# 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

#### 使用源文件编译安装

参见：<https://www.nginx.com/resources/wiki/start/topics/tutorials/install/>

1. 下载源文件

http://nginx.org/download/nginx-1.8.0.tar.gz?\_ga=1.224427226.1175265148.1456706625

1. 解压进入文件目录
2. 输入下列指令
3. ./configure

make

sudo make install

1. 安装过程中需要的一些库：

PCRE包：apt-get install libpcre3 libpcre3-dev

gzip包：apt-get install zlib1g-dev

### 命令管理

**命令管理主要有两种方式：1.nginx命令发送信号2.kill指令给工作进程发送信号**

参考地址：<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. 配置测试过程中抑制非错误消息。 |
| -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 |
| -g directives | 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安全关闭工作进程 |

#### 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信号给新的主进程来强制它退出。
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

**root和index中的内容放在location外，它们能继承到各个location中。**

参考地址：<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

坏的配置：

**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

坏的配置：

**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是个大坑，要用它之前确保你测试通过。平时尽量使用别的办法代替它**

参考地址：<https://www.nginx.com/resources/wiki/start/topics/depth/ifisevil/>

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-rewrite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

**命名方式：确切名称，通配符，正则表达式和混合的办法，请求转发优先级顺序**

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）

#### 服务器名称

参考地址：<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）

**可使用空名，具体ip，default\_server 标签设置为默认，没匹配的请求最终发向默认服务器。**

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

参考地址：<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

怎么记录各个服务器的连接数？

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

**用于检查反响代理中工作服务器的健康程度，和记录恢复过程**

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 决定了服务器多少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/)

需要理解ssl的机制？

#### 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”.

#### https延伸

HTTP协议和安全协议同属于应用层（[OSI模型](https://zh.wikipedia.org/wiki/OSI%E6%A8%A1%E5%9E%8B)的最高层），具体来讲，安全协议工作在HTTP之下，运输层之上：安全协议向运行HTTP的进程提供一个类似于TCP的套接字，供进程向其中注入报文，安全协议将报文加密并注入运输层套接字；或是从运输层获取加密报文，解密后交给对应的进程。严格地讲，HTTPS并不是一个单独的协议，而是对工作在一加密连接（[TLS](https://zh.wikipedia.org/wiki/%E4%BC%A0%E8%BE%93%E5%B1%82%E5%AE%89%E5%85%A8)或SSL）上的常规HTTP协议的称呼。

##### 定义

**超文本传输安全协议**（英语：**Hypertext Transfer Protocol Secure**，[缩写](https://zh.wikipedia.org/wiki/%E7%B8%AE%E5%AF%AB)：**HTTPS**，也被称为**HTTP over TLS**，**HTTP over SSL**或**HTTP Secure**）是一种[网络安全](https://zh.wikipedia.org/wiki/%E7%B6%B2%E7%B5%A1%E5%AE%89%E5%85%A8)传输协议。在[计算机网络](https://zh.wikipedia.org/wiki/%E8%A8%88%E7%AE%97%E6%A9%9F%E7%B6%B2%E7%B5%A1)上，HTTPS经由[超文本传输协议](https://zh.wikipedia.org/wiki/%E8%B6%85%E6%96%87%E6%9C%AC%E4%BC%A0%E8%BE%93%E5%8D%8F%E8%AE%AE)进行通讯，但利用[SSL/TLS](https://zh.wikipedia.org/wiki/%E4%BC%A0%E8%BE%93%E5%B1%82%E5%AE%89%E5%85%A8)来对数据包进行加密。HTTPS开发的主要目的，是提供对[网络](https://zh.wikipedia.org/wiki/%E7%B6%B2%E8%B7%AF)服务器的[身份认证](https://zh.wikipedia.org/wiki/%E8%BA%AB%E4%BB%BD%E9%AA%8C%E8%AF%81)，保护交换数据的隐私与[完整性](https://zh.wikipedia.org/wiki/%E5%AE%8C%E6%95%B4%E6%80%A7)。

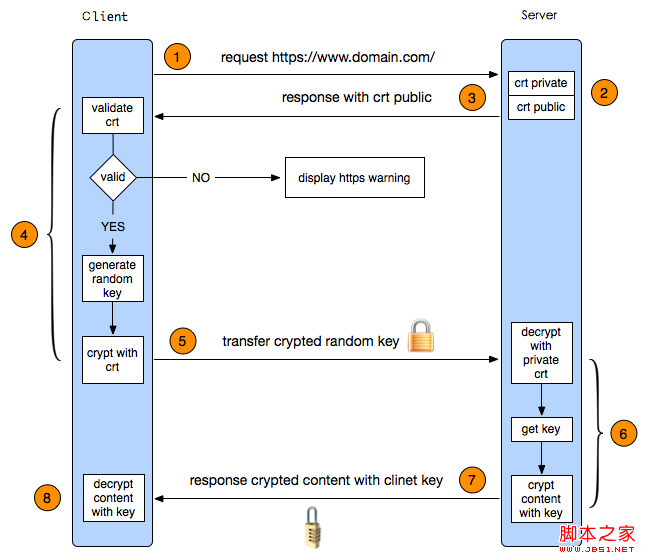
##### 主要思想：

HTTPS的主要思想是在不安全的网络上创建一安全信道，并可在使用适当的加密包和*服务器证书可被验证且可被信任时*，对[窃听](https://zh.wikipedia.org/wiki/%E7%AB%8A%E8%81%BD)和[中间人攻击](https://zh.wikipedia.org/wiki/%E4%B8%AD%E9%97%B4%E4%BA%BA%E6%94%BB%E5%87%BB)提供合理的防护。

