强力的测试，使用起来才会有信心，测试要用最为极端的调试。另外还有一个好习惯是做好测试输出，比如 hashtable 和 string 函数，他们的测试是有很大的区别的，以后发生了问题每次都重复来测试既伤神又耗力，做好一个输出，比如在 web 系统中这种输出就可以在 web 页面中看到调试，这种灵活的调试有利于稳定的工程进行，Nginx 就可以对于单独的一个请求进行调试，只有对于内部的运行料如执掌才会有优秀的产品。

3、向上取倍数，Nginx 实现内存对齐的宏

#define ngx_align(d, a) (((d) + (a - 1)) & ~(a - 1))

对于a，传入cpu的二级cache的Line大小，通过ngx_cpuinfo函数，可以获取ngx_cacheline_size的大小，一般intel为 64 或者 128。

计算宏ngx_align(1,64)=64，只要输入的d<64，则结果总为 64，如果输入的d=65,则结果为 128，以此类推。

进行内存池管理的时候，对于小于 64 字节的内存，给分配 64 字节，使之总是cpu二级缓存读写行的大小倍数，从而有利于cpu二级缓存的存取速度和效率。

ngx_cpuinfo函数，实际调用了汇编代码，获取cpu二级cache的行大小！

关于CPU缓存，参考链接：

http://hi.baidu.com/%D0%A1%DE%B1%B1%F0%D7%DF/blog/item/4d07e10719ec57cb7b89472d.html

i386 体系下的检测代码如下：

```
/* auto detect the L2 cache line size of modern and widespread CPUs */

void
ngx_cpuinfo(void)
{
    u_char    *vendor;
    uint32_t   vbuf[5], cpu[4];

    vbuf[0] = 0;
    vbuf[1] = 0;
    vbuf[2] = 0;
    vbuf[3] = 0;
    vbuf[4] = 0;

    ngx_cpuid(0, vbuf);

    vendor = (u_char *) &vbuf[1];

    if (vbuf[0] == 0) {
        return;
    }

    ngx_cpuid(1, cpu);

    if (ngx_strcmp(vendor, "GenuineIntel") == 0) {

        switch ((cpu[0] & 0xf00) >> 8) {
```

```
        /* Pentium */
        case 5:
            ngx_cacheline_size = 32;
            break;

        /* Pentium Pro, II, III */
        case 6:
            ngx_cacheline_size = 32;

            if ((cpu[0] & 0xf0) >= 0xd0) {
                /* Intel Core */
                ngx_cacheline_size = 64;
            }

            break;

        /*
         * Pentium 4, although its cache line size is 64 bytes,
         * it prefetches up to two cache lines during memory read
         */
        case 15:
            ngx_cacheline_size = 128;
            break;
        }

    } else if (ngx_strcmp(vendor, "AuthenticAMD") == 0) {
        ngx_cacheline_size = 64;
    }
}

static ngx_inline void
ngx_cpuid(uint32_t i, uint32_t *buf)
{

    /*
     * we could not use %ebx as output parameter if gcc builds PIC,
     * and we could not save %ebx on stack, because %esp is used,
     * when the -fomit-frame-pointer optimization is specified.
     */

    __asm__ (

    "    mov    %%ebx, %%esi; "
```

```
    "    cpuid;                "
    "    mov    %%eax, (%1);    "
    "    mov    %%ebx, 4(%1); "
    "    mov    %%edx, 8(%1); "
    "    mov    %%ecx, 12(%1); "

    "    mov    %%esi, %%ebx; "

    : : "a" (i), "D" (buf) : "ecx", "edx", "esi", "memory" );
}
```

4、Nginx 的内存池管理分析(a)

Nginx 的内存管理，主要是用来实现防止内存泄露，和内存碎片，而并没有真正的预先分配获得大量内存。因此实际上和普通意义上的内存池有一定的区别。预先分配的内存通常也就是 1 块 16K 大小的内存快，大内存直接使用 malloc 分配，16k*n 小内存块中，用来处理各种小的内存资源需求，并且不是放，对一个连接来说，连接完成后统一释放，既防止了小内存碎片，也防止了泄露。由于系统操作中，分配小块内存的频率远远高于分配大块内存的频率，因此实际大大较小了 malloc 的系统调用，同时实现也比较简练，也不用向通常的内存池管理一样，多任务环境下，还需要考虑加锁的问题。而锁将大大降低程序的性能。

1，内存池就是一个单链表
2，拥有独立的多个内存池，而不是全局唯一的内存池，比如针对每个新建立的连接，创建一个新的内存池，并在连接结束后，一次性释放资源
3，内存池本身分配小内存，防止碎片，大内存直接使用 malloc，多个大内存构成单链表，并挂到 pool 的 large 成员
4，每个连接处理，针对一些需要在连接关闭的时候需要做清理的工作，可以在 pool 中注册回调处理函数，比如 unlink 文件，关闭文件等操作，防止系统资源，如文件句柄的泄漏！
5，小内存一旦分配使用，直到整个池释放，才被回收，这样管理简单，最重要的是效率高，没有碎片，不用来回的在小内存中管理，而且小内存块可以被完整使用！(16k，应该可以应付很多次调用了！通常一个连接处理，能用到多少个 16K 的内存块呢？比如复杂的邮件应用？)而
6，大内存，可以通过调用释放函数，在摧毁池之前，提前回收！
存疑问：线程模式下安全性如何？每连接一个池，保证了各个内存操作的独立性？

Nginx 管理内存的宗旨，就是:
1,高效，不在小内存上花费太多时间
2,没有碎片
3,简单，只用到单链表，不引入复杂的操作
4,隔离设计，每个连接单独一个池，方便回收，防止泄露，针对单连接，调用 destroy 可以完全放心没有内存的泄漏，很适合网络应用程序的开发

以下是一些整理：

内存分配函数
ngx_palloc(ngx_pool_t *pool, size_t size)
ngx_memalign，该函数没有使用，而用宏实现了内存对齐
#define ngx_align_ptr(p, a) \
(u_char *) (((uintptr_t) (p) + ((uintptr_t) a - 1)) & ~((uintptr_t) a - 1))
内存结构体：

```
struct ngx_pool_s {
    u_char              *last;记录上次的位置(在内存块中，上次的位置)
    u_char              *end;(记录内存块最后的位置)
    ngx_pool_t          *current;
    ngx_chain_t         *chain;//buf 结构体的单链表组成，该 buf 可以使 mmap 内存(ngx_write_chain_to_temp_file,
可以实现多块内存同时写文件操作)
    ngx_pool_t          *next;
    ngx_pool_large_t    *large;//大内存的分配，一个单链表
    ngx_pool_cleanup_t  *cleanup;//针对有需要清理的数据，如删除文件等等，将之
    ngx_log_t           *log;
};
struct ngx_chain_s {
    ngx_buf_t    *buf;//buf 结构体
    ngx_chain_t *next;
};
```

