# Nginx学习笔记

参考：<https://www.nginx.com/resources/wiki/start/>

Edited by oliugian

## 起步

### Nginx安装

#### Ubuntu版

（其它系统参见<http://nginx.org/en/docs/install.html>）

Stable版本

下载密钥来签名nginx包和存储库（http://nginx.org/en/linux\_packages.html）

执行指令：

sudo apt-key add nginx\_signing.key

把下行代码中的codename替换成ubuntu发行名称，并加到 /etc/apt/sources.list结尾。

deb http://nginx.org/packages/ubuntu/ *codename* nginx

deb-src http://nginx.org/packages/ubuntu/ *codename* nginx

apt-get update

apt-get install nginx

### 命令管理

https://www.nginx.com/resources/wiki/start/topics/tutorials/commandline/

#### 启动Nginx

/usr/sbin/nginx

#### 基本命令

形如 nginx –v

|  |  |
| --- | --- |
| -?, -h | Print help.打印帮助 |
| -v | Print version.打印版本 |
| -V | Print NGINX version, compiler version and configure parameters.打印nginx版本，编译版本，配置信息 |
| -t | Don’t run, just test the configuration file. NGINX checks configuration for correct syntax and then try to open files referred in configuration.测试nginx配置信息是否正确，试着打开与配置关联的文件。 |
| -q | Suppress non-error messages during configuration testing. |
| -s signal | Send signal to a master process: stop, quit, reopen, reload. (version >= 0.7.53)发送信号给主进程 |
| -p prefix | Set prefix path (default: /usr/local/nginx/). (version >= 0.7.53)设置前缀地址 |
| -c filename | Specify which configuration file NGINX should use instead of the default.指定配置信息 |
| -gdirectives | Set [global](http://nginx.org/en/docs/http/ngx_http_core_module.html) directives. (version >= 0.7.4)设置全局指令 |

注意：nginx仅有很少的命令参数，它的配置全部由配置信息配置。

Nginx有两种方式来控制正在运行的系统

第一种 通过nginx –s signal来控制

* stop — fast shutdown
* quit — graceful shutdown
* reload — reloading the configuration file
* reopen — reopening the log files

第二中 根据NGINX主进程id来发送信号进行控制

ps –ax | grep nginx来获取ngnix进程id或者在配置信息中首部有指定pid地址，可查看。

可使用形如以下指令进行操作

kill -QUIT **$(** cat /usr/local/nginx/logs/nginx.pid **)**

|  |  |
| --- | --- |
| TERM, INT | Quick shutdown 快速关闭 |
| QUIT | Graceful shutdown 安全优雅的关闭 |
| KILL | Halts a stubborn process 强行关掉某个进程 |
| HUP | Configuration reload 重新加载配置信息，启动新的进程，安全关闭旧进程  Start the new worker processes with a new configuration  Gracefully shutdown the old worker processes |
| USR1 | Reopen the log files |
| USR2 | Upgrade Executable on the fly平滑升级 |
| WINCH | Gracefully shutdown the worker processes安全关闭工作经常 |

#### 使用信号设置配置

/usr/sbin/nginx -c /etc/nginx/nginx.conf

当nginx收到一个HUP指令时候，它会试图解析配置信息（被指定的那个，如果没指定，则使用默认的）。如果成功加载，试图使用新的配置信息，即重新打开log文件和监听的socket。如果成功，nginx运行新的工作进程和信号来安全的关闭旧进程。注意：旧的工作进程关闭监听的socket但是会继续给正在服务的客户端服务。当服务完所有的客户端，才关闭。如果nginx不能成功的加载新的配置信息，它将继续执行旧的配置信息。

#### Nginx平滑升级（USR2指令）

即nginx升级或者添加或移除服务器模块。你可以在没有服务器中断或者请求遗失的情况下进行。

步骤如下：

1. 用新的版本更换旧的二进制文件
2. 给主进程发送USR2信号

它会更改它的.pid文件为.pid.oldbin，然后执行新的二进制文件从而启动新的主进程和新的工作进程。

在这段时间，有两个nginx实例在运行，一起处理新来的请求。为了把旧的工作进程给淘汰掉，你必须发送WINCH指令给旧的主进程来使它安全的关闭。

在这之后旧的工作进程会全部退出，新的工作进程将处理新来的请求。

在这个时候你依然能还原旧的服务器，因为它还没有关闭自己的socket。它会执行下面几步：

1. 发送HUP信号给旧的主进程，这会在没有重新加载配置文件的情况下启动工作进程。
2. 发送QUIT信号给新的主进程来安全的退出它的工作进程。
3. 发送TERM信号给新的住精彩来强制它退出。
4. 如果因为某些原因新的工作进程没有退出，那么发送KILL信号给它们。

当一个新的主进程退出之后，旧的主进程移除它的.pid文件中.oldbin后缀。所有事都如最初。

如果更新成功，你想继续保持新的服务，发送QUIT信号给旧的主进程来关闭它。

### 陷阱和常见错误

https://www.nginx.com/resources/wiki/start/topics/tutorials/config\_pitfalls/

#### 不要使用chmod 777。可使用namei –om /path/to/check来查询权限。

#### 在location 模块的root

坏配置

**server** {

**server\_name** www.example.com;

**location** / {

**root** /var/www/nginx-default/;

*# [...]*

}

**location** /foo {

**root** /var/www/nginx-default/;

*# [...]*

}

**location** /bar {

**root** /var/www/nginx-default/;

*# [...]*

}

}

把root放到location模块可以正常运行，但是当你添加root模块到所有location，则没有被匹配到的location模块将没有root。好的配置如下：

**server** {

**server\_name** www.example.com;

**root** /var/www/nginx-default/;

**location** / {

*# [...]*

}

**location** /foo {

*# [...]*

}

**location** /bar {

*# [...]*

}

}

#### 太多index指令

坏配置：

**http** {

**index** index.php index.htm index.html;

**server** {

**server\_name** www.example.com;

**location** / {

**index** index.php index.htm index.html;

*# [...]*

}

}

**server** {

**server\_name** example.com;

**location** / {

**index** index.php index.htm index.html;

*# [...]*

}

**location** /foo {

**index** index.php;

*# [...]*

}

}

}

不用重复那么多行index，简单使用它一次就够了。它只需要在你的http{}模块里定义，然后将会继承下去。

好的配置：

**http** {

**index** index.php index.htm index.html;

**server** {

**server\_name** www.example.com;

**location** / {

*# [...]*

}

}

**server** {

**server\_name** example.com;

**location** / {

*# [...]*

}

**location** /foo {

*# [...]*

}

}

}

#### if是邪恶的

if指令在上下文中使用时候有问题，甚至可能导致segfaults。尽量不要使用它。

* 只有在[return](http://nginx.org/en/docs/http/ngx_http_rewrite_module.html?&_ga=1.114985991.1803309096.1455860144#return) 和[rewrite](http://nginx.org/en/docs/http/ngx_http_rewrite_module.html#rewrite) ... last上下文中if才能保证100%正确。

只有在合适的测试和了解if之后才使用它。

##### 怎么替换它

使用[try\_files](http://nginx.org/en/docs/http/ngx_http_core_module.html?&_ga=1.146826679.1803309096.1455860144#try_files)。某些情况下使用[return](http://nginx.org/en/docs/http/ngx_http_rewrite_module.html?&_ga=1.114985991.1803309096.1455860144#return) 和[rewrite](http://nginx.org/en/docs/http/ngx_http_rewrite_module.html#rewrite) ... last。某些情况下把ifs移动到server等级。（这里它是安全的因为只有rewrite模块指令被允许在里面）。

例如：以下设置可以用来安全地更改用来处理请求的location。

**location** / {

**error\_page** 418 = @other;

**recursive\_error\_pages** on;

**if** ($something) {

**return** 418;

}

*# some configuration*

**...**

}

location @other {

*# some other configuration*

**...**

}

在某些时候好，使用嵌入的脚本模块（[embedded perl](http://nginx.org/en/docs/http/ngx_http_perl_module.html), or various [NGINX 3rd Party Modules](https://www.nginx.com/resources/wiki/modules/))来处理。

##### 例子

这些例子来跟你解释为什么if是邪恶的，最后不要在尝试它。

*# Here is collection of unexpectedly buggy configurations to show that*

*# if inside location is evil.*

*# only second header will be present in response*

*# not really bug, just how it works*

**location** /only-one-if {

**set** $true 1;

**if** ($true) {

**add\_header** X-First 1;

}

**if** ($true) {

**add\_header** X-Second 2;

}

**return** 204;

}

*# request will be sent to backend without uri changed*

*# to '/' due to if*

**location** /proxy-pass-uri {

**proxy\_pass** http://127.0.0.1:8080/;

**set** $true 1;

**if** ($true) {

*# nothing*

}

}

*# try\_files wont work due to if*

**location** /if-try-files {

**try\_files** /file @fallback;

**set** $true 1;

**if** ($true) {

*# nothing*

}

}

*# nginx will SIGSEGV*

**location** /crash {

**set** $true 1;

**if** ($true) {

*# fastcgi\_pass here*

**fastcgi\_pass** 127.0.0.1:9000;

}

**if** ($true) {

*# no handler here*

}

}

*# alias with captures isn't correcly inherited into implicit nested*

*# location created by if*

**location** ~\* ^/if-and-alias/(?<file>.\*) {

**alias** /tmp/$file;

**set** $true 1;

**if** ($true) {

*# nothing*

}

}

BAD:

**server** {

**server\_name** example.com \*.example.com;

**if** ($host ~\* ^www\.(.+)) {

**set** $raw\_domain $1;

**rewrite** ^/(.\*)$ $raw\_domain/$1 permanent;

}

*# [...]*

}

**}**

这个例子中无论请求的是子域还是普通的网址，这个指令总是要检查host的值，这非常低效。

GOOD:

**server** {

**server\_name** www.example.com;

**return** 301 $scheme://example.com$request\_uri;

}

**server** {

**server\_name** example.com;

*# [...]*

}

检查文件是否存在

BAD:

**server** {

**root** /var/www/example.com;

**location** / {

**if** (!-f $request\_filename) {

**break**;

}

}

}

GOOD:

**server** {

**root** /var/www/example.com;

**location** / {

**try\_files** $uri $uri/ /index.html;

}

}

##### 为什么这个会发生而且一直没被解决呢？

Directive “if” is part of rewrite module which evaluates instructions imperatively. On the other hand, NGINX configuration in general is declarative. At some point due to users demand an attempt was made to enable some non-rewrite directives inside “if”, and this lead to situation we have now. It mostly works, but... see above.在另一方面，NGINX配置文件一般是声明的，在一些地方由于用户需要尝试去是的某些if中的non-rewritezhi指令。

看起来唯一的解决办法是使得if中的non-rewrite指令完全失效。这会破坏很多里面配置。

##### 建议：

如果你想使用if。请确保你明白它怎么工作。一些基本的想法可能会发现如[这里](http://agentzh.blogspot.com/2011/03/how-nginx-location-if-works.html)。做正确的测试。

#### Front controller模式的web应用程序

1.不同应用程序使用的参数名不同

2.有些程序甚至不需要查询字段，它们能直接读取request\_uri。

### 常见应用

#### 服务器名称

http://nginx.org/en/docs/http/server\_names.html

服务器名称由server\_name指令定义。它们可以用精确的名称，通配符或者正则表达式定义。如下可见：

server {

listen 80;

server\_name example.org www.example.org;

...

}

server {

listen 80;

server\_name \*.example.org;

...

}

server {

listen 80;

server\_name mail.\*;

...

}

server {

listen 80;

server\_name ~^(?<user>.+)\.example\.net$;

...