HTTPS的信任继承基于预先安装在浏览器中的[证书颁发机构](https://zh.wikipedia.org/wiki/%E8%AF%81%E4%B9%A6%E9%A2%81%E5%8F%91%E6%9C%BA%E6%9E%84)（如VeriSign、Microsoft等）（意即“我信任证书颁发机构告诉我应该信任的”）。因此，一个到某网站的HTTPS连接可被信任，[当且仅当](https://zh.wikipedia.org/wiki/%E5%BD%93%E4%B8%94%E4%BB%85%E5%BD%93)：

1. 用户相信他们的浏览器正确实现了HTTPS且安装了正确的证书颁发机构；
2. 用户相信证书颁发机构仅信任合法的网站；
3. 被访问的网站提供了一个有效的证书，意即，它是由一个被信任的证书颁发机构签发的（大部分浏览器会对无效的证书发出警告）；
4. 该证书正确地验证了被访问的网站（如，访问https://example时收到了给“Example Inc.”而不是其它组织的证书）；
5. 或者互联网上相关的节点是值得信任的，或者用户相信本协议的加密层（[TLS](https://zh.wikipedia.org/wiki/%E4%BC%A0%E8%BE%93%E5%B1%82%E5%AE%89%E5%85%A8)或SSL）不能被窃听者破坏。

##### 请求流程



**1. 客户端发起HTTPS请求**

　　这个没什么好说的，就是用户在浏览器里输入一个https网址，然后连接到server的443端口。

**2. 服务端的配置**

　　采用HTTPS协议的服务器必须要有一套数字证书，可以自己制作，也可以向组织申请。区别就是自己颁发的证书需要客户端验证通过，才可以继续访问，而使用受信任的公司申请的证书则不会弹出提示页面(startssl就是个不错的选择，有1年的免费服务)。这套证书其实就是一对公钥和私钥。如果对公钥和私钥不太理解，可以想象成一把钥匙和一个锁头，只是全世界只有你一个人有这把钥匙，你可以把锁头给别人，别人可以用这个锁把重要的东西锁起来，然后发给你，因为只有你一个人有这把钥匙，所以只有你才能看到被这把锁锁起来的东西。

**3. 传送证书**

　　这个证书其实就是公钥，只是包含了很多信息，如证书的颁发机构，过期时间等等。

**4. 客户端解析证书**

　　这部分工作是有客户端的TLS来完成的，首先会验证公钥是否有效，比如颁发机构，过期时间等等，如果发现异常，则会弹出一个警告框，提示证书存在问题。如果证书没有问题，那么就生成一个随机值。然后用证书对该随机值进行加密。就好像上面说的，把随机值用锁头锁起来，这样除非有钥匙，不然看不到被锁住的内容。

**5. 传送加密信息**

　　这部分传送的是用证书加密后的随机值，目的就是让服务端得到这个随机值，以后客户端和服务端的通信就可以通过这个随机值来进行加密解密了。

**6. 服务段解密信息**

　　服务端用私钥解密后，得到了客户端传过来的随机值(私钥)，然后把内容通过该值进行对称加密。所谓对称加密就是，将信息和私钥通过某种算法混合在一起，这样除非知道私钥，不然无法获取内容，而正好客户端和服务端都知道这个私钥，所以只要加密算法够彪悍，私钥够复杂，数据就够安全。

**7. 传输加密后的信息**

　　这部分信息是服务段用私钥加密后的信息，可以在客户端被还原。

**8. 客户端解密信息**

客户端用之前生成的私钥解密服务段传过来的信息，于是获取了解密后的内容。整个过程第三方即使监听到了数据，也束手无策。

##### TLS协议

TLS协议的优势在于它是与[应用层](https://zh.wikipedia.org/wiki/%E5%BA%94%E7%94%A8%E5%B1%82)协议独立无关的。高层的应用层协议（例如：[HTTP](https://zh.wikipedia.org/wiki/%E8%B6%85%E6%96%87%E6%9C%AC%E4%BC%A0%E8%BE%93%E5%8D%8F%E8%AE%AE)、[FTP](https://zh.wikipedia.org/wiki/%E6%96%87%E4%BB%B6%E4%BC%A0%E8%BE%93%E5%8D%8F%E8%AE%AE)、[Telnet](https://zh.wikipedia.org/wiki/Telnet)等等）能透明的创建于TLS协议之上。TLS协议在应用层协议通信之前就已经完成加密算法、通信密钥的协商以及服务器认证工作。在此之后应用层协议所传送的数据都会被加密，从而保证通信的私密性。