5、Nginx 的内存池管理分析(b)

buf 结构可能有多种不同的类型，如内存中的，或者文件中的(0 copy 有用)，
内存中的，内存大小为 last-pos
文件中的，数据块大小为 file_last-file_pos

```
struct ngx_buf_s {
    u_char          *pos;
    u_char          *last;//结束位置
    off_t           file_pos;
    off_t           file_last;

    u_char          *start;         /* start of buffer *///内存起始位置
    u_char          *end;           /* end of buffer *///内存结束位置
    ngx_buf_tag_t    tag;
    ngx_file_t      *file;
    ngx_buf_t       *shadow;


    /* the buf's content could be changed */
    unsigned         temporary:1;

    /*
```

```
     * the buf's content is in a memory cache or in a read only memory
     * and must not be changed
     */
    unsigned         memory:1;

    /* the buf's content is mmap()ed and must not be changed */
    unsigned         mmap:1;

    unsigned         recycled:1;
    unsigned         in_file:1;
    unsigned         flush:1;
    unsigned         sync:1;
    unsigned         last_buf:1;
    unsigned         last_in_chain:1;//是否该链中的最后一个块

    unsigned         last_shadow:1;
    unsigned         temp_file:1;

    unsigned         zerocopy_busy:1;

    /* STUB */ int    num;
};
```

****

函数功能：创建内存池,分配一个特定大小空间，并初始化池

```
ngx_pool_t *
ngx_create_pool(size_t size, ngx_log_t *log)
```

实际应用：

****

main 函数:init_cycle.pool = ngx_create_pool(1024, log)，单位大小为 1K 字节块的池

ngx_init_cycle 函数中 ngx_create_pool(NGX_CYCLE_POOL_SIZE, log);NGX_CYCLE_POOL_SIZE 大小 16K 字节！

ngx_event_accept: 接收到新连接的时候，创建 connection 上下文，并建立一个池 c->pool = ngx_create_pool(ls->pool_size, ev-

>log);这个是针对连接上下文创建的池

针对某个连接，小内存一直分配，直到连接全部完毕，才回收，大内存可以通过 ngx_pfree 来释放，可以做到方便的内存回收和分配，

以连接为单位！或者其他的为单位

```
ngx_pool_t *
ngx_create_pool(size_t size, ngx_log_t *log)
{
    ngx_pool_t *p;
```

```
    p = ngx_alloc(size, log);//实际调用 malloc 分配内存，log 用来记录日志
    if (p == NULL) {
        return NULL;
    }

    p->last = (u_char *) p + sizeof(ngx_pool_t);
    p->end = (u_char *) p + size;
    p->current = p;
    p->chain = NULL;
    p->next = NULL;
    p->large = NULL;
    p->cleanup = NULL;
    p->log = log;

    return p;
}


struct ngx_pool_large_s {
    ngx_pool_large_t     *next;
    void                 *alloc;
};
```

数组管理

****

ngx_array_push(ngx_array_t *a)

函数功能：在数组中，找到一个可用元素空间，返回该元素空间指针！如果空间不够，进行数组扩展内存，必要时需要复制原有数据

。

****

ngx_array_push(ngx_array_t *a)

```
struct ngx_array_s {
    void        *elts;//整个数组的起始地址！
    ngx_uint_t   nelts;//当前拥有了数据的元素个苏
    size_t       size;//单个数组元素的单位大小
    ngx_uint_t   nalloc;//已经分配了内存的全部的元素个数(最大可用元素)
    ngx_pool_t *pool;//在某个内存池上作的操作
};
```

数组可能是静态分配的，比如 ngx_write_chain_to_file 函数中的数组

```
struct iovec *iov, iovs[NGX_IOVS];
    vec.elts = iovs;
```

vec.size = sizeof(struct iovec);
vec.nalloc = NGX_IOVS;
vec.pool = pool;

ngx_array_push 函数实现的功能：
找到当前数组中第一个可用的空间，并返回其指针，实际使用的元素 nelts 加 1，如果 nelts 的数目达到 nalloc，则在内存池 pool 中，分

配并增加一个元素大小的内存。
如果最后一个元素正好是最近 pool 中分配使用的内存，并且 pool 中还有足够的空间，则将 nalloc++,既可以！elts 的内存继续保持联系

的内存块！否则调用 ngx_palloc 在 pool 上分配 2 倍的该元素的空间！
ngx_memcpy(new, a->elts, size);并将原有的 elts cp 到新分配的空间中。
将数组的 elts 指向新的空间地址，nalloc=nalloc*2;

****
函数功能
void * ngx_palloc(ngx_pool_t *pool, size_t size)
实现在 pool 中，获取 size 大小的可用空间，如果空间不能分配到，则创建扩展池块或者直接申请大内存块。
****
如果满足如下条件：
1，需分配空间大小不大于 NGX_MAX_ALLOC_FROM_POOL(也就是 ngx_pagesize -1)
2，需分配空间大小不大于池当前总的大小(实际物理内存大小减掉 ngx_pool_t 结构体内存对齐后的空间，为真正池可用分配的空间)
则：
首先，定位当前池中开始分配的位置，
取当前池，如果分配大小小于 sizeof(int) ，或者单数字节，开始位置就是 last
否则，先对于最后的 last 指针，做一个对齐 ngx_align_ptr(p->last, NGX_ALIGNMENT);
#define NGX_ALIGNMENT sizeof(unsigned long) ，也就是对无符号 long 的字节数对齐，如 64 位 8 字节对齐，32 位 4 字节对齐。
再判断剩下的空间，如果足够，则返回该地址，并记录 last。
如果池剩下的空间，不够 NGX_ALIGNMENT，不够对齐，则 current 指向 next，pool 的链表的下一个！从下一个链表中查找可用空间！并

记录 pool 的当前池的位置
一直循环，直到有一个链表可以分配到空间，就退出！

如果最后还是没有空间可以分配，则 ngx_create_pool，创建一个新块，并在新块中分配内存。
新建块的大小等于单个池块的大小！
并把新建的 pool,挂接到原有池的链表尾部！
不过不满足上面的两个条件，也就是需要分配的是大内存块，则
直接调用 ngx_alloc，分配内存
在 ngx_palloc，将 p 挂接到该 pool 的的 large 链表中去！