}

当使用名称进行虚拟服务器搜索，如果名称匹配不止一个指定名称。则第一个匹配的服务器将被选择。优先顺序如下：

1. 确切名称
2. 以\*开头的最长通配符名称
3. 以\*结尾的最长通配符名称
4. 第一个被匹配的正则表达式（在配置文件中出现的顺序）

##### 通配符名称

通配符名称可能包含\*在名字的开头或结尾，而且在一个.的边界。而正则表达式没有这种限制。”\*.example.org”不仅能匹配[www.example.org](http://www.example.org)还能匹配[www.sub.example.org](http://www.sub.example.org).

一个特殊的通配符”.example.org”，既能匹配确切地址”example.org”,还能匹配”\*.example.org”。

##### 正则表达式名称

Nginx使用的正则表达式和perl中的是兼容的。为了使用正则表达式，服务器名称要以波浪符号开头：

server\_name ~^www\d+\.example\.net$;

否则将被对待成确切地址或者通配符地址。不要忘了设置”^””$”锚，语法上不需要，可是逻辑上需要。注意域名的.需要被\转义。包含{}的字符应该被引用：

server\_name "~^(?<name>\w\d**{**1,3**}**+)\.example\.net$";

否则nginx会失败并展示错误信息：

directive "server\_name" is not terminated by ";" in ...

一个命名的正则表达式捕获后能被当成后面的变量：

server {

server\_name ~^(www\.)?(**?<domain>**.+)$;

location / {

root /sites/**$domain**;

}

}

PCRE包能命名捕获使用以下语法：

|  |  |
| --- | --- |
| ?<*name*> | Perl 5.10 compatible syntax, supported since PCRE-7.0 |
| ?'*name*' | Perl 5.10 compatible syntax, supported since PCRE-7.0 |
| ?P<*name*> | Python compatible syntax, supported since PCRE-4.0 |

如果nginx运行失败并展示如下：

pcre\_compile() failed: unrecognized character after (?< in ...

说明PCRE包过时了，”?P<name>”应该被取代。

命名捕获也可以以数字形式:

server {

server\_name ~^(www\.)?(.+)$;

location / {

root /sites/**$2**;

}

}

##### 其它名称（Miscellaneous names）

有很多可以被特殊处理的服务器名。

如果想在非默认的服务器模块上处理没有“host”头字段的请求，空名需要被指定：

server {

listen 80;

server\_name example.org www.example.org "";

...

}

如果在server模块中server\_name没有被指定，那么将使用空名作为server\_name.

注意：版本0.8.48以前使用hostname作为server\_name。

如果有人使用IP地址而不是域名发请求，则“host”请求头字段包含ip地址和请求，它将被使用ip地址作为server name的服务器处理：

server {

listen 80;

server\_name example.org

www.example.org

""

**192.168.1.1**

;

...

}

“\_”server\_name 如下：

server {

listen 80 default\_server;

server\_name \_;

return 444;

}

某个无效域名（不会与真正名称相交）

使用server\_name指令没有办法指定catch-all name或者default server。可以使用端口定义，如\*：80，\*：8080：

server {

listen 80;

listen 8080 default\_server;

server\_name example.net;

...

}

server {

listen 80 default\_server;

listen 8080;

server\_name example.org;

...

}

#### 使用nginx作为httpload balancer

http://nginx.org/en/docs/http/load\_balancing.html

##### 介绍

跨多个应用程序实例的负载均衡是优化资源利用率​​，最大限度地提高吞吐量，降低延迟，并确保容错配置一个常用的技术。它可以使用的nginx作为一个非常有效的HTTP负载平衡器将流量分配到多个应用服务器并提高性能，可扩展性和Web应用程序的可靠性。

##### 负载均衡办法

1. 轮询（默认）：发向应用服务器的请求以轮询的形式去分发。
2. 最少连接：下一个请求被分发到有最少活动连接的服务器上。
3. ip哈希：一个哈希的办法用来决定哪个服务器应该被选择来处理下一个请求。基于客户的IP地址来决定。

##### 默认负载均衡配置

一个最简易的负载均衡配置如下：

http {

upstream myapp1 {

server srv1.example.com;

server srv2.example.com;

server srv3.example.com;

}

server {

listen 80;

location / {

proxy\_pass http://myapp1;

}

}

}

以上配置中没有指定负载均衡办法，则默认为轮询。所有的请求被myapp1代理，nginx使用http负载均衡来分发请求。

在nginx中的反向代理包括对http，https，fastcgi，uwsgi，scgi和memcached的负载均衡。

为了为HTTPS配置负载均衡，仅需要”https”作为协议。

当为FastCGI，uwsgi,SCGI 或者 memcached时候配置负载均衡的时候，分别使用[fastcgi\_pass](http://nginx.org/en/docs/http/ngx_http_fastcgi_module.html#fastcgi_pass), [uwsgi\_pass](http://nginx.org/en/docs/http/ngx_http_uwsgi_module.html#uwsgi_pass), [scgi\_pass](http://nginx.org/en/docs/http/ngx_http_scgi_module.html#scgi_pass), 和 [memcached\_pass](http://nginx.org/en/docs/http/ngx_http_memcached_module.html#memcached_pass) 指令。

##### 最少连接负载均衡

另一个负载均衡的准则是最少连接，最少连接允许在某些请求需要更长时间去完成的情况下以更公平的方式去控制应用实例的负载。

通过使用最少连接夫在聚会的办法，nginx设法不要去通过过多的请求去超负载某个繁忙的服务器应用，而是把这些新的请求分发到稍微空闲一些的服务器上。

为了激活最少连接的负载均衡策略，只需要将[least\_conn](http://nginx.org/en/docs/http/ngx_http_upstream_module.html#least_conn)指令配置到server group中。

upstream myapp1 {

least\_conn;

server srv1.example.com;

server srv2.example.com;

server srv3.example.com;

}

##### Session 持久化（ip-hash）

要注意的是，通过配置轮询和最少连接的负载均衡，每个随后的客户请求可能被分发到不同服务器，并不能保证同个客户端一直指向同个服务器。

如果想要把一个客户端绑定到一个指定的服务器上，也就是是的客户端的session持久化，为了使用选择一个特定的服务器，可以使用ip-hash的负载均衡策略。

通过ip-hash的办法，客户的ip地址被用做hash的key值，来决定服务器组里的哪个服务器应该选择来处理这个客户的请求。这种办法确保同一个客户的请求始终发往同一个服务器，除非这个服务器不可用。

配置如下：

upstream myapp1 {

ip\_hash;

server srv1.example.com;

server srv2.example.com;

server srv3.example.com;

}

##### 加权负载均衡配置

通过使用服务器权重能更进一步的影响控制nginx的负载均衡算法。

在以上的例子中，没有配置服务器权重，这意味着所有指定的服务器将被一种特定的负载均衡办法来相同对待。

特别是在轮询的策略当中，它意味着请求或多或少的以比较均衡的方式发送给服务器，给他提供足够多的请求，然后这些请求会被以均衡的方式执行，并且快速完成。

当weight参数被指定到某一个server中是，这个weight权重就被当成负载均衡选择的一部分。配置如下：

upstream myapp1 {

server srv1.example.com weight=3;

server srv2.example.com;

server srv3.example.com;

}

在以上配置中，每五个新的请求中，3个会被发送到srv1，其它的分别得到一个。

在最少连接和ip-hash的负载均衡处理中同样可以使用权重的办法。（最近版本）

##### 健康检查

在nginx中实现的反向代理包括对服务器的健康检查。如果某个特定服务器的应答因为出错失败了，nginx会标志这个服务器在失败的状态，将会在一小段时间内避免接下来的进站请求被发送到这个服务器上。

[max\_fails](http://nginx.org/en/docs/http/ngx_http_upstream_module.html#server) 指令设置了在[fail\_timeout](http://nginx.org/en/docs/http/ngx_http_upstream_module.html#server)时间内，与服务器联系，连续尝试都不成功的次数。它的默认值为1；如果为0-；那么健康检查对于这个服务器就失去作用了。[fail\_timeout](http://nginx.org/en/docs/http/ngx_http_upstream_module.html#server)参数 决定了服务器多上时间将会被标志为失败状态。在系统失败的[fail\_timeout](http://nginx.org/en/docs/http/ngx_http_upstream_module.html#server)的时间间隔后，nginx会开始通过一个live的客户请求去优雅的探测服务器，如果探测成功，这个服务器会被标志为live状态。

##### Further reading

In addition, there are more directives and parameters that control server load balancing in nginx, e.g. [proxy\_next\_upstream](http://nginx.org/en/docs/http/ngx_http_proxy_module.html#proxy_next_upstream), [backup](http://nginx.org/en/docs/http/ngx_http_upstream_module.html#server),[down](http://nginx.org/en/docs/http/ngx_http_upstream_module.html#server), and [keepalive](http://nginx.org/en/docs/http/ngx_http_upstream_module.html#keepalive). For more information please check our [reference documentation](http://nginx.org/en/docs/).

Last but not least, [application load balancing](https://www.nginx.com/products/application-load-balancing/), [application health checks](https://www.nginx.com/products/application-health-checks/), [activity monitoring](https://www.nginx.com/products/live-activity-monitoring/) and [on-the-fly reconfiguration](https://www.nginx.com/products/on-the-fly-reconfiguration/) of server groups are available as part of our paid NGINX Plus subscriptions.

The following articles describe load balancing with NGINX Plus in more detail:

* [Load Balancing with NGINX and NGINX Plus](https://www.nginx.com/blog/load-balancing-with-nginx-plus/)
* [Load Balancing with NGINX and NGINX Plus part 2](https://www.nginx.com/blog/load-balancing-with-nginx-plus-part2/)

#### 配置HTTPS服务器

<http://nginx.org/en/docs/http/configuring_https_servers.html>

为了配置HTTPS服务器，ssl参数需要在server模块中监听的sockets端口处启动。在server 中的证书和密钥文件也需要被指定。如下配置：

server {

listen 443 **ssl**;

server\_name www.example.com;

ssl\_certificate **www.example.com.crt**;

ssl\_certificate\_key **www.example.com.key**;

ssl\_protocols TLSv1 TLSv1.1 TLSv1.2;

ssl\_ciphers HIGH:!aNULL:!MD5;

...

}

服务器证书是一个公有的实体，它会发送到每个连接到服务器的客户端。而密钥是一个安全的实体，应该被存在一个限制访问的文件中，但是对于主进程来说是访问的。密钥也能与证书被存在同一个文件中：

ssl\_certificate www.example.com.cert;

ssl\_certificate\_key www.example.com.cert;

这种情况下，对于该文件的访问权利也需要被限制，及时证书和密钥被存在同一个文件中，只有证书会被发送回到客户中。

[ssl\_protocols](http://nginx.org/en/docs/http/ngx_http_ssl_module.html#ssl_protocols) 和[ssl\_ciphers](http://nginx.org/en/docs/http/ngx_http_ssl_module.html#ssl_ciphers)指令被用来限制连接到只含有更高版本和密钥的SSL/TLS 。默认情况下，nginx使用“ssl\_protocols TLSv1 TLSv1.1 TLSv1.2” and “ssl\_ciphers HIGH:!aNULL:!MD5”, 所以明确的去配置它们是没有必要的，注意这些指令的默认值[changed](http://nginx.org/en/docs/http/configuring_https_servers.html#compatibility)过几次。

##### HTTPS 服务器优化

SSL操作消耗额外的cpu资源。在多处理器系统中，不少于可用cpu核心数量的工作进程将会运行。Cpu中最忙的操作是ssl握手。有两个方式来减少每个客户进行这些操作的数量。第一个种方式是启用keepalive连接，来通过一个连接来发送多个请求。第二种方式是重用ssl会话参数来避免ssl在平行和接下来连接中的握手。这些会话被存储在一个被工作进程所共享的ssl 会话高速缓存中。可以通过[ssl\_session\_cache](http://nginx.org/en/docs/http/ngx_http_ssl_module.html#ssl_session_cache)指令来配置它。一兆的高速缓存里包含大概4000个会话。默认的高速缓存超时时间为5分钟，可以通过[ssl\_session\_timeout](http://nginx.org/en/docs/http/ngx_http_ssl_module.html#ssl_session_timeout)指令来增加它。下面是一个拥有10兆共享会话高速缓存多核心系统的优化配置：

**worker\_processes auto**;

http {

**ssl\_session\_cache shared:SSL:10m**;

**ssl\_session\_timeout 10m**;

server {

listen 443 ssl;

server\_name www.example.com;

**keepalive\_timeout 70**;

ssl\_certificate www.example.com.crt;

ssl\_certificate\_key www.example.com.key;

ssl\_protocols TLSv1 TLSv1.1 TLSv1.2;

ssl\_ciphers HIGH:!aNULL:!MD5;

...