**传输层安全协议**（英语：Transport Layer Security，[缩写](https://zh.wikipedia.org/wiki/%E7%B8%AE%E5%AF%AB)：**TLS**），及其前身**安全套接层**（Secure Sockets Layer，[缩写](https://zh.wikipedia.org/wiki/%E7%B8%AE%E5%AF%AB)：**SSL**）是一种[安全协议](https://zh.wikipedia.org/wiki/%E5%AE%89%E5%85%A8%E5%8D%8F%E8%AE%AE)，目的是为[互联网](https://zh.wikipedia.org/wiki/%E7%B6%B2%E9%9A%9B%E7%B6%B2%E8%B7%AF)通信，提供安全及数据[完整性](https://zh.wikipedia.org/wiki/%E5%AE%8C%E6%95%B4%E6%80%A7)保障。

* 当客户端连接到支持TLS协议的服务器要求创建安全连接并列出了受支持的密码组合（加密密码算法和加密哈希函数），握手开始。
* 服务器从该列表中决定加密和散列函数，并通知客户端。
* 服务器发回其数字证书，此证书通常包含服务器的名称、受信任的证书颁发机构（CA）和服务器的公钥。
* 客户端确认其颁发的证书的有效性。
* 为了生成会话密钥用于安全连接，客户端使用服务器的公钥加密随机生成的密钥，并将其发送到服务器，只有服务器才能使用自己的私钥解密。
* 利用随机数，双方生成用于加密和解密的对称密钥。这就是TLS协议的握手，握手完毕后的连接是安全的，直到连接（被）关闭。如果上述任何一个步骤失败，TLS握手过程就会失败，并且断开所有的连接。

##### 其它

###### 与HTTP的差异

与[HTTP](https://zh.wikipedia.org/wiki/HTTP)的[URL](https://zh.wikipedia.org/wiki/%E7%BB%9F%E4%B8%80%E8%B5%84%E6%BA%90%E5%AE%9A%E4%BD%8D%E7%AC%A6)由“http://”起始且默认使用[端口](https://zh.wikipedia.org/wiki/TCP/UDP%E7%AB%AF%E5%8F%A3%E5%88%97%E8%A1%A8)80不同，HTTPS的URL由“https://”起始且默认使用[端口](https://zh.wikipedia.org/wiki/TCP/UDP%E7%AB%AF%E5%8F%A3%E5%88%97%E8%A1%A8)443。

HTTP是不安全的，且攻击者通过[监听](https://zh.wikipedia.org/wiki/%E7%9B%A3%E8%81%BD)和[中间人攻击](https://zh.wikipedia.org/wiki/%E4%B8%AD%E9%97%B4%E4%BA%BA%E6%94%BB%E5%87%BB)等手段，可以获取网站帐户和敏感信息等。HTTPS被设计为可防止前述攻击，并（在没有使用旧版本的SSL时）被认为是安全的。

###### 服务器设置

要使一网络服务器准备好接受HTTPS连接，管理员必须创建一[数字证书](https://zh.wikipedia.org/wiki/%E6%95%B0%E5%AD%97%E8%AF%81%E4%B9%A6)，并交由[证书颁发机构](https://zh.wikipedia.org/wiki/%E8%AF%81%E4%B9%A6%E9%A2%81%E5%8F%91%E6%9C%BA%E6%9E%84)签名以使浏览器接受。证书颁发机构会验证数字证书持有人和其声明的为同一人。浏览器通常都预装了证书颁发机构的证书，所以他们可以验证该签名。

###### 作为访问控制

HTTPS也可被用作客户端[认证](https://zh.wikipedia.org/wiki/%E8%AE%A4%E8%AF%81)手段来将一些信息限制给合法的用户。要做到这样，管理员通常会给每个用户创建证书（通常包含了用户的名字和电子邮件地址）。这个证书会被放置在浏览器中，并在每次连接到服务器时由服务器检查。

###### 局限性

TLS有两种策略：简单策略和交互策略。交互策略更为安全，但需要用户在他们的浏览器中安装个人的[证书](https://zh.wikipedia.org/wiki/%E6%95%B0%E5%AD%97%E8%AF%81%E4%B9%A6)来进行[认证](https://zh.wikipedia.org/wiki/%E8%AE%A4%E8%AF%81)。

不管使用了哪种策略，协议所能提供的保护总强烈地依赖于浏览器的实现和服务器软件所支持的[加密算法](https://zh.wikipedia.org/wiki/%E5%8A%A0%E5%AF%86%E7%AE%97%E6%B3%95)。