****
函数功能:该函数用来创建池，也用来创建池中单个块！记住，池的话，一直需要记录 pool 的指针，因为虽然和其他块的数据结构一样

，但池作为表头，还担有存放大内存链和 cleanup 的任务！
void
ngx_destroy_pool(ngx_pool_t *pool)
实现在 pool 中，增加空间
****
首先对 cleanup 进行调用，将清理函数运行完毕
其次，释放 large 链，将池的 large 释放完毕
最后释放 pool 链
清理池，只针对 pool 链表的头节点，进行 cleanup 和 large 释放。

****
函数功能：在池中，释放某块大内存，只释放大内存块，池内分配的小内存块，并没有释放
ngx_pfree(ngx_pool_t *pool, void *p)
****
查询 pool 的 large 链表，如发现 p 的指针就是某个节点记录的 alloc 指针，就释放该 p,但在 large 链表中，并不删除 large 节点
****
函数功能：
ngx_pool_cleanup_t *
ngx_pool_cleanup_add(ngx_pool_t *p, size_t size)
****
在池中增加 cleanup 节点，push 到节点表头，并分配 size 大小的空间，返回该分配的节点，节点包括 handler 和 data 两个数据成员
调用程序在成功后，需要注册 handler 回调函数

6、Nginx 数据结构 数组，列表

@ core/ngx_array.{h,c}

```
struct ngx_array_s {
    void *elts; /*  数组内容空间的指针  [= size * nalloc] */
    ngx_uint_t nelts; /*数组已使用的元素个数*/
    size_t size; /*  数组元素大小  */
    ngx_uint_t nalloc; /*  数组可使用的元素个数  */
    ngx_pool_t *pool /*  数组内容内存分配器指针  */
}
```

ngx_array_create => 输入分配器，元素个数，元素大小，返回数组指针
    数组和数组内容的内存由本配器分配

ngx_array_destroy => 输入数组指针

算法很简单，如果要被释放的内存刚好就是分配器刚分配的，那就调整分配器的 last，指向分配前的位置。问题是，这样就需要分配和释放遵循严格的对称性，否则就会难以释放内存

ngx_array_push => 输入数组指针，输出数组内容空间的指针
nelts++
如果 nelts < nalloc，没什么，直接返回
如果 nalloc * size + elts 刚好等于 p->last，并且 p->last + size <= p->end，直接调整 p->last 和 nalloc
如果不满足上一条，就 palloc 个新的数组，大小为 2 * nalloc，然后 ngx_memcpy，原内存不释放

ngx_array_push_n => 输入数组指针和要增加几个，输出数组内容空间的指针
实现跟上一个一样，就是把 1 个扩展为 n 个

@ core/ngx_list.{h,c}

Nginx 列表通过 chunk 技术实现，接口只有两个 ngx_list_create 和 ngx_list_push，前者用于创建一个包含 n 个 size 大小元素的列表，后者则是后插入元素，并在必要的时候扩容
list 本身记录 每个 chunk 包含 nalloc 个元素，每个元素大小为 size，part 是第一个 chunk，last 是最后一个 chunk 的指针
chunk 就是个简化的数组，elts 是空间，nelts 是用了多少个，next 指向下一个 chunk，也就是 ngx_list_part_t
ngx_list_push 先检查 last_part 的 nelts 是否等于 nalloc，也就是判断是否还有地方，没有的话就分配一个 ngx_list_part_t，正好也就预分配了 nalloc 个元素空间
由此可以看出，ngx_list 在插入和随机访问上都很有优势，算得上是个相当可爱的实现

7、Nginx 源代码分析

Nginx 由以下几个元素组成：

1. worker（进程）
2. thread（线程）
3. connection（连接）
4. event（事件）
5. module（模块）
6. pool（内存池）
7. cycle（全局设置）
8. log（日志）

大概就这些元素组成的。

整个程序从 main()开始算

```
ngx_max_module = 0;
for (i = 0; ngx_modules[i]; i++) {
    ngx_modules[i]->index = ngx_max_module++;
}
```

这几句比较关键，对加载的模块点一下数，看有多少个。ngx_modules 并不是在原代码中被赋值的，你先执行一下./configure 命令生成用于编译的 make 环境。在根目录会多出来一个文件夹 objs，找到 ngx_modules.c 文件，默认情况下 Nginx 会加载大约 30 个模块，的确不少，如果你不需要那个模块尽量还是去掉好一些。

接下来比较重要的函数是 ngx_init_cycle()，这个函数初始化系统的配置以及网络连接等，如果是多进程方式加载的会继续调用 ngx_master_process_cycle()，这是 main 函数中调用的最关键的两个函数。

ngx_init_cycle()实际上是个复杂的初始化函数，首先是加载各子模块的配置信息、并初始化各组成模块。

任何模块都有两个重要接口组成，一个是 create_conf，一个是 init_conf。分别是创建配置和初始化配置信息。

模块按照先后顺序依次初始化，大概是这样的：

```
&ngx_core_module,
&ngx_errlog_module,
&ngx_conf_module,
&ngx_events_module,
&ngx_event_core_module,
&ngx_epoll_module,
&ngx_http_module,
&ngx_http_core_module,
&ngx_http_log_module,
```

首先是内核模块、错误日志、配置模块、事件模块、时间内核模块、EPOLL 模块、http 模块、http 内核模块、http 日志模块，剩下的模块都算不上关键。

epoll 是比较关键的核心模块之一，Nginx 兼容多种 IO 控制模型，memecached 用的是 libevent 不如 Nginx 彻头彻尾是自己实现的。

在 ngx_init_cycle()中对模块初始化完毕后，调用 ngx_open_listening_sockets()函数对 socket 进行了初始化。

在 listen 上 80 端口以后，调用模块的另外一个重要接口 init_module 对各模块进行初始化。

并不是每个模块都对 init_module 接口进行了定义，在比较重要的模块中仅有 ngx_http_log_module 对这个接口进行了定义。

ngx_init_cycle()返回后，主要的工作都是在 ngx_master_process_cycle()函数中继续进行的。

ngx_master_process_cycle()函数中的重要过程有调用 ngx_start_worker_processes()生成多个子进程，一般 Nginx 是多进程的。

ngx_start_worker_processes()函数内部调用 ngx_worker_process_cycle()函数建立每个进程的实际工 作内容，在这个函数中首先调用 ngx_create_thread()初始化各线程。我们知道每个线程都有一个启动处理函数，Nginx 的

线程处理函数为 ngx_worker_thread_cycle()，内部过程中最重要的是对 ngx_event_thread_process_posted()函数的调用，用于实际处理每一次请求。