##### SSL证书链

一些浏览器可能报告一个由知名认证机构认证的的证书，但是其它的可能会不出问题的接受这个证书。这是因为认证机构使用了中间证书来认证服务器证书，这个中间证书不存在于被广泛知道的被分发到特定浏览器的可信证书中。在这种情况下，当局提供了可以被级联到认证服务器证书中的证书捆绑。在文件中服务器证书一定要在链似证书之前。

如

$ cat www.example.com.crt bundle.crt > www.example.com.chained.crt

这个文件要在 [ssl\_certificate](http://nginx.org/en/docs/http/ngx_http_ssl_module.html#ssl_certificate) 中指定。

server {

listen 443 ssl;

server\_name www.example.com;

ssl\_certificate www.example.com.chained.crt;

ssl\_certificate\_key www.example.com.key;

...

}

如果服务器证书和束级联的顺序错误，nginx会无法启动，并展示下列错误信息：

SSL\_CTX\_use\_PrivateKey\_file(" ... /www.example.com.key") failed

(SSL: error:0B080074:x509 certificate routines:

X509\_check\_private\_key:key values mismatch)

因为nginx会试图使用束的第一个证书而不是服务器的证书。

浏览器常常保存它们接收到的被认证的中间证书，所以这些浏览器可能已经有了这些需要的中间证书，也可能不会报告问题。为了保证服务器发送了完整的证书链，可使用openssl的命令行：

$ openssl s\_client -connect www.godaddy.com:443

...

Certificate chain

0 s:/C=US/ST=Arizona/L=Scottsdale/1.3.6.1.4.1.311.60.2.1.3=US

/1.3.6.1.4.1.311.60.2.1.2=AZ/O=GoDaddy.com, Inc

/OU=MIS Department/**CN=www.GoDaddy.com**

/serialNumber=0796928-7/2.5.4.15=V1.0, Clause 5.(b)

i:/C=US/ST=Arizona/L=Scottsdale/O=GoDaddy.com, Inc.

/OU=http://certificates.godaddy.com/repository

/CN=Go Daddy Secure Certification Authority

/serialNumber=07969287

1 s:/C=US/ST=Arizona/L=Scottsdale/O=GoDaddy.com, Inc.

/OU=http://certificates.godaddy.com/repository

/CN=Go Daddy Secure Certification Authority

/serialNumber=07969287

i:/C=US/O=The Go Daddy Group, Inc.

/OU=Go Daddy Class 2 Certification Authority

2 s:/C=US/O=The Go Daddy Group, Inc.

/OU=Go Daddy Class 2 Certification Authority

i:/L=ValiCert Validation Network/O=**ValiCert, Inc.**

/OU=ValiCert Class 2 Policy Validation Authority

/CN=http://www.valicert.com//emailAddress=info@valicert.com

...

在上面例子中，[www.godaddy.com服务器证书#0](http://www.godaddy.com服务器证书#0) 的”s”主题由是证书#1主题的发行者（”i”），而它由广为人知的发行者 ValiCert，Inc认证。它的证书被存在浏览器的内建证书跟基地中。

如果证书链没有被添加的话，那么只有服务器证书#0能被看见。

##### 单一的 http/https服务器

我们可以配置一个服务器使它能同时处理http和https的请求：

server {

listen 80;

listen 443 ssl;

server\_name www.example.com;

ssl\_certificate www.example.com.crt;

ssl\_certificate\_key www.example.com.key;

...

}

先于0.7.14的SSL版本不能有选择的给单独的sockets监听启用。以上的配置，SSL只能通过[ssl](http://nginx.org/en/docs/http/ngx_http_ssl_module.html#ssl)指令来启用全部服务器，使它可以建立成一个HTTP/HTTPS服务器。监听指令的ssl参数被添加去解决这个问题。这种ssl指令的用法在现在是鼓励的。

##### 基于名称的HTTPS服务器

配置两个或更多服务器的HTTPS监听一个IP地址时，一个常见的​​问题就出现了：

server {

listen 443 ssl;

server\_name www.example.com;

ssl\_certificate www.example.com.crt;

...

}

server {

listen 443 ssl;

server\_name www.example.org;

ssl\_certificate www.example.org.crt;

...

}

使用该配置，一个浏览器将接收到默认服务器的证书，即[www.example.com](http://www.example.com)而忽略请求的服务器名的。这是通过SSL协议的行为造成的。SSL连接是浏览器发送一个HTTP请求之前建立，nginx不知道被请求的服务器的名称。因此，它可能只提供默认的服务器的证书。

解决这个问题的最古老和最可靠的方法是为每一个HTTPS服务器分配一个独立的IP地址：

server {

listen 192.168.1.1:443 ssl;

server\_name www.example.com;

ssl\_certificate www.example.com.crt;

...

}

server {

listen 192.168.1.2:443 ssl;

server\_name www.example.org;

ssl\_certificate www.example.org.crt;

...

}

##### 有多个名称的SSL证书

有别的办法允许在几个HTTPS服务器之间分享单独ip地址。可是，这些办法都有自己的缺点。其中一个是在SubjectAltName证书区域当中使用多个名称的证书，例如[www.example.com](http://www.example.com) 和 [www.example.org](http://www.example.org)。可是SubjectAltName的长度是受限制的。

另一个办法是给证书使用通配符的名字，例如\*.example.org.一个通配证书能保护特定域名的所有子域，但仅在一个等级上。这个证书匹配[www.example.org,但是不匹配example.org](http://www.example.org,但是不匹配example.org)和[www.sub.example.org](http://www.sub.example.org).这两个方法可以结合在一起。一个证书在SubjectAltNameZ区域中可以包含确切的和通配的名字。例如，example.org和\*.example.org。

更好的做法是使用几个名字的认证文件，并把它的私钥文件放在配置文件http等级，使它可以集成到所有的服务器：

ssl\_certificate common.crt;

ssl\_certificate\_key common.key;

server {

listen 443 ssl;

server\_name www.example.com;

...

}

server {

listen 443 ssl;

server\_name www.example.org;

...

}

##### 服务器名称的征兆

一个更通用的在一个单一ip地址上运行多个https服务器的解决方案是[TLS Server Name Indication extension](http://en.wikipedia.org/wiki/Server_Name_Indication)。它允许浏览器在SSL握手期间发送一个请求的服务器名。因此，服务器知道它需要使用哪个证书为这次连接。可是，SNI限制服务器的支持。目前它支持的浏览器版本如下：

* Opera 8.0;
* MSIE 7.0 (but only on Windows Vista or higher);
* Firefox 2.0 and other browsers using Mozilla Platform rv:1.8.1;
* Safari 3.2.1 (Windows version supports SNI on Vista or higher);
* and Chrome (Windows version supports SNI on Vista or higher, too).

只有域名能通过SNI，可是，有些浏览器可能错误的把服务器的ip当成它的名字发送过来，如果一个请求包含ip地址。

为了在nginx中使用SNI，它必须支持nginx二进制文件中被编译生成的openSSL lib和运行过程中动态链接进来的lib。OpenSSL支持SNI 0.9.8f之后的版本，之前的版本需要它配置“—enable-tlsext”的选项。OpenSSL 0.9.8j之后的版本这个选项变为默认。如果nginx有SNI的支持，则nginx使用“-V”的选项能展示出这个。

$ nginx -V

...

TLS SNI support enabled

...

可是，如果SNI-enabled动态链接到nginx中的OpenSSL lib 没有SNI的支持，nginx会展示一些警告：

nginx was built with SNI support, however, now it is linked

dynamically to an OpenSSL library which has no tlsext support,

therefore SNI is not available

##### 兼容性

* The SNI support status has been shown by the “-V” switch since 0.8.21 and 0.7.62.
* The ssl parameter of the [listen](http://nginx.org/en/docs/http/ngx_http_core_module.html#listen) directive has been supported since 0.7.14. Prior to 0.8.21 it could only be specified along with the default parameter.
* SNI has been supported since 0.5.32.
* The shared SSL session cache has been supported since 0.5.6.
* Version 1.9.1 and later: the default SSL protocols are TLSv1, TLSv1.1, and TLSv1.2 (if supported by the OpenSSL library).
* Version 0.7.65, 0.8.19 and later: the default SSL protocols are SSLv3, TLSv1, TLSv1.1, and TLSv1.2 (if supported by the OpenSSL library).
* Version 0.7.64, 0.8.18 and earlier: the default SSL protocols are SSLv2, SSLv3, and TLSv1.
* Version 1.0.5 and later: the default SSL ciphers are “HIGH:!aNULL:!MD5”.
* Version 0.7.65, 0.8.20 and later: the default SSL ciphers are “HIGH:!ADH:!MD5”.
* Version 0.8.19: the default SSL ciphers are “ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM”.
* Version 0.7.64, 0.8.18 and earlier: the default SSL ciphers are  
  “ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP”.

#### 转换重写规则

在托管中的人们只使用Apache的.htaccess 文件来配置所有的事情。通常把如下的规则：

RewriteCond %{HTTP\_HOST} example.org

RewriteRule (.\*) <http://www.example.org$1>

转化为

server {

listen 80;

server\_name www.example.org example.org;

if ($http\_host = example.org) {

rewrite (.\*) http://www.example.org$1;

}

...

}

这是错误，无效的办法。正确的办法是为example.org定义一个别的服务器：

server {

listen 80;

server\_name example.org;

return 301 http://www.example.org$request\_uri;

}

server {

listen 80;

server\_name www.example.org;

...

}

在版本0.9.1之前的版本，重定向被写成：

rewrite ^ http://www.example.org$request\_uri?;

另一个例子，代替了“upside-down”的逻辑，不仅仅是example.com和www.example.com

RewriteCond %{HTTP\_HOST} !example.com

RewriteCond %{HTTP\_HOST} !www.example.com

RewriteRule (.\*) <http://www.example.com$1>

可以这样简单转化：

server {

listen 80;

server\_name example.com www.example.com;

...