HTTPS并不能防止站点被[网络蜘蛛](https://zh.wikipedia.org/wiki/%E7%BD%91%E7%BB%9C%E8%9C%98%E8%9B%9B)抓取。在某些情形中，被加密资源的[URL](https://zh.wikipedia.org/wiki/%E7%BB%9F%E4%B8%80%E8%B5%84%E6%BA%90%E5%AE%9A%E4%BD%8D%E7%AC%A6)可仅通过截获请求和响应的大小推得，[[10]](https://zh.wikipedia.org/wiki/%E8%B6%85%E6%96%87%E6%9C%AC%E4%BC%A0%E8%BE%93%E5%AE%89%E5%85%A8%E5%8D%8F%E8%AE%AE#cite_note-10)这就可使攻击者同时知道明文（公开的静态内容）和密文（被加密过的明文），从而使[选择密文攻击](https://zh.wikipedia.org/wiki/%E9%80%89%E6%8B%A9%E5%AF%86%E6%96%87%E6%94%BB%E5%87%BB)成为可能。

因为[SSL](https://zh.wikipedia.org/wiki/%E5%AE%89%E5%85%A8%E5%A5%97%E6%8E%A5%E5%B1%82)在HTTP之下工作，对上层协议一无所知，所以SSL服务器只能为一个IP地址/端口组合提供一个证书。[[11]](https://zh.wikipedia.org/wiki/%E8%B6%85%E6%96%87%E6%9C%AC%E4%BC%A0%E8%BE%93%E5%AE%89%E5%85%A8%E5%8D%8F%E8%AE%AE#cite_note-11)这就意味着在大部分情况下，使用HTTPS的同时支持[基于名字的虚拟主机](https://zh.wikipedia.org/wiki/%E8%99%9A%E6%8B%9F%E4%B8%BB%E6%9C%BA)是不很现实的。一种叫[域名指示](https://zh.wikipedia.org/wiki/%E5%9F%9F%E5%90%8D%E6%8C%87%E7%A4%BA)（SNI）的方案通过在加密连接创建前向服务器发送主机名解决了这一问题。[Firefox](https://zh.wikipedia.org/wiki/Firefox) 2、[Opera](https://zh.wikipedia.org/wiki/Opera) 8和运行在[Windows Vista](https://zh.wikipedia.org/wiki/Windows_Vista)的[Internet Explorer](https://zh.wikipedia.org/wiki/Internet_Explorer) 7都加入了对SNI的支持。[[12]](https://zh.wikipedia.org/wiki/%E8%B6%85%E6%96%87%E6%9C%AC%E4%BC%A0%E8%BE%93%E5%AE%89%E5%85%A8%E5%8D%8F%E8%AE%AE#cite_note-12)[[13]](https://zh.wikipedia.org/wiki/%E8%B6%85%E6%96%87%E6%9C%AC%E4%BC%A0%E8%BE%93%E5%AE%89%E5%85%A8%E5%8D%8F%E8%AE%AE#cite_note-13)[[14]](https://zh.wikipedia.org/wiki/%E8%B6%85%E6%96%87%E6%9C%AC%E4%BC%A0%E8%BE%93%E5%AE%89%E5%85%A8%E5%8D%8F%E8%AE%AE#cite_note-14)

因为HTTPS连接所用的[公钥](https://zh.wikipedia.org/wiki/%E5%85%AC%E9%92%A5)以明文传输，因此中国大陆的[防火长城](https://zh.wikipedia.org/wiki/%E9%98%B2%E7%81%AB%E9%95%BF%E5%9F%8E)可以对特定网站按照匹配的黑名单证书，通过伪装成对方向连接两端的计算机发送RST包干扰两台计算机间正常的[TCP](https://zh.wikipedia.org/wiki/TCP)通讯，以打断与特定IP地址之间的443端口握手，或者直接使握手的数据包丢弃，导致握手失败，从而导致TLS连接失败。[[15]](https://zh.wikipedia.org/wiki/%E8%B6%85%E6%96%87%E6%9C%AC%E4%BC%A0%E8%BE%93%E5%AE%89%E5%85%A8%E5%8D%8F%E8%AE%AE#cite_note-15)这也是一种互联网信息审查和屏蔽的技术手段。

#### 转换重写规则

并不知道这是啥！

参考地址：<http://nginx.org/en/docs/http/converting_rewrite_rules.html>

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：

People who during their shared hosting life used to configure *everything* using *only* Apache’s .htaccess files, usually translate the following rules:

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;

}

怎么发送要进行websocket协议转换的请求？

#### WebSocket 代理

参考地址：<http://nginx.org/en/docs/http/websocket.html>

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。可是它在反向代理中不能使用，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。

##### Hop-by-hop延伸：

为了确定高速缓存和无缓存代理中的行为。我们把http头划分为两种：

end-to-end：它们被转发到请求或者返回结果的最后接收地址。在返回结果中的end-to-end头需要被存成高速缓存实体的一部分，并以任意返回结果格式传送。

Hop-by-hop:

它们只有在单一传输等级连接上才有意义，不会存在高速缓存或者被代理转发。

下列的HTTP/1.1头是hop-by-hop类型：

-Connection

-keep-Alive

-proxy-authenticate

-Proxy-Authorization

-TE

-Trailers

-Transfer-Encoding

-Upgrade

其它的所有在HTTP/1.1中定义头字段是end-to-end头。

## 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

两种方式，kill发送指令和命令行发送指令

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

通过抢accept\_mutex锁来接收请求并处理，保证各个进程之间互不影响。

##### 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。

事件被注册入epoll中，进程调度epoll中事件过程是阻塞的，超时时间由一颗维护所有事件超时时间的红黑树提供的最小时间计算的。平时进程使用epoll中的epoll\_wait来等待事件被触发。

##### 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。

##### Epoll延伸：

epoll是linux内核的可拓展I/O事件通知机制，耗时为O(1)。

###### 程序接口：

int epoll\_create(int size);

创建epoll对象并回传其描述符。

int epoll\_ctl(int epfd, int op, int fd, **struct** epoll\_event \*event);

将要交由内核管控的文件描述符加入epoll对象并设置触发条件。

int epoll\_wait(int epfd, **struct** epoll\_event \*events, int maxevents, int timeout);

等待已注册之事件被触发或计时终了。

###### 两种触发模式：

epoll提供edge-triggered及level-triggered模式。在edge-trigger模式中，epoll\_wait仅会在新的事件首次被加入epoll 对象时返回；于level-triggered模式下，epoll\_wait在事件状态未变更前将不断被触发。

举例来说，倘若有一个已经于epoll注册之[管线](https://zh.wikipedia.org/w/index.php?title=Anonymous_pipe&action=edit&redlink=1)接获数据，epoll\_wait将返回，并发出数据读取的信号。现假设缓冲器的数据仅有部分被读取并处理，在level-triggered模式下，任何对epoll\_wait之调用都将即刻返回，直到缓冲器中的数据全部被读取；然而，在edge-triggered的情境下，epoll\_wait仅会于再次接收到新数据(亦即，新数据被写入管线)时返回。

对tcp的封装，保存到ngx\_connection\_t的结构体。可作为客户端给其它server发请求。

### 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。

跟一个进程能打开的最大fd数有关(nofile)，可通过work\_connections设置,超过nofile则为nofile

##### 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

通过计算ngx\_accept\_disabled的值来判断是否去获取accept\_mutex锁，进而获取新请求。大于0时不去获取该锁。

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;

}

}

}

##### Socket延伸

参考：<https://zh.wikipedia.org/zh-cn/Berkeley%E5%A5%97%E6%8E%A5%E5%AD%97>

###### Socket是什么？

 Socket是应用层与TCP/IP协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。

###### Socket的位置？



###### 处理过程：



###### API概要

这个列表是一个Berkeley套接字API库提供的函数或者方法的概要：

socket() 创建一个新的确定类型的套接字，类型用一个整型数值标识（[文件描述符](https://zh.wikipedia.org/wiki/%E6%96%87%E4%BB%B6%E6%8F%8F%E8%BF%B0%E7%AC%A6)），并为它分配系统资源。

bind() 一般用于服务器端，将一个套接字与一个套接字地址结构相关联，比如，一个指定的本地端口和IP地址。

listen() 用于服务器端，使一个绑定的TCP套接字进入监听状态。

connect() 用于客户端，为一个套接字分配一个自由的本地端口号。 如果是TCP套接字的话，它会试图获得一个新的TCP连接。

accept() 用于服务器端。 它接受一个从远端客户端发出的创建一个新的TCP连接的接入请求，创建一个新的套接字，与该连接相应的套接字地址相关联。

send()和recv(),或者write()和read(),或者recvfrom()和sendto(), 用于往/从远程套接字发送和接受数据。

close() 用于系统释放分配给一个套接字的资源。 如果是TCP，连接会被中断。

gethostbyname()和gethostbyaddr() 用于解析主机名和地址。

select() 用于修整有如下情况的套接字列表： 准备读，准备写或者是有错误。

poll() 用于检查套接字的状态。 套接字可以被测试，看是否可以写入、读取或是有错误。

getsockopt() 用于查询指定的套接字一个特定的套接字选项的当前值。

setsockopt() 用于为指定的套接字设定一个特定的套接字选项。

对http请求的封装，放到ngx\_http\_request\_t结构中，处理过程分阶段进行。先处理请求行，解析的参数放在该结构中，接着处理请求头，根据一些特殊请求头运行对应的处理程序（hash映射表中），然后对请求体进行多阶段处理，最后通过filter进行结果处理。