在初始化线程结束后，首先调用 ngx_process_events_and_timers()函数，该函数继续调用 ngx_process_events 接口监听事件，一般情况下对应的函数是 ngx_epoll_process_events()，如果使用的是其它种 类的 IO 模型，则应该实现相应的实际函数。这个接口负责把事件投递到 ngx_posted_events 事件队列里，并在 ngx_event_thread_process_posted()函数中进行处理。

8、Nginx代码分析之（一）——初探

发现Nginx是无意间在浏览器中看到新浪的一个错误页面“Nginx ...”，不由起了好奇心。google了一把，发现这是一个支持负载均衡的反向代理服务器，俄罗斯人开发的，虽然没有走GNU或BSD的License，但是也算是一个开源软件。

用工具确认了一下，新浪blog应该是用的Nginx没错，下面是执行curl -I http://blog.sina.com.cn/ 的结果

```
HTTP/1.1 200 OK
Via: 1.1 ISASERVER
Connection: Keep-Alive
Proxy-Connection: Keep-Alive
Content-Length: 365631
Expires: Wed, 07 Nov 2007 09:31:17 GMT
Date: Wed, 07 Nov 2007 09:26:17 GMT
Content-Type: text/html
Server: Nginx/0.5.32
~~~~~~~~~~~~~~~~这一行
Last-Modified: Wed, 07 Nov 2007 09:18:01 GMT
Cache-Control: max-age=300
X-Via: Tj-206
Accept-Ranges: bytes
X-Via: Proxy by Tj-197
```

开源代理服务器最熟悉的还是Squid和Apache，但这两者都是正反向代理通吃的，而作为反向代理，实际上和正向代理有较大的差别。我想既然新浪也用它，那自然有它的独到之处。查了一下，中文的网页上说它的HTTP性能可以达到 13000TPS以上，但是没有说明数据的出处，国外的网站上暂时找不到相应的数据，但很多人拿它和lighttpd相比。

很快下载了Nginx 0.5.32版本的代码，代码不多，才 8 万多行，在openssl的基础上支持HTTPS。和Apache的 30 多万行相比，精简了很多，

作为web server或反向代理，要的就是一个快，要做到快，除了精简的代码之外，更关键的一点就是并发模型。

Apache的弱点就在于它的并发模型是普通的进程/线程池，连接数和进程/线程数是 1：1 的，因此无论是prefork还是worker模式，都将每一个连接对应到一个独立的进程/线程。

这样的并发模型在连接数不太多（1000 以内）时还算可以，但在大规模并发时，其进程/线程总数会非常多。由于Apache本身也比较吃内存，所以到了 1000 以上的并发时，服务器的内存基本上也就被吃的差不多了，操作系统也在频繁地做进程/线程的切换，非常吃力。

相比之下，更高级的大型网络服务系统（如电信的智能网系统）一般采用进程/线程池+状态机的模型——也即接数和进程/线程数是m：n的，这样进程/线程总数就不会由于连接的增多而增多，避免了内存和调度切换的开销，但这种做法对程序逻辑的要求较高，需要一个连接拆分为多个逻辑状态（创建，读，写，关闭等，根据实际业务还可以更加细化）每个进程/线程处理完某一种状态后，需要改变该连接的状态值，后续状态由下一个空闲的进程/线程处理。

Nginx就采用了这样的并发模型，对于连接状态的存储，Nginx主要采用了这样一个复杂结构。

```
struct ngx_connection_s {
    void            *data;
    ngx_event_t     *read;
    ngx_event_t     *write;
    ...
};
```

结构ngx_event_t存储了连接IO状态的详细信息，同时所有的ngx_event_t组成了两个全局的链表，以便进行存取操作。

在这两个数据结构的基础上，Nginx使用了下面这两个函数来完成每个进程/线程的循环

1. ngx_locked_post_event

这个函数负责更新某一个连接的状态，在检查到连接IO状态改变（比如通过select）后被调用。

Nginx以module的方式提供了select语义的多种实现：

poll

devpoll

epoll

eventport

kqueue

rtsig

后面 4 种，都是BSD/Linux为加速IO操作而提供的异步IO模型

2. ngx_event_thread_process_posted

这个函数检查event表，并调用event对应的handler函数，每次处理 1 个event。

这两个函数组合使用，就实现了最基本的m:n并发模型。

9、Nginx代码分析之（二）——Empty Gif是如何工作的

访问新浪时，时常会有一些网页返回空白（但不是“此页无法显示”），从浏览器的信息中可以知道此时服务器返回了一个 1×1 的空白gif图片。

这实际上是Nginx实现的，Nginx有一个名为Empty Gif的module，专门负责此项工作。

由于这个module比较简单，我们就先从它入手，来看看Nginx的模块实现。

模块注册

Empty Gif这个module只有一个文件——ngx_http_empty_gif_module.c

这个文件比较简单，一开始定义并初始化了 3 个变量。

```
static ngx_command_t   ngx_http_empty_gif_commands[] = {...};
static ngx_http_module_t   ngx_http_empty_gif_module_ctx = {...};
ngx_module_t   ngx_http_empty_gif_module = {...};
```

其中只有ngx_http_empty_gif_module是非静态的，我将暂时将其称为module主结构变量，
而其余两个变量都可以由它访问到。

但是如果继续查看Nginx的源码，会发现没有其他地方引用ngx_http_empty_gif_module，
那这个module是怎么注册并应用起来的呢？

如果熟悉Apache的代码，会发现这和Apache 2.0 的module机制非常类似——每个module都对应到一个module主结构变量，通过这个主结构变量可以访问到这个module的其他内容，该module所有的函数也用函数指针的方式存放在这些结构变量中。

而且Apache同样没有其他地方的代码引用到module主结构变量。这是因为module不是必须的，该module在某一个特定的编译版本里是可以不存在的。因此一个module是否有效，不是通过代码来决定，而是通过编译选项来实现。

在Nginx代码的auto目录中，有一个名为sources的文件，根据编译选项（configure的参数）的不同，m4 宏变量HTTP_MODULES的值会发生变化：

如果指定了使用empty gif模块（默认就是使用了），则最终m4 宏变量HTTP_MODULES的值可能如下：

```
HTTP_MODULES="ngx_http_module \
        ngx_http_core_module \
        ngx_http_log_module \
        ngx_http_upstream_module \
        ngx_http_empty_gif_module "
```

注意：这里的ngx_http_empty_gif_module字符串对应了ngx_http_empty_gif_module.c文件中的Module主结构变量名。

编译之前的configure结束后，会在objs目录下生成一个名为ngx_modules.c的文件，此文件的内容如下：

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

extern ngx_module_t   ngx_core_module;
extern ngx_module_t   ngx_errlog_module;
extern ngx_module_t   ngx_conf_module;
...
extern ngx_module_t   ngx_http_empty_gif_module;
...

ngx_module_t *ngx_modules[] = {
    &ngx_core_module,
    &ngx_errlog_module,
    &ngx_conf_module,
    ...
    &ngx_http_empty_gif_module,
    ...
    NULL
};
```