}

server {

listen 80 default\_server;

server\_name \_;

return 301 http://example.com$request\_uri;

}

在版本0.9.1之前的版本，重定向被写成：

rewrite ^ http://www.example.org$request\_uri?;

##### 转换混合规则

传统的混合规则：

DocumentRoot /var/www/myapp.com/current/public

RewriteCond %{DOCUMENT\_ROOT}/system/maintenance.html -f

RewriteCond %{SCRIPT\_FILENAME} !maintenance.html

RewriteRule ^.\*$ %{DOCUMENT\_ROOT}/system/maintenance.html [L]

RewriteCond %{REQUEST\_FILENAME} -f

RewriteRule ^(.\*)$ $1 [QSA,L]

RewriteCond %{REQUEST\_FILENAME}/index.html -f

RewriteRule ^(.\*)$ $1/index.html [QSA,L]

RewriteCond %{REQUEST\_FILENAME}.html -f

RewriteRule ^(.\*)$ $1.html [QSA,L]

RewriteRule ^/(.\*)$ balancer://mongrel\_cluster%{REQUEST\_URI} [P,QSA,L]

应该被转换为：

location / {

root /var/www/myapp.com/current/public;

try\_files /system/maintenance.html

$uri $uri/index.html $uri.html

@mongrel;

}

location @mongrel {

proxy\_pass http://mongrel;

}

#### WebSocket 代理

为了使客户端和服务器的一个连接由http/1.1转为websocket，http/1.1中[protocol switch](http://tools.ietf.org/html/rfc2616#section-14.42)的机制是可用的。

但是这里有块小问题：因为“Upgrade”是一个[hop-by-hop](http://tools.ietf.org/html/rfc2616#section-13.5.1) 头部，它不从客服端传送到代理的服务器。通过代理，客户端可以使用connect办法避免这个问题。可是它在反响代理中不能使用，因为客户端不知道有任何代理服务器，而且在代理服务器上需要一个特殊的运行程序。

自从版本1.3.13之后，nginx实现一块特殊的操作模块，它运行在客户端和代理服务器之间建立隧道，如果这个代理服务器返回一个101代码（switching协议），客户端在请求中的通过“Upgrade”头来请求一个协议转换。

由上可知，包含“upgrade”和“connection”的hop-by-hop头不会从客户端通向代理服务器，因此，为了代理服务器知道客户端的把协议转换为websocket的意图，它们的头需要被明确指定：

location /chat/ {

proxy\_pass http://backend;

proxy\_http\_version 1.1;

proxy\_set\_header Upgrade $http\_upgrade;

proxy\_set\_header Connection "upgrade";

}

一个更复杂的例子如下，发向代理服务器的请求中的“Connection”头区域取决于客户请求头中“Upgrade”区域是否存在：

http {

map $http\_upgrade $connection\_upgrade {

default upgrade;

'' close;

}

server {

...

location /chat/ {

proxy\_pass http://backend;

proxy\_http\_version 1.1;

proxy\_set\_header Upgrade $http\_upgrade;

proxy\_set\_header Connection $connection\_upgrade;

}

}

默认情况下，链接将会关闭，如果代理服务器在60秒内不在发送任何数据。这个超时时间能使用[proxy\_read\_timeout](http://nginx.org/en/docs/http/ngx_http_proxy_module.html#proxy_read_timeout)指令更改。另外，如果连接还存活的话，代理服务器可以被配置为周期行的发送WebSocket ping帧来复位超时和检查。

## 模块结构及处理过程

### Nginx架构初探

参见：<http://tengine.taobao.org/book/chapter_02.html>

nginx在启动后，会有一个master进程和多个worker进程。master进程主要用来管理worker进程，包含：接收来自外界的信号，向各worker进程发送信号，监控worker进程的运行状态，当worker进程退出后(异常情况下)，会自动重新启动新的worker进程。而基本的网络事件，则是放在worker进程中来处理了。

多个worker进程之间是对等的，他们同等竞争来自客户端的请求，各进程互相之间是独立的。一个请求，只可能在一个worker进程中处理，一个worker进程，不可能处理其它进程的请求。worker进程的个数是可以设置的，一般我们会设置与机器cpu核数一致，这里面的原因与nginx的进程模型以及事件处理模型是分不开的。



#### 操作nginx

master来管理worker进程，所以我们只需要与master进程通信就行了。master进程会接收来自外界发来的信号，再根据信号做不同的事情。

1.所以我们要控制nginx，只需要通过kill向master进程发送信号就行了。比如kill -HUP pid。首先master进程在接到信号后，会先重新加载配置文件，然后再启动新的worker进程，并向所有老的worker进程发送信号，告诉他们可以光荣退休了。新的worker在启动后，就开始接收新的请求，而老的worker在收到来自master的信号后，就不再接收新的请求，并且在当前进程中的所有未处理完的请求处理完成后，再退出。

2. nginx在0.8版本之后，引入了一系列命令行参数，来方便我们管理。比如，./nginx -s reload，就是来重启nginx，./nginx -s stop，就是来停止nginx的运行。如何做到的呢？我们还是拿reload来说，我们看到，执行命令时，我们是启动一个新的nginx进程，而新的nginx进程在解析到reload参数后，就知道我们的目的是控制nginx来重新加载配置文件了，它会向master进程发送信号，然后接下来的动作，就和我们直接向master进程发送信号一样了。、

#### Worker进程怎么处理请求

##### Nginx进程模型

当我们提供80端口的http服务时，一个连接请求过来，每个进程都有可能处理这个连接，怎么做到的呢？首先，每个worker进程都是从master进程fork过来，在master进程里面，先建立好需要listen的socket（listenfd）之后，然后再fork出多个worker进程。所有worker进程的listenfd会在新连接到来时变得可读，为保证只有一个进程处理该连接，所有worker进程在注册listenfd读事件前抢accept\_mutex，抢到互斥锁的那个进程注册listenfd读事件，在读事件里调用accept接受该连接。当一个worker进程在accept这个连接之后，就开始读取请求，解析请求，处理请求，产生数据后，再返回给客户端，最后才断开连接，这样一个完整的请求就是这样的了。我们可以看到，一个请求，完全由worker进程来处理，而且只在一个worker进程中处理。

nginx采用这种进程模型有什么好处呢？

当然，好处肯定会很多了。首先，对于每个worker进程来说，独立的进程，不需要加锁，所以省掉了锁带来的开销，同时在编程以及问题查找时，也会方便很多。其次，采用独立的进程，可以让互相之间不会影响，一个进程退出后，其它进程还在工作，服务不会中断，master进程则很快启动新的worker进程。当然，worker进程的异常退出，肯定是程序有bug了，异常退出，会导致当前worker上的所有请求失败，不过不会影响到所有请求，所以降低了风险。

##### Nginx事件处理模型

nginx采用了异步非阻塞的方式来处理请求，也就是说，nginx是可以同时处理成千上万个请求的。

注意，别增加无谓的上下文切换。所以，在nginx里面，最忌讳阻塞的系统调用了。

什么是异步非阻塞？

非阻塞就是，事件没有准备好，马上返回EAGAIN，告诉你，事件还没准备好呢，你慌什么，过会再来吧。

虽然不阻塞了，但你得不时地过来检查一下事件的状态，你可以做更多的事情了，但带来的开销也是不小的。所以，才会有了异步非阻塞的事件处理机制，具体到系统调用就是像select/poll/epoll/kqueue这样的系统调用。它们提供了一种机制，让你可以同时监控多个事件，调用他们是阻塞的，但可以设置超时时间，在超时时间之内，如果有事件准备好了，就返回。

这种机制正好解决了我们上面的两个问题，拿epoll为例(在后面的例子中，我们多以epoll为例子，以代表这一类函数)，当事件没准备好时，放到epoll里面，事件准备好了，我们就去读写，当读写返回EAGAIN时，我们将它再次加入到epoll里面。这样，只要有事件准备好了，我们就去处理它，只有当所有事件都没准备好时，才在epoll里面等着。这样，我们就可以并发处理大量的并发了，当然，这里的并发请求，是指未处理完的请求，线程只有一个，所以同时能处理的请求当然只有一个了，只是在请求间进行不断地切换而已，切换也是因为异步事件未准备好，而主动让出的。这里的切换是没有任何代价，你可以理解为循环处理多个准备好的事件，事实上就是这样的。与多线程相比，这种事件处理方式是有很大的优势的，不需要创建线程，每个请求占用的内存也很少，没有上下文切换，事件处理非常的轻量级。并发数再多也不会导致无谓的资源浪费（上下文切换）。更多的并发数，只是会占用更多的内存而已。

我们之前说过，推荐设置worker的个数为cpu的核数，在这里就很容易理解了，更多的worker数，只会导致进程来竞争cpu资源了，从而带来不必要的上下文切换。

Nginx怎么处理信号和定时器？

首先，信号的处理。对nginx来说，有一些特定的信号，代表着特定的意义。信号会中断掉程序当前的运行，在改变状态后，继续执行。如果是系统调用，则可能会导致系统调用的失败，需要重入。关于信号的处理，大家可以学习一些专业书籍，这里不多说。对于nginx来说，如果nginx正在等待事件（epoll\_wait时），如果程序收到信号，在信号处理函数处理完后，epoll\_wait会返回错误，然后程序可再次进入epoll\_wait调用。

另外，再来看看定时器。由于epoll\_wait等函数在调用的时候是可以设置一个超时时间的，所以nginx借助这个超时时间来实现定时器。nginx里面的定时器事件是放在一颗维护定时器的红黑树里面，每次在进入epoll\_wait前，先从该红黑树里面拿到所有定时器事件的最小时间，在计算出epoll\_wait的超时时间后进入epoll\_wait。所以，当没有事件产生，也没有中断信号时，epoll\_wait会超时，也就是说，定时器事件到了。这时，nginx会检查所有的超时事件，将他们的状态设置为超时，然后再去处理网络事件。由此可以看出，当我们写nginx代码时，在处理网络事件的回调函数时，通常做的第一个事情就是判断超时，然后再去处理网络事件。

### Nginx基本概念

#### connection

在nginx中connection就是对tcp连接的封装，其中包括连接的socket，读事件，写事件。

结合一个tcp连接的生命周期，我们看看nginx是如何处理一个连接的。首先，nginx在启动时，会解析配置文件，得到需要监听的端口与ip地址，然后在nginx的master进程里面，先初始化好这个监控的socket(创建socket，设置addrreuse等选项，绑定到指定的ip地址端口，再listen)，然后再fork出多个子进程出来，然后子进程会竞争accept新的连接。此时，客户端就可以向nginx发起连接了。当客户端与服务端通过三次握手建立好一个连接后，nginx的某一个子进程会accept成功，得到这个建立好的连接的socket，然后创建nginx对连接的封装，即ngx\_connection\_t结构体。接着，设置读写事件处理函数并添加读写事件来与客户端进行数据的交换。最后，nginx或客户端来主动关掉连接，到此，一个连接就寿终正寝了。

当然，nginx也是可以作为客户端来请求其它server的数据的（如upstream模块），此时，与其它server创建的连接，也封装在ngx\_connection\_t中。作为客户端，nginx先获取一个ngx\_connection\_t结构体，然后创建socket，并设置socket的属性（ 比如非阻塞）。然后再通过添加读写事件，调用connect/read/write来调用连接，最后关掉连接，并释放ngx\_connection\_t。

##### nginx进程会有一个连接数的最大上限

这个上限与系统对fd的限制不一样。在操作系统中，通过ulimit -n，我们可以得到一个进程所能够打开的fd的最大数，即nofile，因为每个socket连接会占用掉一个fd，所以这也会限制我们进程的最大连接数，当然也会直接影响到我们程序所能支持的最大并发数，当fd用完后，再创建socket时，就会失败。nginx通过设置worker\_connectons来设置每个进程支持的最大连接数。如果该值大于nofile，那么实际的最大连接数是nofile，nginx会有警告。nginx在实现时，是通过一个连接池来管理的，每个worker进程都有一个独立的连接池，连接池的大小是worker\_connections。这里的连接池里面保存的其实不是真实的连接，它只是一个worker\_connections大小的一个ngx\_connection\_t结构的数组。并且，nginx会通过一个链表free\_connections来保存所有的空闲ngx\_connection\_t，每次获取一个连接时，就从空闲连接链表中获取一个，用完后，再放回空闲连接链表里面。

##### 最大连接数

对于HTTP请求本地资源来说，能够支持的最大并发数量是worker\_connections \* worker\_processes，而如果是HTTP作为反向代理来说，最大并发数量应该是worker\_connections \* worker\_processes/2。因为作为反向代理服务器，每个并发会建立与客户端的连接和与后端服务的连接，会占用两个连接。