#### 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。

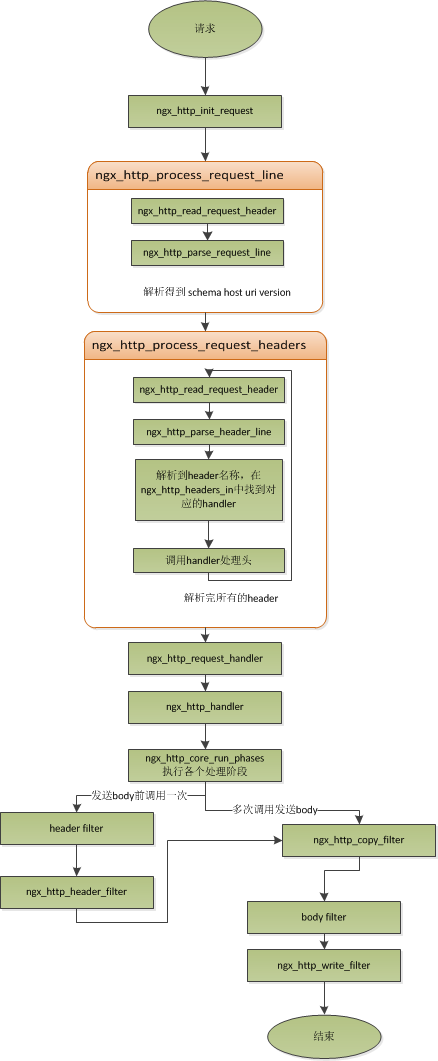
那接下来，简要讲讲nginx是如何处理一个完整的请求的。对于nginx来说，一个请求是从ngx\_http\_init\_request开始的，在这个函数中，会设置读事件为ngx\_http\_process\_request\_line，也就是说，接下来的网络事件，会由ngx\_http\_process\_request\_line来执行。从ngx\_http\_process\_request\_line的函数名，我们可以看到，这就是来处理请求行的，正好与之前讲的，处理请求的第一件事就是处理请求行是一致的。通过ngx\_http\_read\_request\_header来读取请求数据。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

一个连接可执行多个请求，前一个请求完成后在发第二个请求。

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

一个连接进行多次请求，无需等待前个请求返回结果。Nginx端使用buffer存储读取到的请求数据。

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。

延迟关闭，当出错时，为了避免客户端没收到错误信息就发送RST包直接关闭连接，采用的关掉写操作，不发送RST包，等待lingering\_timeou超时时间关闭读操作。

##### 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

以长度而不是’\0’去判断字符串结束

#### ngx\_str\_t

在nginx源码目录的src/core下面的ngx\_string.h|c里面，包含了字符串的封装以及字符串相关操作的api。nginx提供了一个带长度的字符串结构ngx\_str\_t，它的原型如下：

typedef struct {

size\_t len;

u\_char \*data;

} ngx\_str\_t;

字符结束由长度表示，而不是’\0’。好处：不用计算字符串长度，data指针可重复引用别的字符串内存，不用去copy一份。

##### 相关操作

参数必须是一个常量字符串。

#define ngx\_string(str) { sizeof(str) - 1, (u\_char \*) str }

空字符串

#define ngx\_null\_string { 0, NULL }

Text必须为常量字符串

#define ngx\_str\_set(str, text) \

(str)->len = sizeof(text) - 1; (str)->data = (u\_char \*) text

用于设置字符串为空

#define ngx\_str\_null(str) (str)->len = 0; (str)->data = NULL

ngx\_string与ngx\_null\_string是“{，}”格式的，故只能用于赋值时初始化

更多参见：

<http://tengine.taobao.org/book/chapter_02.html#ngx-str-t-100>

#### ngx\_pool\_t

对于nginx处理的每个http request, nginx会生成一个ngx\_pool\_t对象与这个http request关联，所有处理过程中需要申请的资源都从这个ngx\_pool\_t对象中获取，当这个http request处理完成以后，所有在处理过程中申请的资源，都将随着这个关联的ngx\_pool\_t对象的销毁而释放。

它提供了一种机制，帮助管理一系列的资源（如内存，文件等），使得对这些资源的使用和释放统一进行，免除了使用过程中考虑到对各种各样资源的什么时候释放，是否遗漏了释放的担心。

使用ngx\_pool\_t这个数据结构的时候，所有的资源的释放都在这个对象被销毁的时刻，统一进行了释放，那么就会带来一个问题，就是这些资源的生存周期（或者说被占用的时间）是跟ngx\_pool\_t的生存周期基本一致（ngx\_pool\_t也提供了少量操作可以提前释放资源）。

相关结构和位置：

ngx\_pool\_t相关结构及操作被定义在文件src/core/ngx\_palloc.h|c中。

typedef struct ngx\_pool\_s ngx\_pool\_t;

struct ngx\_pool\_s {

ngx\_pool\_data\_t d;

size\_t max;

ngx\_pool\_t \*current;

ngx\_chain\_t \*chain;

ngx\_pool\_large\_t \*large;

ngx\_pool\_cleanup\_t \*cleanup;

ngx\_log\_t \*log;

};

##### 相关操作：

ngx\_pool\_t \*ngx\_create\_pool(size\_t size, ngx\_log\_t \*log);