在此生成了对ngx_http_empty_gif_module变量的引用，并将其放到了ngx_modules表中，

通过相关函数可以进行存取。

这样，在编译时就完成了Empty Gif模块注册的过程。

模块的初始化和应用

初始化一般都是根据配置文件的内容来进行，但和我们一般写程序的做法不同——Nginx并没有在一个统一的地方处理所有的配置，而是让每个模块负责处理自己的配置项，如果没有编译这个模块，则其对应的配置项就无法处理，这也是又一个和Apache的相似之处。

Nginx使用了ngx_command_t结构来描述某一个模块对应的配置项及处理函数。

以Empty Gif模块为例：

```
static ngx_command_t   ngx_http_empty_gif_commands[] = {

    { ngx_string("empty_gif"),
      NGX_HTTP_LOC_CONF|NGX_CONF_NOARGS,
      ngx_http_empty_gif,
      0,
      0,
      NULL },

      ngx_null_command
};
```

上面的定义表明：

1. Empty Gif模块只处理一个配置项——“empty_gif”
2. 这个配置是一个Location相关的配置（NGX_HTTP_LOC_CONF），
   即只有在处理某一个URL子集，如 /test_[0-9]*.gif时才生效。

   实际的配置文件可能如下：

```
location ~ /test_[0-9].gif {
        empty_gif;
}
```

3. 这个配置项不带参数（NGX_CONF_NOARGS）
4. 配置处理函数是ngx_http_empty_gif

ngx_http_empty_gif函数的实现很简单：

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

    clcf = ngx_http_conf_get_module_loc_conf(cf, ngx_http_core_module);
    clcf->handler = ngx_http_empty_gif_handler;

    return NGX_CONF_OK;
}
```

ngx_http_conf_get_module_loc_conf是一个宏，用于获得Location相关的配置表cf中ngx_http_core_module对应的项，获取之后，Empty Gif模块将自己的处理函数挂到了ngx_http_core_module对应的handler上。

这样，Nginx在处理HTTP请求时，如果发现其URL匹配到Empty Gif所属的Location，
如URL(/test_1.gif)匹配到Location(/test_[0-9].gif)，
则使用ngx_http_empty_gif作为处理函数，这个函数直接向浏览器写回一幅 1×1 的空白gif图片。

10、Nginx 连接处理

hash 表

Ngnix 使用 hash 表来协助完成请求的快速处理。

考虑到保存键及其值的 hash 表存储单元的大小不至于超出设定参数(hash bucket size)， 在启动和每次重新配置时，Nginx 为 hash 表选择尽可能小的尺寸。

直到 hash 表超过参数(hash max size)的大小才重新进行选择. 对于大多数 hash 表都有指令来修改这些参数。例如，保存服务器名字的 hash 表是由指令 server_names_hash_max_size 和 server_names_hash_bucket_size 所控制的。参数 hash bucket size 总是等于 hash 表的大小，并且是一路处理器缓存大小的倍数。在减少了在内存中的存取次数后，使在处理器中加速查找 hash 表键值成为可能。如果 hash bucket size 等于一路处理器缓存的大小，那么在查找键的时候，最坏的情况下在内存中查找的次数为 2。第一次是确定存储单元的地址，第二次是在存储单元中查找键值。因此，如果 Nginx 给出需要增大 hash max size 或 hash bucket size 的提示，那么首要的是增大前一个参数的大小.

事件模型

Nginx 支持如下处理连接的方法（I/O 复用方法），这些方法可以通过 use 指令指定。

select - 标准方法。 如果当前平台没有更有效的方法，它是编译时默认的方法。你可以使用配置参数 --with-select_module 和 --without-select_module 来启用或禁用这个模块。

poll - 标准方法。 如果当前平台没有更有效的方法，它是编译时默认的方法。你可以使用配置参数 --with-poll_module 和 --without-poll_module 来启用或禁用这个模块。

kqueue - 高效的方法，使用于 FreeBSD 4.1+, OpenBSD 2.9+, NetBSD 2.0 和 MacOS X. 使用双处理器的 MacOS X 系统使用 kqueue 可能会造成内核崩溃。

epoll - 高效的方法，使用于 Linux 内核 2.6 版本及以后的系统。在某些发行版本中，如 SuSE 8.2, 有让 2.4 版本的内核支持 epoll 的补丁。

rtsig - 可执行的实时信号，使用于 Linux 内核版本 2.2.19 以后的系统。默认情况下整个系统中不能出现大于 1024 个 POSIX 实时(排队)信号。这种情况对于高负载的服务器来说是低效的；所以有必要通过调节内核参数 /proc/sys/kernel/rtsig-max 来增加队列的大小。可是从 Linux 内核版本 2.6.6-mm2 开始， 这个参数就不再使用了，并且对于每个进程有一个独立的信号队列，这个队列的大小可以用 RLIMIT_SIGPENDING 参数调节。

当这个队列过于拥塞，Nginx 就放弃它并且开始使用 poll 方法来处理连接直到恢复正常。
/dev/poll - 高效的方法，使用于 Solaris 7 11/99+, HP/UX 11.22+ (eventport), IRIX 6.5.15+ 和 Tru64 UNIX 5.1A+.
eventport - 高效的方法，使用于 Solaris 10. 为了防止出现内核崩溃的问题， 有必要安装 这个 安全补丁。

11、

十六、问题总结 FAQ

1、 反向代理至后端 apache 网站无法输验证码

在使用 Nginx 做反向代理时，后端为 apache 的 BBS 项目，在注册时无法输入注册码，解决办法：