##### 多个进程竞争请求连接的方式

nginx的处理得先打开accept\_mutex选项，此时，只有获得了accept\_mutex的进程才会去添加accept事件，也就是说，nginx会控制进程是否添加accept事件。nginx使用一个叫ngx\_accept\_disabled的变量来控制是否去竞争accept\_mutex锁。

在第一段代码中，计算ngx\_accept\_disabled的值，这个值是nginx单进程的所有连接总数的八分之一，减去剩下的空闲连接数量，得到的这个ngx\_accept\_disabled有一个规律，当剩余连接数小于总连接数的八分之一时，其值才大于0，而且剩余的连接数越小，这个值越大。再看第二段代码，当ngx\_accept\_disabled大于0时，不会去尝试获取accept\_mutex锁，并且将ngx\_accept\_disabled减1，于是，每次执行到此处时，都会去减1，直到小于0。不去获取accept\_mutex锁，就是等于让出获取连接的机会，很显然可以看出，当空余连接越少时，ngx\_accept\_disable越大，于是让出的机会就越多，这样其它进程获取锁的机会也就越大。不去accept，自己的连接就控制下来了，其它进程的连接池就会得到利用，这样，nginx就控制了多进程间连接的平衡了。

ngx\_accept\_disabled = ngx\_cycle->connection\_n / 8

- ngx\_cycle->free\_connection\_n;

if (ngx\_accept\_disabled > 0) {

ngx\_accept\_disabled--;

} else {

if (ngx\_trylock\_accept\_mutex(cycle) == NGX\_ERROR) {

return;

}

if (ngx\_accept\_mutex\_held) {

flags |= NGX\_POST\_EVENTS;

} else {

if (timer == NGX\_TIMER\_INFINITE

|| timer > ngx\_accept\_mutex\_delay)

{

timer = ngx\_accept\_mutex\_delay;

}

}

}

#### request

这节我们讲request，在nginx中我们指的是http请求，具体到nginx中的数据结构是ngx\_http\_request\_t。ngx\_http\_request\_t是对一个http请求的封装。 我们知道，一个http请求，包含请求行、请求头、请求体、响应行、响应头、响应体。

http请求是典型的请求-响应类型的的网络协议，而http是文件协议，所以我们在分析请求行与请求头，以及输出响应行与响应头，往往是一行一行的进行处理。如果我们自己来写一个http服务器，通常在一个连接建立好后，客户端会发送请求过来。然后我们读取一行数据，分析出请求行中包含的method、uri、http\_version信息。然后再一行一行处理请求头，并根据请求method与请求头的信息来决定是否有请求体以及请求体的长度，然后再去读取请求体。得到请求后，我们处理请求产生需要输出的数据，然后再生成响应行，响应头以及响应体。在将响应发送给客户端之后，一个完整的请求就处理完了。当然这是最简单的webserver的处理方式，其实nginx也是这样做的，只是有一些小小的区别，比如，当请求头读取完成后，就开始进行请求的处理了。nginx通过ngx\_http\_request\_t来保存解析请求与输出响应相关的数据。

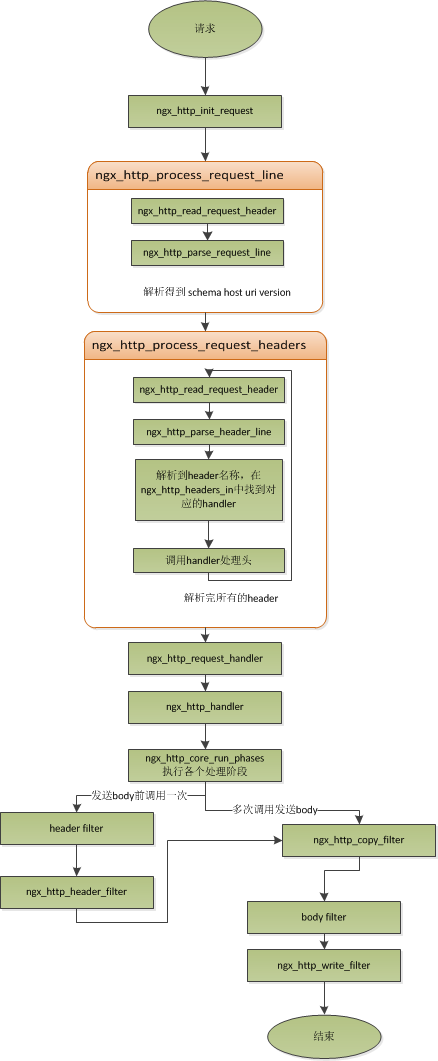
那接下来，简要讲讲nginx是如何处理一个完整的请求的。对于nginx来说，一个请求是从ngx\_http\_init\_request开始的，在这个函数中，会设置读事件为ngx\_http\_process\_request\_line，也就是说，接下来的网络事件，会由ngx\_http\_process\_request\_line来执行。从ngx\_http\_process\_request\_line的函数名，我们可以看到，这就是来处理请求行的，正好与之前讲的，处理请求的第一件事就是处理请求行是一致的。通过ngx\_http\_read\_request\_header来读取请求数据。然后调用ngx\_http\_parse\_request\_line函数来解析请求行。nginx为提高效率，采用状态机来解析请求行，而且在进行method的比较时，没有直接使用字符串比较，而是将四个字符转换成一个整型，然后一次比较以减少cpu的指令数，这个前面有说过。很多人可能很清楚一个请求行包含请求的方法，uri，版本，却不知道其实在请求行中，也是可以包含有host的。比如一个请求GET <http://www.taobao.com/uri> HTTP/1.0这样一个请求行也是合法的，而且host是www.taobao.com，这个时候，nginx会忽略请求头中的host域，而以请求行中的这个为准来查找虚拟主机。另外，对于对于http0.9版来说，是不支持请求头的，所以这里也是要特别的处理。所以，在后面解析请求头时，协议版本都是1.0或1.1。整个请求行解析到的参数，会保存到ngx\_http\_request\_t结构当中。

在解析完请求行后，nginx会设置读事件的handler为ngx\_http\_process\_request\_headers，然后后续的请求就在ngx\_http\_process\_request\_headers中进行读取与解析。ngx\_http\_process\_request\_headers函数用来读取请求头，跟请求行一样，还是调用ngx\_http\_read\_request\_header来读取请求头，调用ngx\_http\_parse\_header\_line来解析一行请求头，解析到的请求头会保存到ngx\_http\_request\_t的域headers\_in中，headers\_in是一个链表结构，保存所有的请求头。而HTTP中有些请求是需要特别处理的，这些请求头与请求处理函数存放在一个映射表里面，即ngx\_http\_headers\_in，在初始化时，会生成一个hash表，当每解析到一个请求头后，就会先在这个hash表中查找，如果有找到，则调用相应的处理函数来处理这个请求头。比如:Host头的处理函数是ngx\_http\_process\_host。

当nginx解析到两个回车换行符时，就表示请求头的结束，此时就会调用ngx\_http\_process\_request来处理请求了。ngx\_http\_process\_request会设置当前的连接的读写事件处理函数为ngx\_http\_request\_handler，然后再调用ngx\_http\_handler来真正开始处理一个完整的http请求。这里可能比较奇怪，读写事件处理函数都是ngx\_http\_request\_handler，其实在这个函数中，会根据当前事件是读事件还是写事件，分别调用ngx\_http\_request\_t中的read\_event\_handler或者是write\_event\_handler。由于此时，我们的请求头已经读取完成了，之前有说过，nginx的做法是先不读取请求body，所以这里面我们设置read\_event\_handler为ngx\_http\_block\_reading，即不读取数据了。刚才说到，真正开始处理数据，是在ngx\_http\_handler这个函数里面，这个函数会设置write\_event\_handler为ngx\_http\_core\_run\_phases，并执行ngx\_http\_core\_run\_phases函数。ngx\_http\_core\_run\_phases这个函数将执行多阶段请求处理，nginx将一个http请求的处理分为多个阶段，那么这个函数就是执行这些阶段来产生数据。因为ngx\_http\_core\_run\_phases最后会产生数据，所以我们就很容易理解，为什么设置写事件的处理函数为ngx\_http\_core\_run\_phases了。在这里，我简要说明了一下函数的调用逻辑，我们需要明白最终是调用ngx\_http\_core\_run\_phases来处理请求，产生的响应头会放在ngx\_http\_request\_t的headers\_out中，这一部分内容，我会放在请求处理流程里面去讲。nginx的各种阶段会对请求进行处理，最后会调用filter来过滤数据，对数据进行加工，如truncked传输、gzip压缩等。这里的filter包括header filter与body filter，即对响应头或响应体进行处理。filter是一个链表结构，分别有header filter与body filter，先执行header filter中的所有filter，然后再执行body filter中的所有filter。在header filter中的最后一个filter，即ngx\_http\_header\_filter，这个filter将会遍历所有的响应头，最后需要输出的响应头在一个连续的内存，然后调用ngx\_http\_write\_filter进行输出。ngx\_http\_write\_filter是body filter中的最后一个，所以nginx首先的body信息，在经过一系列的body filter之后，最后也会调用ngx\_http\_write\_filter来进行输出(有图来说明)。

这里要注意的是，nginx会将整个请求头都放在一个buffer里面，这个buffer的大小通过配置项client\_header\_buffer\_size来设置，如果用户的请求头太大，这个buffer装不下，那nginx就会重新分配一个新的更大的buffer来装请求头，这个大buffer可以通过large\_client\_header\_buffers来设置，这个large\_buffer这一组buffer，比如配置4 8k，就是表示有四个8k大小的buffer可以用。注意，为了保存请求行或请求头的完整性，一个完整的请求行或请求头，需要放在一个连续的内存里面，所以，一个完整的请求行或请求头，只会保存在一个buffer里面。这样，如果请求行大于一个buffer的大小，就会返回414错误，如果一个请求头大小大于一个buffer大小，就会返回400错误。在了解了这些参数的值，以及nginx实际的做法之后，在应用场景，我们就需要根据实际的需求来调整这些参数，来优化我们的程序了。

处理流程图：



以上这些，就是nginx中一个http请求的生命周期了。我们再看看与请求相关的一些概念吧。

##### keepalive

当然，在nginx中，对于http1.0与http1.1也是支持长连接的。什么是长连接呢？我们知道，http请求是基于TCP协议之上的，那么，当客户端在发起请求前，需要先与服务端建立TCP连接，而每一次的TCP连接是需要三次握手来确定的，如果客户端与服务端之间网络差一点，这三次交互消费的时间会比较多，而且三次交互也会带来网络流量。当然，当连接断开后，也会有四次的交互，当然对用户体验来说就不重要了。而http请求是请求应答式的，如果我们能知道每个请求头与响应体的长度，那么我们是可以在一个连接上面执行多个请求的，这就是所谓的长连接，但前提条件是我们先得确定请求头与响应体的长度。对于请求来说，如果当前请求需要有body，如POST请求，那么nginx就需要客户端在请求头中指定content-length来表明body的大小，否则返回400错误。也就是说，请求体的长度是确定的，那么响应体的长度呢？先来看看http协议中关于响应body长度的确定：

1. 对于http1.0协议来说，如果响应头中有content-length头，则以content-length的长度就可以知道body的长度了，客户端在接收body时，就可以依照这个长度来接收数据，接收完后，就表示这个请求完成了。而如果没有content-length头，则客户端会一直接收数据，直到服务端主动断开连接，才表示body接收完了。
2. 而对于http1.1协议来说，如果响应头中的Transfer-encoding为chunked传输，则表示body是流式输出，body会被分成多个块，每块的开始会标识出当前块的长度，此时，body不需要通过长度来指定。如果是非chunked传输，而且有content-length，则按照content-length来接收数据。否则，如果是非chunked，并且没有content-length，则客户端接收数据，直到服务端主动断开连接。