创建一个初始节点大小为size的pool，log为后续在该pool上进行操作时输出日志的对象。 需要说明的是size的选择，size的大小必须小于等于NGX\_MAX\_ALLOC\_FROM\_POOL，且必须大于sizeof(ngx\_pool\_t)。

更多操作参见：

<http://tengine.taobao.org/book/chapter_02.html#ngx-pool-t-100>

#### ngx\_chain\_t(100%)

nginx的filter模块在处理从别的filter模块或者是handler模块传递过来的数据（实际上就是需要发送给客户端的http response）。这个传递过来的数据是以一个链表的形式(ngx\_chain\_t)。而且数据可能被分多次传递过来。也就是多次调用filter的处理函数，以不同的ngx\_chain\_t。

该结构被定义在src/core/ngx\_buf.h|c。下面我们来看一下ngx\_chain\_t的定义。

typedef struct ngx\_chain\_s ngx\_chain\_t;

struct ngx\_chain\_s {

ngx\_buf\_t \*buf;

ngx\_chain\_t \*next;

};

就2个字段，next指向这个链表的下个节点。buf指向实际的数据。所以在这个链表上追加节点也是非常容易，只要把末尾元素的next指针指向新的节点，把新节点的next赋值为NULL即可。

ngx\_chain\_t \*ngx\_alloc\_chain\_link(ngx\_pool\_t \*pool);

该函数创建一个ngx\_chain\_t的对象，并返回指向对象的指针，失败返回NULL。

#define ngx\_free\_chain(pool, cl) \

cl->next = pool->chain; \

pool->chain = cl

该宏释放一个ngx\_chain\_t类型的对象。如果要释放整个chain，则迭代此链表，对每个节点使用此宏即可。

**注意:** 对ngx\_chaint\_t类型的释放，并不是真的释放了内存，而仅仅是把这个对象挂在了这个pool对象的一个叫做chain的字段对应的chain上，以供下次从这个pool上分配ngx\_chain\_t类型对象的时候，快速的从这个pool->chain上取下链首元素就返回了，当然，如果这个链是空的，才会真的在这个pool上使用ngx\_palloc函数进行分配。

#### ngx\_buf\_t

这个ngx\_buf\_t就是这个ngx\_chain\_t链表的每个节点的实际数据。该结构实际上是一种抽象的数据结构，它代表某种具体的数据。这个数据可能是指向内存中的某个缓冲区，也可能指向一个文件的某一部分，也可能是一些纯元数据（元数据的作用在于指示这个链表的读取者对读取的数据进行不同的处理）。

该数据结构位于src/core/ngx\_buf.h|c文件中。我们来看一下它的定义。

struct ngx\_buf\_s {

u\_char \*pos;

u\_char \*last;

off\_t file\_pos;

off\_t file\_last;

u\_char \*start; /\* start of buffer \*/

u\_char \*end; /\* end of buffer \*/

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;

unsigned last\_shadow:1;

unsigned temp\_file:1;

/\* STUB \*/ int num;

};

|  |  |
| --- | --- |
| **pos:** | 当buf所指向的数据在内存里的时候，pos指向的是这段数据开始的位置。 |
| **last:** | 当buf所指向的数据在内存里的时候，last指向的是这段数据结束的位置。 |
| **file\_pos:** | 当buf所指向的数据是在文件里的时候，file\_pos指向的是这段数据的开始位置在文件中的偏移量。 |
| **file\_last:** | 当buf所指向的数据是在文件里的时候，file\_last指向的是这段数据的结束位置在文件中的偏移量。 |
| **start:** | 当buf所指向的数据在内存里的时候，这一整块内存包含的内容可能被包含在多个buf中(比如在某段数据中间插入了其他的数据，这一块数据就需要被拆分开)。那么这些buf中的start和end都指向这一块内存的开始地址和结束地址。而pos和last指向本buf所实际包含的数据的开始和结尾。 |
| **end:** | 解释参见start。 |
| **tag:** | 实际上是一个void\*类型的指针，使用者可以关联任意的对象上去，只要对使用者有意义。 |
| **file:** | 当buf所包含的内容在文件中时，file字段指向对应的文件对象。 |
| **shadow:** | 当这个buf完整copy了另外一个buf的所有字段的时候，那么这两个buf指向的实际上是同一块内存，或者是同一个文件的同一部分，此时这两个buf的shadow字段都是指向对方的。那么对于这样的两个buf，在释放的时候，就需要使用者特别小心，具体是由哪里释放，要提前考虑好，如果造成资源的多次释放，可能会造成程序崩溃！ |
| **temporary:** | 为1时表示该buf所包含的内容是在一个用户创建的内存块中，并且可以被在filter处理的过程中进行变更，而不会造成问题。 |
| **memory:** | 为1时表示该buf所包含的内容是在内存中，但是这些内容却不能被进行处理的filter进行变更。 |
| **mmap:** | 为1时表示该buf所包含的内容是在内存中, 是通过mmap使用内存映射从文件中映射到内存中的，这些内容却不能被进行处理的filter进行变更。 |
| **recycled:** | 可以回收的。也就是这个buf是可以被释放的。这个字段通常是配合shadow字段一起使用的，对于使用ngx\_create\_temp\_buf 函数创建的buf，并且是另外一个buf的shadow，那么可以使用这个字段来标示这个buf是可以被释放的。 |
| **in\_file:** | 为1时表示该buf所包含的内容是在文件中。 |
| **flush:** | 遇到有flush字段被设置为1的的buf的chain，则该chain的数据即便不是最后结束的数据（last\_buf被设置，标志所有要输出的内容都完了），也会进行输出，不会受postpone\_output配置的限制，但是会受到发送速率等其他条件的限制。 |
| **sync:** |  |
| **last\_buf:** | 数据被以多个chain传递给了过滤器，此字段为1表明这是最后一个buf。 |
| **last\_in\_chain:** | 在当前的chain里面，此buf是最后一个。特别要注意的是last\_in\_chain的buf不一定是last\_buf，但是last\_buf的buf一定是last\_in\_chain的。这是因为数据会被以多个chain传递给某个filter模块。 |
| **last\_shadow:** | 在创建一个buf的shadow的时候，通常将新创建的一个buf的last\_shadow置为1。 |
| **temp\_file:** | 由于受到内存使用的限制，有时候一些buf的内容需要被写到磁盘上的临时文件中去，那么这时，就设置此标志 。 |