```
proxy_redirect              http://blog.chinarenservice.com/ /;
```

2、 利用 Nginx url hash 提高 squid 服务器命中率

url hash 是用于提高 squid 命中率的一种架构算法，一般现行的架构通常是使用 dns 轮询或 lvs 等将访问量负载均衡到数台 squid，这样做可以使 squid 的访问量做到了均衡，但是忽略了一个重要方面--数据量。在这种架构下，每台 squid 的数据量虽然是一致的，但通常都是满载，并且存在数据重复缓存的情况。如果后端服务器数据容量或者用户的访问热点数远远超过缓存机器的内存容量，甚至配置的 disk cache 容量，那么 squid 将会大量使用磁盘或者不停与后端服务器索取内容。
在新的架构下，使用 Nginx 架载于 squid 之前，如果 squid 机器有 4 台，那么在这 4 台机器上装上 Nginx，Nginx 使用 80 端口，而 squid 改为 3128 端口或其他端口。Nginx 的效率非常高，消耗内存也非常少，所以并不需考虑加装 Nginx 所带来的性能损耗。然后在 Nginx 上配置 url hash，使访问量根据 url 均衡分布到各台 squid，根据 url 分流之后，每一个 url 就会只存在于一台 squid 中，每台 squid 的数据都会完全不同。我们有 4 台机器，每台 2G 内存的话，原先极有可能因为数据大量重复，内存使用率仍然为 2G，而现在我们经过数据均衡分布，8G 内存可以达到充分利用。
是否会存在访问不均的情况呢？是有可能的，但是根据大数原理，访问量基本可以保持一致，只要不存在单一的特别夸张的热点。
假如 squid 是利用 squidclient 来刷新数据的话，新的架构提供了更高效的方法：在后端服务器中模拟 url hash 的算法来找到内容所在的 squid，然后对此服务器刷新内容即可。在旧的架构中，需要遍历所有的服务器，比较低效。
具体配置如下：
Nginx 本身并没有提供 url hash 功能（暂时），需要安装第三方模块 ngx_http_upstream_hash_module
http://wiki.codemongers.com/NginxHttpUpstreamRequestHashModule?action=AttachFile&do=get&target=Nginx_u

```
[root@Chinarenservice oracle]# pstream_hash-0.2.tar.gz
[root@Chinarenservice oracle]# cd Nginx-0.7.32
[root@Chinarenservice oracle]# patch -p0 < /path/to/upstream/hash/directory/Nginx-0.7.32.patch
[root@Chinarenservice oracle]#./configure  --add-module=path/to/upstream/hash/directory
[root@Chinarenservice oracle]# make; make install
```

完成安装
配置：
在 upstream 中加入 hash 语句，server 语句中不能写入 weight 等其他的参数，hash_method 是使用的 hash 算法

```
upstream backend {
    server squid1:3128;
```

```
    server squid2:3128;
    hash    $request_uri;
    hash_method crc32;
}
```

hash 算法可以使用 crc32 和默认的 simple，在 java 中可利用 java.util.zip.CRC32 类实现，simple 算法的 c 语言实现如下

```
#define ngx_hash(key, c)    ((u_int) key * 31 c)
u_int ngx_hash_key(u_char *data, size_t len)
{
    u_int   i, key;
    key = 0;
    for (i = 0; i < len; i ) {
        key *= 31;
        key = data[i];
    }
    return key;
}
```

java 代码（随手写未测试）：

```
public static long getSimpleHash(String data)
{
 long key = 0;
 char[] chars = data.toCharArray();
 for (int i=0; i<chars.length; i ){
  key *= 31;
  key = (int) chars[i];
 }
 return key;
}
```

然后对生成的 key 和 upstream 里的服务器数量做一次求余计算，得到服务器号。

提供 hash 算法的目的如前所述，是便于后端服务器迅速找到内容对应的 squid 服务器。

在 ngx_http_upstream_hash_module 模块里有一个 hash_again 的标签，可以解决 squid 意外死机的问题。不过，如果使用了该标签，那么后端的计算对应服务器的方法就会出现错误。可以使用的办法为，提供一台备份的 squid 服务器，假如有 squid 死机，那么在 Nginx 里设置 error_page 404 和 502 到这台备份服务器，后端刷新缓存时亦要同时刷备份服务器。

3、 Nginx 实践 使用 memcached 模块加速 PHP 应用程序

Nginx 有一个 memcached_module,可以直接从后端的 memached 服务器中读取内容,

直接输出.

通过这个模块,可以极大的提升动态页面的访问速度.

我的实践中, 曾经用这个模块快速的解决了由于代码造成的一些瓶颈问题.

memcached 可以通过 upstream 来从多台 memcached 服务器中读取,也可以支持热备份.

```
upstream memcached {
server 172.28.3.78:11211;
server 172.28.3.78:11212;
server 172.28.3.78:11213;
server 172.28.3.78:11214;
server 172.28.3.78:11215;
server 172.28.3.79:11211;
server 172.28.3.79:11212;
server 172.28.3.79:11213;
server 172.28.3.79:11214;
                    }
server {
listen 80;
server_name blog.chinarenservice.com;
limit_conn one 10;
charset utf-8;
access_log /usr/local/Nginx/logs/blog.log main;
root /var/InfiNET/web;
default_type text/html;
index app;
location = / {
set $memcached_key $host$uri;
default_type text/html;
memcached_pass memcached;
error_page 404 = /missing;
}
location /missing {
internal;
rewrite ^/missing$ /app/site/blog last;
}
include php_app.conf;
}
```

在这个例子中, Nginx 首先从 memcached 中读取(缓存的 key 可以用 memcached_key 来设置.),
如果命中,则直接输出内容,注意,需要设置 default_type ,否则可能不能正常显示.
如果没有命中,memcached_module 会返回 404,因此可以用 error_page 404 来转发到后端的应用程序,
本例中是先转发到/missing ,而/missing 则重写到实际的后端应用地址中.
Nginx 只负责从 memcached 中读取内容,但不负责设置.因此,需要你的应用程序自行将相应的页面缓存
设置到 memcached 中.
在我的部署中, 我是通过设置一个 PAGE_CACHE_ID 给 php 的 fastcgi:
(在 fastcgi_params 中加入):
fastcgi_param PAGE_CACHE_ID $memcached_key;
php 程序一旦检测到这个环境变量,
那么会自动将当前页面的内容设置到 memcached 缓存中,由于是在 php 框架中自动作的,所以可以随时切换,不
需要修改.

缺点:
很多事情都不是那么完美, 对于 Nginx 的这个 module,其最大的问题就是不支持压缩,
因此,在使用 php 的 memcache 函数设置缓存时,必须取消压缩,否则将无法正确输出.
这样带来的一个小问题就是:
由于一般页面至少数 k 以上,这样对于内存的消耗是比较大.
因此, 在这个 module 支持压缩之前, 可以考虑备选方案:
使用独立的 FASTCGI 服务器来替换 memcache 模块.
通过 CPAN 模块,实现起来非常简单!

4、 Nginx 上的 Memcached 应用改进

由于 Nginx 的 memcached module 不支持压缩, 直接缓存页面, 非常浪费内存.
实验了 2 个方案:
1. 直接将压缩的数据输出,并添加一个 header:Content-Encoding:deflate
按说这是效率最高的,不过,很遗憾,Safari 和 FF 都可以正常显示页面,但是**** IE, 只能显示空白页面.
2. 取消 PECL 内置的 COMPRESS, 手动使用 PHP 的 gzcompress 函数进行压缩,这样,Nginx 可以通过添加 header, 直接输出 memcached 的内容,由浏览器自行解压.
缺点: 对于不支持 gzip 的客户端,将无法正确获得内容. 因此,最好还是判断一下客户端是否支持 gzip.这样, 下面的方案是比较好的选择.
3. 还是老办法,用 perl 写了 memcached 的 fastcgi, 独立运行. 使用 IO::Uncompress::AnyInflate 解压, 并输出结果. 这个还是不错的, 使用了独立的 fastcgi server 的形式, 效率不低. 而且只占用一个 Memcached 链接, 我也作了简单的统计, 可以监控缓存的命中率.
我使用了 Memcached::libmemcached 这个 CPAN module , 它使用了 libmemcached 这个最近最受关注的库. 效率比传统的 Client 要高效.
几个注意事项:
1. 部署时注意, 为了使用 error_page 进行转发, 需要 在 Nginx 相关配置中加入:
fastcgi_intercept_errors on;
默认是 off,因此, fastcgi 中的 404 等错误将不被 Nginx 进行重新转发.