从上面，我们可以看到，除了http1.0不带content-length以及http1.1非chunked不带content-length外，body的长度是可知的。此时，当服务端在输出完body之后，会可以考虑使用长连接。能否使用长连接，也是有条件限制的。如果客户端的请求头中的connection为close，则表示客户端需要关掉长连接，如果为keep-alive，则客户端需要打开长连接，如果客户端的请求中没有connection这个头，那么根据协议，如果是http1.0，则默认为close，如果是http1.1，则默认为keep-alive。如果结果为keepalive，那么，nginx在输出完响应体后，会设置当前连接的keepalive属性，然后等待客户端下一次请求。当然，nginx不可能一直等待下去，如果客户端一直不发数据过来，岂不是一直占用这个连接？所以当nginx设置了keepalive等待下一次的请求时，同时也会设置一个最大等待时间，这个时间是通过选项keepalive\_timeout来配置的，如果配置为0，则表示关掉keepalive，此时，http版本无论是1.1还是1.0，客户端的connection不管是close还是keepalive，都会强制为close。

如果服务端最后的决定是keepalive打开，那么在响应的http头里面，也会包含有connection头域，其值是”Keep-Alive”，否则就是”Close”。如果connection值为close，那么在nginx响应完数据后，会主动关掉连接。所以，对于请求量比较大的nginx来说，关掉keepalive最后会产生比较多的time-wait状态的socket。一般来说，当客户端的一次访问，需要多次访问同一个server时，打开keepalive的优势非常大，比如图片服务器，通常一个网页会包含很多个图片。打开keepalive也会大量减少time-wait的数量。

##### pipe

在http1.1中，引入了一种新的特性，即pipeline。那么什么是pipeline呢？pipeline其实就是流水线作业，它可以看作为keepalive的一种升华，因为pipeline也是基于长连接的，目的就是利用一个连接做多次请求。如果客户端要提交多个请求，对于keepalive来说，那么第二个请求，必须要等到第一个请求的响应接收完全后，才能发起，这和TCP的停止等待协议是一样的，得到两个响应的时间至少为2\*RTT。而对pipeline来说，客户端不必等到第一个请求处理完后，就可以马上发起第二个请求。得到两个响应的时间可能能够达到1\*RTT。nginx是直接支持pipeline的，但是，nginx对pipeline中的多个请求的处理却不是并行的，依然是一个请求接一个请求的处理，只是在处理第一个请求的时候，客户端就可以发起第二个请求。这样，nginx利用pipeline减少了处理完一个请求后，等待第二个请求的请求头数据的时间。其实nginx的做法很简单，前面说到，nginx在读取数据时，会将读取的数据放到一个buffer里面，所以，如果nginx在处理完前一个请求后，如果发现buffer里面还有数据，就认为剩下的数据是下一个请求的开始，然后就接下来处理下一个请求，否则就设置keepalive。

##### lingering\_close

lingering\_close，字面意思就是延迟关闭，也就是说，当nginx要关闭连接时，并非立即关闭连接，而是先关闭tcp连接的写，再等待一段时间后再关掉连接的读。为什么要这样呢？我们先来看看这样一个场景。nginx在接收客户端的请求时，可能由于客户端或服务端出错了，要立即响应错误信息给客户端，而nginx在响应错误信息后，大分部情况下是需要关闭当前连接。nginx执行完write()系统调用把错误信息发送给客户端，write()系统调用返回成功并不表示数据已经发送到客户端，有可能还在tcp连接的write buffer里。接着如果直接执行close()系统调用关闭tcp连接，内核会首先检查tcp的read buffer里有没有客户端发送过来的数据留在内核态没有被用户态进程读取，如果有则发送给客户端RST报文来关闭tcp连接丢弃write buffer里的数据，如果没有则等待write buffer里的数据发送完毕，然后再经过正常的4次分手报文断开连接。所以,当在某些场景下出现tcp write buffer里的数据在write()系统调用之后到close()系统调用执行之前没有发送完毕，且tcp read buffer里面还有数据没有读，close()系统调用会导致客户端收到RST报文且不会拿到服务端发送过来的错误信息数据。那客户端肯定会想，这服务器好霸道，动不动就reset我的连接，连个错误信息都没有。

在上面这个场景中，我们可以看到，关键点是服务端给客户端发送了RST包，导致自己发送的数据在客户端忽略掉了。所以，解决问题的重点是，让服务端别发RST包。再想想，我们发送RST是因为我们关掉了连接，关掉连接是因为我们不想再处理此连接了，也不会有任何数据产生了。对于全双工的TCP连接来说，我们只需要关掉写就行了，读可以继续进行，我们只需要丢掉读到的任何数据就行了，这样的话，当我们关掉连接后，客户端再发过来的数据，就不会再收到RST了。当然最终我们还是需要关掉这个读端的，所以我们会设置一个超时时间，在这个时间过后，就关掉读，客户端再发送数据来就不管了，作为服务端我会认为，都这么长时间了，发给你的错误信息也应该读到了，再慢就不关我事了，要怪就怪你RP不好了。当然，正常的客户端，在读取到数据后，会关掉连接，此时服务端就会在超时时间内关掉读端。这些正是lingering\_close所做的事情。协议栈提供 SO\_LINGER 这个选项，它的一种配置情况就是来处理lingering\_close的情况的，不过nginx是自己实现的lingering\_close。lingering\_close存在的意义就是来读取剩下的客户端发来的数据，所以nginx会有一个读超时时间，通过lingering\_timeout选项来设置，如果在lingering\_timeout时间内还没有收到数据，则直接关掉连接。nginx还支持设置一个总的读取时间，通过lingering\_time来设置，这个时间也就是nginx在关闭写之后，保留socket的时间，客户端需要在这个时间内发送完所有的数据，否则nginx在这个时间过后，会直接关掉连接。当然，nginx是支持配置是否打开lingering\_close选项的，通过lingering\_close选项来配置。 那么，我们在实际应用中，是否应该打开lingering\_close呢？这个就没有固定的推荐值了，如Maxim Dounin所说，lingering\_close的主要作用是保持更好的客户端兼容性，但是却需要消耗更多的额外资源（比如连接会一直占着）。

### Nginx基本数据结构

### Nginx配置系统

nginx的配置系统由一个主配置文件和其他一些辅助的配置文件构成。这些配置文件均是纯文本文件，全部位于nginx安装目录下的conf目录下。

#### 指令上下文

nginx.conf中的配置信息，根据其逻辑上的意义，对它们进行了分类，也就是分成了多个作用域，或者称之为配置指令上下文。不同的作用域含有一个或者多个配置项。

当前nginx支持的几个指令上下文：

|  |  |
| --- | --- |
| **main:** | nginx在运行时与具体业务功能（比如http服务或者email服务代理）无关的一些参数，比如工作进程数，运行的身份等。 |
| **http:** | 与提供http服务相关的一些配置参数。例如：是否使用keepalive啊，是否使用gzip进行压缩等。 |
| **server:** | http服务上支持若干虚拟主机。每个虚拟主机一个对应的server配置项，配置项里面包含该虚拟主机相关的配置。在提供mail服务的代理时，也可以建立若干server.每个server通过监听的地址来区分。 |
| **location:** | http服务中，某些特定的URL对应的一系列配置项。 |
| **mail:** | 实现email相关的SMTP/IMAP/POP3代理时，共享的一些配置项（因为可能实现多个代理，工作在多个监听地址上）。 |

通常http上下文和mail上下文一定是出现在main上下文里的。在一个上下文里，可能包含另外一种类型的上下文多次。例如：如果http服务，支持了多个虚拟主机，那么在http上下文里，就会出现多个server上下文。

存在于main上下文中的配置指令如下:

* user
* worker\_processes
* error\_log
* events
* http
* mail

存在于http上下文中的指令如下:

* server

存在于mail上下文中的指令如下：

* server
* auth\_http
* imap\_capabilities

存在于server上下文中的配置指令如下：

* listen
* server\_name
* access\_log
* location
* protocol
* proxy
* smtp\_auth
* xclient

存在于location上下文中的指令如下：

* index
* root

当然，这里只是一些示例。具体有哪些配置指令，以及这些配置指令可以出现在什么样的上下文中，需要参考nginx的使用文档。

### 模块化体系结构

nginx的内部结构是由核心部分和一系列的功能模块所组成。这样划分是为了使得每个模块的功能相对简单，便于开发，同时也便于对系统进行功能扩展。为了便于描述，下文中我们将使用nginx core来称呼nginx的核心功能部分。

nginx提供了web服务器的基础功能，同时提供了web服务反向代理，email服务反向代理功能。nginx core实现了底层的通讯协议，为其他模块和nginx进程构建了基本的运行时环境，并且构建了其他各模块的协作基础。

#### 模块概述：

nginx将各功能模块组织成一条链，当有请求到达的时候，请求依次经过这条链上的部分或者全部模块，进行处理。每个模块实现特定的功能。

有两个模块比较特殊，他们居于nginx core和各功能模块的中间。这两个模块就是http模块和mail模块。这2个模块在nginx core之上实现了另外一层抽象，处理与HTTP协议和email相关协议（SMTP/POP3/IMAP）有关的事件，并且确保这些事件能被以正确的顺序调用其他的一些功能模块。

#### 模块分类

|  |  |
| --- | --- |
| **event module:** | 搭建了独立于操作系统的事件处理机制的框架，及提供了各具体事件的处理。包括ngx\_events\_module， ngx\_event\_core\_module和ngx\_epoll\_module等。nginx具体使用何种事件处理模块，这依赖于具体的操作系统和编译选项。 |
| **phase handler:** | 此类型的模块也被直接称为handler模块。主要负责处理客户端请求并产生待响应内容，比如ngx\_http\_static\_module模块，负责客户端的静态页面请求处理并将对应的磁盘文件准备为响应内容输出。 |
| **output filter:** | 也称为filter模块，主要是负责对输出的内容进行处理，可以对输出进行修改。例如，可以实现对输出的所有html页面增加预定义的footbar一类的工作，或者对输出的图片的URL进行替换之类的工作。 |
| **upstream:** | upstream模块实现反向代理的功能，将真正的请求转发到后端服务器上，并从后端服务器上读取响应，发回客户端。upstream模块是一种特殊的handler，只不过响应内容不是真正由自己产生的，而是从后端服务器上读取的。 |
| **load-balancer:** | 负载均衡模块，实现特定的算法，在众多的后端服务器中，选择一个服务器出来作为某个请求的转发服务器。 |

### 请求处理流程

nginx使用一个多进程模型来对外提供服务，其中一个master进程，多个worker进程。master进程负责管理nginx本身和其他worker进程。

所有实际上的业务处理逻辑都在worker进程。worker进程中有一个函数，执行无限循环，不断处理收到的来自客户端的请求，并进行处理，直到整个nginx服务被停止。

worker进程中，ngx\_worker\_process\_cycle()函数就是这个无限循环的处理函数。在这个函数中，一个请求的简单处理流程如下：

1. 操作系统提供的机制（例如epoll, kqueue等）产生相关的事件。
2. 接收和处理这些事件，如是接受到数据，则产生更高层的request对象。
3. 处理request的header和body。
4. 产生响应，并发送回客户端。
5. 完成request的处理。
6. 重新初始化定时器及其他事件。

#### HTTP Reques处理阶段

从nginx的内部来看，一个HTTP Request的处理过程涉及到以下几个阶段。

1. 初始化HTTP Request（读取来自客户端的数据，生成HTTP Request对象，该对象含有该请求所有的信息）。
2. 处理请求头。
3. 处理请求体。
4. 如果有的话，调用与此请求（URL或者Location）关联的handler。
5. 依次调用各phase handler进行处理。