```
fastcgi_pass unix:/tmp/Nginx-mcached.socket;
fastcgi_param GATEWAY_INTERFACE CGI/1.1;
fastcgi_param REQUEST_URI $request_uri;
fastcgi_param REQUEST_METHOD $request_method;
fastcgi_param CONTENT_TYPE $content_type;
fastcgi_param CONTENT_LENGTH $content_length;
fastcgi_param PATH_INFO $fastcgi_script_name;
fastcgi_param QUERY_STRING $query_string;
fastcgi_param DOCUMENT_URI $document_uri;
fastcgi_param DOCUMENT_ROOT $document_root;
fastcgi_param SERVER_PROTOCOL $server_protocol;
fastcgi_param REMOTE_ADDR $remote_addr;
fastcgi_param REMOTE_PORT $remote_port;
fastcgi_param SERVER_ADDR $server_addr;
fastcgi_param SERVER_PORT $server_port;
fastcgi_param SERVER_NAME $server_name;
```

需要缓存的 location:

```
location /v {
set $memcached_key $scheme://$host$request_uri;
include mcached.conf;
error_page 404 = /missing$request_uri;
}
```

在/missing 则重写到实际的后端去.

2. PHP PECL 压缩时使用了 deflate, 因此需要使用 inflate 进行解压,不可用 gunzip, 为了方便,可以用 IO::Uncompress::AnyInflate , 这样可以自动使用相应的算法.

5、 Nignx 配合 Memcached 提升 400%性能

原始文章: http://www.igvita.com/2008/02/11/Nginx-and-memcached-a-400-boost/

原文引用了 http://nubyonrails.com/articles/about-this-blog-memcached

和 http://blog.leetsoft.com/2007/5/22/the-secret-to-memcached

都是讲述怎样在 Rails 中使用 memcached 的（主要是 key 的设定方式和怎样对 memcached 的使用做类的包装)

文章在两篇引用文章的基础上直接将 memcached 的使用提升到 url 层，通过 Nginx 对 memcached 的原生支持，讲述了一种利用 url rewrite 机制来直接将 url 请求地址作为 key 给 memcached 做处理，用以提升访问速度。

首先在编译 Nginx 时要添加 memcached 模块编译，Nginx 的 memcached 模块配置参数说明见 http://wiki.codemongers.com/NginxHttpMemcachedModule

文章讲述利用 url rewrite 机制来做 key validate 的自动化处理,主要讲了一下 url rewrite 和 mime type 的配合问题。

本文启发

1.网站 url 地址要好规划，这样可以将不同类别的资源做分组，方便以后的统一处理，添加缓冲之类的。

2.怎样确定内容的修改频度,自动话的处理？如果是动态的对数据敏感新强的页面。

3.有没有方法可以在 url 层做 validate time 的规范话处理（资源的更新频度，每小时，每天等等)。

6、 Nginx 出现的 413 Request Entity Too Large 错误

这个错误一般在上传文件的时候出现，打开 Nginx 主配置文件 Nginx.conf，找到 http{}段，添加

```
client_max_body_size 8m;
```

要是跑 php 的话这个大小 client_max_body_size 要和 php.ini 中的如下值的最大值一致或者稍大，这样就不会因为提交数据大小不一致出现的错误。

```
post_max_size = 8M

upload_max_filesize = 2M
```

7、 解决 504 Gateway Time-out

504 Gateway Time-out 问题常见于使用 Nginx 作为 web server 的服务器的网站,遇到这个问题是在升级 discuz 论坛的时候遇到的，一般看来, 这种情况可能是由于 Nginx 默认的 fastcgi 进程响应的缓冲区太小造成的, 这将导致 fastcgi 进程被挂起, 如果你的 fastcgi 服务对这个挂起处理的不好, 那么最后就极有可能导致 504

Gateway Time-out
现在的网站，尤其某些论坛有大量的回复和很多内容的，一个页面甚至有几百 K，默认的 fastcgi 进程响应的缓冲区是 8K，我们可以设置大点。在 Nginx.conf 里，加入:

```
fastcgi_buffers 8 128k
```

这表示设置 fastcgi 缓冲区为 8×128k
当然如果您在进行某一项即时的操作，可能需要 Nginx 的超时参数调大点，例如设置成 60 秒:

```
send_timeout 60;
```

只是调整了这两个参数，结果就是没有再显示那个超时，可以说效果不错，但是也可能是由于其他的原因，目前关于 Nginx 的资料不是很多，很多事情都需要长期的经验累计才有结果，期待您的发现哈!

8、 Nginx 502 Bad Gateway 错误

1)、第一种方法：

Nginx 502 错误的原因比较多，是因为在代理模式下后端服务器出现问题引起的。可以肯定的是，这些错误都不是 Nginx 本身的问题，一定要从后端找原因！但 Nginx 把这些出错都揽在自己身上了，着实让 Nginx 的推广者备受置疑，毕竟从字眼上理解，bad gateway？不就是 bad Nginx 吗？让不了解的人看到，会直接把责任推在 Nginx 身上，希望 Nginx 下一个版本会把出错提示写稍微友好一些，至少不会是现在简单的一句 502 Bad Gateway，另外还不忘附上自己的大名。
502 错误最通常的出现情况就是后端主机当机，当然还有。在 upstream 配置里有这么一项配置：
proxy_next_upstream，这个配置指定了 Nginx 在从一个后端主机取数据遇到何种错误时会转到下一个后端主机，里头写上的就是会出现 502 的所有情况拉，默认是 error timeout，error 就是当机、断线之类的，timeout 就是读取堵塞超时，比较容易理解。一般是全写上的：

```
proxy_next_upstream error timeout invalid_header http_500 http_503;
```

不过现在可能我要去掉 http_500 这一项了，http_500 指定后端返回 500 错误时会转一个主机，后端的 jsp 出错的话，本来会打印一堆 stacktrace 的错误信息，现在被 502 取代了。但公司的程序员可不这么认为，他们认定是 Nginx 出现了错误，我实在没空跟他们解释 502 的原理了……
invalid_header 我也没认真查清到底指的什么，我也很想先把它弄下来。
503 错误就可以保留，因为后端通常是 apache resin，如果 apache 死机就是 error，但 resin 死机，仅仅是 503，所以还是有必要保留的。对 Nginx fastcgi 使用的情况，我现在用得不多，不熟就不乱说了，不过仍然可以肯定的是，问题同样不是出于 Nginx 本身！

2)、第二种方法：

```
large_client_header_buffers 4 32k;
```

加在 http 段中

9、 400 bad request 错误的原因和解决办法

Nginx 的 400 错误比较难查找原因，因为此错误并不是每次都会出现的，另外，出现错误的时候，通常在浏览器和日志里看不到任何有关提示。经长时间观察和大量试验查明，此乃 request header 过大所引起，request 过大，通常是由于 cookie 中写入了较大的值所引起。所幸在 Nginx 中是有办法解决这个问题：
在 Nginx.conf 中，将 client_header_buffer_size 和 large_client_header_buffers 都调大，可缓解此问题。

其中主要配置是 client_header_buffer_size 这一项，默认是 1k，所以 header 小于 1k 的话是不会出现问题的。
按我现在配置是：

```
client_header_buffer_size 16k;
large_client_header_buffers 4 64k;
```

这个配置可接收 16k 以下的 header，在浏览器中 cookie 的字节数上限会非常大，所以实在是不好去使用那最大值。
最好的解决办法当然是不要往 cookie 里写入太多的东西，不过如果是一个很大的网站，那么在一个二级域名写入了顶级域名下的 cookie 似乎是不好控制的，这需要制定一个规范来控制顶级域名的 cookie 写入量才可以解决得了。
这个可能也是 Nginx 的一个 bug，因为 buffer 这个词义上说为缓冲，也就是说，如果没取完的话，是会循环取直至取完的，但是 Nginx 并没有进行循环的动作直接返回了 400 错误。Nginx 的下一个版本可能会修正这个问题。

10、 Nginx Awstats 日志分析

http://www.sunnyu.com/?p=84

11、 Nginx Upload 上传模块

在Nginx网站的模块列表页中发现一个Nginx服务器的上传模块http://www.grid.net.ru/nginx/upload.en.html
该模块通过 Nginx 服务器来接受用户上传的文件，在 Nginx 接受完文件以后再转给后端的程序做处理。它自动分析客户端的上传请求，将上传的文件保存到 upload_store 所指向的目录位置. 然后这些文件信息将被从原始的请求中剔除，重新组装好上传参数后转到后端由 upload_pass 指定的位置去处理，这样就可以任意处理上传的文件。每一个上传的 file 字段值将可以由 upload_set_form_field 指定的值替换. 文件的内容可以由$upload_tmp_path 变量读到或简单的移到其他位置. 将文件删除由 upload_cleanup 指定控制。
upload_set_form_field 可以使用的几个变量

- $upload_field_name
  原始的文件字段
- $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
  上传的文件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

- $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

官方的设置举例

```
        # 上传表单应该提交到这个地址
        location /upload {
            # 将请求体转到这个位置
            upload_pass    /test;

            # 将上传的文件保存到这个目录下
            # 目录是被散列化的,应该存在子目录 0 1 2 3 4 5 6 7 8 9
            upload_store /tmp 1;

            # 允许上传的文件被用户 user 只读
            upload_store_access user:r;

            # 设置请求体的字段(添加自己后端处理的信息)
            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，可以正则表达式表示
            upload_pass_form_field "^submit$|^description$";
        }

        # 将请求转到后端的地址处理
        location /test {
            proxy_pass    http://localhost:8080;
        }
    }
```

将模块添加到 Nginx 中的方法，下载源代码解压后，为 nginx 配置额外模块（需要重新编译）：

```
[root@Chinarenservice oracle]# tar xvzf nginx_upload_module-2.0.7.tar.gz
[root@Chinarenservice oracle]# cd nginx
[root@Chinarenservice oracle]#./configure –add-module=/usr/local/nginx_upload_module-2.0.7
[root@Chinarenservice oracle]# make
[root@Chinarenservice oracle]# make install
```

12、 Nginx SSL 配置：

Nginx 支持 SSL 应用，在开始编译时需要提前编译 Openssl 包，然后再编译进 Nginx 支持：

1)、编译：

```
[root@Chinarenservice oracle]# tar -zxvf openssl-0.9.8g.tar.gz
```

```
[root@Chinarenservice oracle]# cd openssl-0.9.8g
[root@Chinarenservice openssl-0.9.8g]# ./config --prefix=/usr/local/openssl/
[root@Chinarenservice openssl-0.9.8g]# make
[root@Chinarenservice openssl-0.9.8g]# make install
[root@Chinarenservice openssl-0.9.8g]#cd ..
[root@Chinarenservice openssl-0.9.8g]# tar –zxvf nginx-0.7.32.tar.gz
[root@Chinarenservice openssl-0.9.8g]# cd nginx-0.7.32
[root@Chinarenservice nginx-0.7.32]# ./configure --prefix=/usr/local/nginx --with-http_ssl_module
--with-pcre=/freeke/soft/pcre-7.7 --with-http_stub_status_module --with-http_realip_module
--with-http_addition_module --with-http_sub_module --with-openssl=/freeke/soft/openssl-0.9.8g
```

2）、配置：

```
server {
        listen                  443;
        ssl                     on;
        ssl_certificate         /usr/local/nginx/conf/www.chinarenservice.com.cer;
        ssl_certificate_key     /usr/local/nginx/conf/www.chinarenservice.com.key;
        server_name             www.chinarenservice.com;
        access_log      off;
        location / {
                proxy_pass              http://api;
                proxy_redirect          http://www.chinarenservice.com/ /;
                proxy_set_header        Host $host;
                proxy_set_header        X-Real-IP $remote_addr;
                proxy_set_header        X-Forwarded-For $proxy_add_x_forwarded_for;
                }

        }
```

13、

十七、参考资料

http://user.qzone.qq.com/56802890