#### phash handler阶段

在这里，我们需要了解一下phase handler这个概念。phase字面的意思，就是阶段。所以phase handlers也就好理解了，就是包含若干个处理阶段的一些handler。

在每一个阶段，包含有若干个handler，再处理到某个阶段的时候，依次调用该阶段的handler对HTTP Request进行处理。

通常情况下，一个phase handler对这个request进行处理，并产生一些输出。通常phase handler是与定义在配置文件中的某个location相关联的。

一个phase handler通常执行以下几项任务：

1. 获取location配置。
2. 产生适当的响应。
3. 发送response header。
4. 发送response body。

当nginx读取到一个HTTP Request的header的时候，nginx首先查找与这个请求关联的虚拟主机的配置。如果找到了这个虚拟主机的配置，那么通常情况下，这个HTTP Request将会经过以下几个阶段的处理（phase handlers）：

|  |  |
| --- | --- |
| **NGX\_HTTP\_POST\_READ\_PHASE:** | |
|  | 读取请求内容阶段 |
| **NGX\_HTTP\_SERVER\_REWRITE\_PHASE:** | |
|  | Server请求地址重写阶段 |
| **NGX\_HTTP\_FIND\_CONFIG\_PHASE:** | |
|  | 配置查找阶段: |
| **NGX\_HTTP\_REWRITE\_PHASE:** | |
|  | Location请求地址重写阶段 |
| **NGX\_HTTP\_POST\_REWRITE\_PHASE:** | |
|  | 请求地址重写提交阶段 |
| **NGX\_HTTP\_PREACCESS\_PHASE:** | |
|  | 访问权限检查准备阶段 |
| **NGX\_HTTP\_ACCESS\_PHASE:** | |
|  | 访问权限检查阶段 |
| **NGX\_HTTP\_POST\_ACCESS\_PHASE:** | |
|  | 访问权限检查提交阶段 |
| **NGX\_HTTP\_TRY\_FILES\_PHASE:** | |
|  | 配置项try\_files处理阶段 |
| **NGX\_HTTP\_CONTENT\_PHASE:** | |
|  | 内容产生阶段 |
| **NGX\_HTTP\_LOG\_PHASE:** | |
|  | 日志模块处理阶段 |

#### content handler 过程

在内容产生阶段，为了给一个request产生正确的响应，nginx必须把这个request交给一个合适的content handler去处理。如果这个request对应的location在配置文件中被明确指定了一个content handler，那么nginx就可以通过对location的匹配，直接找到这个对应的handler，并把这个request交给这个content handler去处理。这样的配置指令包括像，perl，flv，proxy\_pass，mp4等。

如果一个request对应的location并没有直接有配置的content handler，那么nginx依次尝试:

1. 如果一个location里面有配置 random\_index on，那么随机选择一个文件，发送给客户端。
2. 如果一个location里面有配置 index指令，那么发送index指令指明的文件，给客户端。
3. 如果一个location里面有配置 autoindex on，那么就发送请求地址对应的服务端路径下的文件列表给客户端。
4. 如果这个request对应的location上有设置gzip\_static on，那么就查找是否有对应的.gz文件存在，有的话，就发送这个给客户端（客户端支持gzip的情况下）。
5. 请求的URI如果对应一个静态文件，static module就发送静态文件的内容到客户端。

#### filter处理阶段

内容产生阶段完成以后，生成的输出会被传递到filter模块去进行处理。filter模块也是与location相关的。所有的fiter模块都被组织成一条链。输出会依次穿越所有的filter，直到有一个filter模块的返回值表明已经处理完成。

这里列举几个常见的filter模块，例如：

1. server-side includes。
2. XSLT filtering。
3. 图像缩放之类的。
4. gzip压缩。

在所有的filter中，有几个filter模块需要关注一下。按照调用的顺序依次说明如下：

|  |  |
| --- | --- |
| **write:** | 写输出到客户端，实际上是写到连接对应的socket上。 |
| **postpone:** | 这个filter是负责subrequest的，也就是子请求的。 |
| **copy:** | 将一些需要复制的buf(文件或者内存)重新复制一份然后交给剩余的body filter处理。 |

#### 不同类型服务器下的请求流程

##### 基于域名的虚拟服务器

server {

listen 80;

server\_name example.org www.example.org;

...

}

server {

listen 80;

server\_name example.net www.example.net;

...

}

server {

listen 80;

server\_name example.com www.example.com;

...

}

以上配置nginx会根据请求头中的“Host”来决定请求将转发到那个服务器。如果它的值没有匹配，或者不包含头字段，则nginx会将请求转发到这个端口上的默认服务器。Nginx标准情况下把第一个server当成默认服务器。可以使用下面配置进行明确指定。

server {

listen 80 **default\_server**;

server\_name example.net www.example.net;

...

}

注意：这个default\_server是监听端口的属性而不是一个服务器名称。

##### 怎么防止未定义服务器名的服务器处理请求

如果没有Host头字段的请求不被允许，服务器可以放弃这个请求，如下定义：

server {

listen 80;

server\_name "";

return 444;

}

Server\_name为空能匹配不存在”Host”头字段的请求。

##### 混合的基于域名和基于ip的虚拟服务器

server {

listen 192.168.1.1:80;

server\_name example.org www.example.org;

...

}

server {

listen 192.168.1.1:80;

server\_name example.net www.example.net;

...

}

server {

listen 192.168.1.2:80;

server\_name example.com www.example.com;

...

}

以上配置中，nginx首先测试请求中的IP地址和端口和server模块监听指令。然后它测试请求中的头字段和server中匹配ip地址和端口的模块的server\_name实体。如果server\_name没找到，requests会被默认服务器处理。

注意：由于default\_server是监听端口的属性，则不同端口能定义不同默认服务器。如下：

server {

listen 192.168.1.1:80;

server\_name example.org www.example.org;

...

}

server {

listen 192.168.1.1:80 **default\_server**;

server\_name example.net www.example.net;

...

}

server {

listen 192.168.1.2:80 **default\_server**;

server\_name example.com www.example.com;

...

}

##### 一个简单的PHP站点配置

server {

listen 80;

server\_name example.org www.example.org;

root /data/www;

location / {

index index.html index.php;

}

location ~\* \.(gif|jpg|png)$ {

expires 30d;

}

location ~ \.php$ {

fastcgi\_pass localhost:9000;

fastcgi\_param SCRIPT\_FILENAME

$document\_root$fastcgi\_script\_name;

include fastcgi\_params;

}

}

Nginx首先不管它列出的顺序，来通过文字字符串搜索最具体的前缀地址。以上配置中唯一的前缀是“/”，因此它匹配任何请求并作为最后的一种处理手段。然后nginx检查在配置文件中列出的正则表达式所给的地址，第一个匹配的会停止搜索，nginx将使用这个地址。如果没有正则表达式匹配一个请求，那么nginx会使用之前发现的最具体的前缀地址。

注意：所有类型的地址只测试不带参数的请求行中的URI部分。这是因为在查询字符中的参数会以好几种方式给出。例如：

/index.php?user=john&page=1

/index.php?page=1&user=john

此外，任何人能再查询字符中请求任何东西。

/index.php?page=1&something+else&user=john

先分析一下以上配置信息：

一个“/logo.gif”请求首先被前缀地址“/”匹配，然后被正则表达式“\.(gif|jpg|png)$”，因此它将被后面地址所处理。使用指令“root /data/www”这个请求被映射到文件/data/www/logo.gif，然后这文件会被发送到客户端。

一个”/index.php”请求也首先匹配前缀地址”/”然后被正则表达式“\.(php)$”。因此，它将被后面地址处理然后这个请求将被分发到在localhost:9000监听的FastCGIf服务器。fastcgi\_param指令将FastCGI的SCRIPT\_FILENAME参数设置为“/data/www/index.php”。$document\_root变量等于root指令，$fastcgi\_script\_name等于request的URI，这里为“index.php”。

一个“/about.html”请求仅仅被前缀地址“/”匹配。因此，它将被该地址处理。使用指令“root /data/www”，请求被映射到文件/data/www/about.html，然后这个文件被发送会客户端。

处理请求“/”会更复杂。它仅被前缀地址“/”匹配。因此它被该地址处理。然后index指令根据它的参数和“root /data/www”指令测试index文件是否存在。如果/data/www/index.html文件不存在，但是/data/www/index.php存在。那么指令会重定向到“/index.php”。然后nginx再一次查询地址，如果请求被客户端发送过来。由我们之前所见，重定向请求最终会被FastCGI服务器处理。

## 配置文件及参数解释

### 核心功能

http://nginx.org/en/docs/ngx\_core\_module.html

#### 目录

[accept\_mutex](http://nginx.org/en/docs/ngx_core_module.html#accept_mutex)  
     [accept\_mutex\_delay](http://nginx.org/en/docs/ngx_core_module.html#accept_mutex_delay)  
     [daemon](http://nginx.org/en/docs/ngx_core_module.html#daemon)  
     [debug\_connection](http://nginx.org/en/docs/ngx_core_module.html#debug_connection)  
     [debug\_points](http://nginx.org/en/docs/ngx_core_module.html#debug_points)  
     [error\_log](http://nginx.org/en/docs/ngx_core_module.html#error_log)  
     [env](http://nginx.org/en/docs/ngx_core_module.html#env)  
     [events](http://nginx.org/en/docs/ngx_core_module.html#events)  
     [include](http://nginx.org/en/docs/ngx_core_module.html#include)  
     [load\_module](http://nginx.org/en/docs/ngx_core_module.html#load_module)  
     [lock\_file](http://nginx.org/en/docs/ngx_core_module.html#lock_file)  
     [master\_process](http://nginx.org/en/docs/ngx_core_module.html#master_process)  
     [multi\_accept](http://nginx.org/en/docs/ngx_core_module.html#multi_accept)  
     [pcre\_jit](http://nginx.org/en/docs/ngx_core_module.html#pcre_jit)  
     [pid](http://nginx.org/en/docs/ngx_core_module.html#pid)  
     [ssl\_engine](http://nginx.org/en/docs/ngx_core_module.html#ssl_engine)  
     [thread\_pool](http://nginx.org/en/docs/ngx_core_module.html#thread_pool)  
     [timer\_resolution](http://nginx.org/en/docs/ngx_core_module.html#timer_resolution)  
     [use](http://nginx.org/en/docs/ngx_core_module.html#use)  
     [user](http://nginx.org/en/docs/ngx_core_module.html#user)  
     [worker\_aio\_requests](http://nginx.org/en/docs/ngx_core_module.html#worker_aio_requests)  
     [worker\_connections](http://nginx.org/en/docs/ngx_core_module.html#worker_connections)  
     [worker\_cpu\_affinity](http://nginx.org/en/docs/ngx_core_module.html#worker_cpu_affinity)  
     [worker\_priority](http://nginx.org/en/docs/ngx_core_module.html#worker_priority)  
     [worker\_processes](http://nginx.org/en/docs/ngx_core_module.html#worker_processes)  
     [worker\_rlimit\_core](http://nginx.org/en/docs/ngx_core_module.html#worker_rlimit_core)  
     [worker\_rlimit\_nofile](http://nginx.org/en/docs/ngx_core_module.html#worker_rlimit_nofile)  
     [working\_directory](http://nginx.org/en/docs/ngx_core_module.html#working_directory)

#### 指令

|  |  |
| --- | --- |
| Syntax: | **accept\_mutex** on | off; |
| Default: | accept\_mutex on; |
| Context: | events |

如果accept\_mutex 启用，工作进程会轮流接受新请求连接。否则，所有的工作进程都会收到新连接，当新连接的数量较低时候，一些工作进程会浪费系统资源。

|  |  |
| --- | --- |
| Syntax: | **accept\_mutex\_delay** *time*; |
| Default: | accept\_mutex\_delay 500ms; |
| Context: | events |

如果accept\_mutex启用，指定一个最大时间，即如果其它工作进程正在接收新连接，某个个工作进程试图重新开始去接收新请求的时间。

|  |  |
| --- | --- |
| Syntax: | **daemon** on | off; |
| Default: | daemon on; |
| Context: | main |

决定nginx是否为守护进程，主要在开发过程中使用

|  |  |
| --- | --- |
| Syntax: | **debug\_connection** *address* | *CIDR* | unix:; |
| Default: | — |
| Context: | events |

为指定客户端连接启用debugging日志。其它连接将使用在[error\_log](http://nginx.org/en/docs/ngx_core_module.html#error_log) 指令中确立的logging等级。Debugged连接由IPv4或者IPv6地址或网络指定。一个连接也能被主机名指定。对于使用UNIX-domain 套接字，添加“unix:”参数。

events {

debug\_connection 127.0.0.1;

debug\_connection localhost;

debug\_connection 192.0.2.0/24;

debug\_connection ::1;

debug\_connection 2001:0db8::/32;

debug\_connection unix:;

...

}

For this directive to work, nginx needs to be built with --with-debug, see “[A debugging log](http://nginx.org/en/docs/debugging_log.html)”.

|  |  |
| --- | --- |
| Syntax: | **debug\_points** abort | stop; |
| Default: | — |
| Context: | main |

这个指令用来debug。

当内部错误出现，例如重启工作进程上的sockets泄漏，启用debug\_points 能为使用系统的debugger来更好分析而导出一个主要文件或者关闭某个进程。

|  |  |
| --- | --- |
| Syntax: | **error\_log** *file* | stderr | syslog:server=*address*[,*parameter*=*value*] | memory:*size* [debug | info | notice |warn | error | crit | alert | emerg]; |
| Default: | error\_log logs/error.log error; |
| Context: | main, http, mail, stream, server, location |

配置loging信息。在统一等级上能指定多个logs。

第一个参数指定保存log信息的文件。stderr为标准错误文件。使用”syslog:”前缀能配置[syslog](http://nginx.org/en/docs/syslog.html)。”memory:”和”buffer size”指定[cyclic memory buffer](http://nginx.org/en/docs/debugging_log.html#memory) ，它主要用于debug。

第二个参数决定logging的等级。Log等级按照严重性排列。设置某个certain的log等级会使得指定或者更多服务器所有日志信息被记录。例如，默认等级的error可能会产生error，crit，alert和emerg信息。如果不写这个参数，默认是error。

For debug logging to work, nginx needs to be built with --with-debug, see “[A debugging log](http://nginx.org/en/docs/debugging_log.html)”.

|  |  |
| --- | --- |
| Syntax: | **env** *variable*[=*value*]; |
| Default: | env TZ; |
| Context: | main |

默认情况下，nginx会移除继承于它的父程序所有的环境变量，除了TZ变量。这个指令允许保留一些继承下来的变量，更改它们的值，或者产生新的环境变量。这些变量如下：

在一个可执行文件的[live upgrade](http://nginx.org/en/docs/control.html#upgrade)过程中继承下来的；

在[ngx\_http\_perl\_module](http://nginx.org/en/docs/http/ngx_http_perl_module.html) 模块中使用的；

被工作进程使用的。需要记住的事，通过这种方式控制系统库不是一直都可以的，因为对于系统库在初始化阶段在它们通过这个指令来设置之前去检查变量 是很普遍的。这个错误在之前被提起过。[live upgrade](http://nginx.org/en/docs/control.html#upgrade)

TZ变量对于[ngx\_http\_perl\_module](http://nginx.org/en/docs/http/ngx_http_perl_module.html)模块是一直可继承和可用的，除非他被明确指定：

env MALLOC\_OPTIONS;

env PERL5LIB=/data/site/modules;

env OPENSSL\_ALLOW\_PROXY\_CERTS=1;

Nginx环境变量只能在nginx内部使用，不能由用户设置。

|  |  |
| --- | --- |
| Syntax: | **events** { ... } |
| Default: | — |
| Context: | main |

提供了配置文件上下文，在这里可以指定指令去影响连接进程。

|  |  |
| --- | --- |
| Syntax: | **include** *file* | *mask*; |
| Default: | — |
| Context: | any |

包含别的文件或多个文件去匹配特殊的mask到配置文件中个。包换的文件需要语法正确。

Usage example:

include mime.types;

include vhosts/\*.conf;

|  |  |
| --- | --- |
| Syntax: | **load\_module** *file*; |
| Default: | — |
| Context: | main |

This directive appeared in version 1.9.11.

加载动态模块。

Example:

load\_module modules/ngx\_mail\_module.so;

|  |  |
| --- | --- |
| Syntax: | **lock\_file** *file*; |
| Default: | lock\_file logs/nginx.lock; |
| Context: | main |

Nginx使用锁定机制来实现[accept\_mutex](http://nginx.org/en/docs/ngx_core_module.html#accept_mutex) 和对共享内存的连续访问。在大多数系统中，locks使用原子操作来实现，它的指令被无视。在另外的系统中，“lock file”机制被使用，这个指令指定lock files的前缀名称。

|  |  |
| --- | --- |
| Syntax: | **master\_process** on | off; |
| Default: | master\_process on; |
| Context: | main |

决定是否工作精彩要启动。这个指令为开发者准备。

|  |  |
| --- | --- |
| Syntax: | **multi\_accept** on | off; |
| Default: | multi\_accept off; |
| Context: | events |

如果multi\_accept不可用，那么工作进程会每次接收一个新连接。反之，工作进程会接受所有的新连接。

如果[kqueue](http://nginx.org/en/docs/events.html#kqueue) 连接处理办法的办法被使用，那么这个指令会被忽略，因为它报告需要被接收的新请求个数。

|  |  |
| --- | --- |
| Syntax: | **pcre\_jit** on | off; |
| Default: | pcre\_jit off; |
| Context: | main |

这个指令在版本1.1.12出现

启用或者关闭对“just-in-time compilation”的使用，这用在解析配置过程中正则表达式。PCREJIT能显著加速处理正则表达式。

JIT在由-enable-jit配置参数建立的从版本8.20开始的PCRE库中是可用的。当PCRE库由nginx建立时，通过-with-pcre-jit配置参数能启用对JIT的支持。

|  |  |
| --- | --- |
| Syntax: | **pid** *file*; |
| Default: | pid nginx.pid; |
| Context: | main |

定义存储主进程id的文件

|  |  |
| --- | --- |
| Syntax: | **ssl\_engine** *device*; |
| Default: | — |
| Context: | main |

定义SSL加速器的名字

|  |  |
| --- | --- |
| Syntax: | **thread\_pool** *name* threads=*number* [max\_queue=*number*]; |
| Default: | thread\_pool default threads=32 max\_queue=65536; |
| Context: | main |

这个指令在版本1.7.11出现

为被用作多线程读取发送文件的不阻塞工作进程的线程池命名。

Threads的参数定义来在线程池中线程的数量。

在事件中，线程池中所有的线程都是忙碌的，一个新的任务会在队列中等待。max\_queue参数限制允许等待在队列中的任务的数目。默认情况下，最多由65536个任务可以在队列中等待，当队列超负荷，这个任务会报错。

|  |  |
| --- | --- |
| Syntax: | **timer\_resolution** *interval*; |
| Default: | — |
| Context: | main |

减少工作进程计时器的分辨率，从而减少gettimeofday()的系统调用。默认情况下，gettimeofday()在每一次内核事件被接受。通过降低分辨率，gettimeofday()只有在一个指定时间间隔内才被调用一次。

Example:

timer\_resolution 100ms;

内部实现依靠这些办法：

* the EVFILT\_TIMER filter if kqueue is used;
* timer\_create() if eventport is used;
* setitimer() otherwise.

|  |  |
| --- | --- |
| Syntax: | **use** *method*; |
| Default: | — |
| Context: | events |

指定使用的[connection processing](http://nginx.org/en/docs/events.html)办法。正常情况下无需确切指定，因为nginx会默认使用最高效的办法。

|  |  |
| --- | --- |
| Syntax: | **user** *user* [*group*]; |
| Default: | user nobody nobody; |
| Context: | main |

定义被工作进程使用的用户和组凭据。如果组被忽略，那么组名用的和user名一样。

|  |  |
| --- | --- |
| Syntax: | **worker\_aio\_requests** *number*; |
| Default: | worker\_aio\_requests 32; |
| Context: | events |

This directive appeared in versions 1.1.4 and 1.0.7.

当使用通过[epoll](http://nginx.org/en/docs/events.html#epoll)连接方法的[aio](http://nginx.org/en/docs/http/ngx_http_core_module.html#aio)，为单一的工作进程设置优秀的一部I/O操作。

|  |  |
| --- | --- |
| Syntax: | **worker\_connections** *number*; |
| Default: | worker\_connections 512; |
| Context: | events |

设置一个工作进程能同时打开的最大连接数。

需要记住的是所有的连接（包括代理服务器，和其它）不仅仅是来自客户端的连接。另一个需要考虑的是真实的同时连接数不能超过当前限制的最多文件打开数。这个可由 [worker\_rlimit\_nofile](http://nginx.org/en/docs/ngx_core_module.html#worker_rlimit_nofile)更改。

|  |  |
| --- | --- |
| Syntax: | **worker\_cpu\_affinity** *cpumask* ...; **worker\_cpu\_affinity** auto [*cpumask*]; |
| Default: | — |
| Context: | main |

把工作进程绑定到cup集合中。每个cpu集合由一个所允许的cpu位掩码表示。这里应该是一个给每个工作进程定义的分开的集合。默认情况下，工作进程不会绑定到任何特定cup。

For example,

worker\_processes 4;

worker\_cpu\_affinity 0001 0010 0100 1000;

binds each worker process to a separate CPU, while

worker\_processes 2;

worker\_cpu\_affinity 0101 1010;

将以第一个进程绑定到CPU0/CPU2.第二个工作进程绑定到CPU1/CPU3.第二个例子对于超线程更适合。

auto这个值（1.9.10）允许自动绑定工作进程到可用的cpu：

worker\_processes auto;

worker\_cpu\_affinity auto;

可选的mask参数被用来限制cpu可用性（动态绑定）：

worker\_cpu\_affinity auto 01010101;

|  |  |
| --- | --- |
| Syntax: | **worker\_priority** *number*; |
| Default: | worker\_priority 0; |
| Context: | main |

定义工作进程的调度优先级。一个负数意味着更高的优先级，允许的返回在-20到20之间。

Example:

worker\_priority -10;

|  |  |
| --- | --- |
| Syntax: | **worker\_processes** *number* | auto; |
| Default: | worker\_processes 1; |
| Context: | main |

定义工作进程的数量。

最佳的值取决于很多因素，包括CPU核心数量，存储数据的硬盘驱动器和负载模式的数量。当你还有疑问的时候，把它设置为cup内核的数量将是一个很好的开端，这个值会自动检测使用它。

The auto parameter is supported starting from versions 1.3.8 and 1.2.5.

|  |  |
| --- | --- |
| Syntax: | **worker\_rlimit\_core** *size*; |
| Default: | — |
| Context: | main |

为工作进程更改核心文件的大小限制。不用重启主进程就能使用它。

|  |  |
| --- | --- |
| Syntax: | **worker\_rlimit\_nofile** *number*; |
| Default: | — |
| Context: | main |

为工作进程更改最大打开文件数量的限制。不用重启主进程就能使用它。

|  |  |
| --- | --- |
| Syntax: | **working\_directory** *directory*; |
| Default: | — |
| Context: | main |

为工作经常定义目前的工作目录。主要用在书写核心文件，也就是说工作进程需要有这个特殊目录写权限。