对于此对象的创建，可以直接在某个ngx\_pool\_t上分配，然后根据需要，给对应的字段赋值。也可以使用定义好的2个宏：

#define ngx\_alloc\_buf(pool) ngx\_palloc(pool, sizeof(ngx\_buf\_t))

#define ngx\_calloc\_buf(pool) ngx\_pcalloc(pool, sizeof(ngx\_buf\_t))

这两个宏使用类似函数，也是不说自明的。

对于创建temporary字段为1的buf（就是其内容可以被后续的filter模块进行修改），可以直接使用函数ngx\_create\_temp\_buf进行创建。

ngx\_buf\_t \*ngx\_create\_temp\_buf(ngx\_pool\_t \*pool, size\_t size);

该函数创建一个ngx\_but\_t类型的对象，并返回指向这个对象的指针，创建失败返回NULL。

对于创建的这个对象，它的start和end指向新分配内存开始和结束的地方。pos和last都指向这块新分配内存的开始处，这样，后续的操作可以在这块新分配的内存上存入数据。

|  |  |
| --- | --- |
| **pool:** | 分配该buf和buf使用的内存所使用的pool。 |
| **size:** | 该buf使用的内存的大小。 |

为了配合对ngx\_buf\_t的使用，nginx定义了以下的宏方便操作。

#define ngx\_buf\_in\_memory(b) (b->temporary || b->memory || b->mmap)

返回这个buf里面的内容是否在内存里。

#define ngx\_buf\_in\_memory\_only(b) (ngx\_buf\_in\_memory(b) && !b->in\_file)

返回这个buf里面的内容是否仅仅在内存里，并且没有在文件里。

#define ngx\_buf\_special(b) \

((b->flush || b->last\_buf || b->sync) \

&& !ngx\_buf\_in\_memory(b) && !b->in\_file)

返回该buf是否是一个特殊的buf，只含有特殊的标志和没有包含真正的数据。

#define ngx\_buf\_sync\_only(b) \

(b->sync \

&& !ngx\_buf\_in\_memory(b) && !b->in\_file && !b->flush && !b->last\_buf)

返回该buf是否是一个只包含sync标志而不包含真正数据的特殊buf。

#define ngx\_buf\_size(b) \

(ngx\_buf\_in\_memory(b) ? (off\_t) (b->last - b->pos): \

(b->file\_last - b->file\_pos))

返回该buf所含数据的大小，不管这个数据是在文件里还是在内存里。

#### ngx\_array\_t(100%)

ngx\_array\_t是nginx内部使用的数组结构。nginx的数组结构在存储上与大家认知的C语言内置的数组有相似性，比如实际上存储数据的区域也是一大块连续的内存。但是数组除了存储数据的内存以外还包含一些元信息来描述相关的一些信息。下面我们从数组的定义上来详细的了解一下。ngx\_array\_t的定义位于src/core/ngx\_array.c|h里面。

typedef struct ngx\_array\_s ngx\_array\_t;

struct ngx\_array\_s {

void \*elts;

ngx\_uint\_t nelts;

size\_t size;

ngx\_uint\_t nalloc;

ngx\_pool\_t \*pool;

};

|  |  |
| --- | --- |
| **elts:** | 指向实际的数据存储区域。 |
| **nelts:** | 数组实际元素个数。 |
| **size:** | 数组单个元素的大小，单位是字节。 |
| **nalloc:** | 数组的容量。表示该数组在不引发扩容的前提下，可以最多存储的元素的个数。当nelts增长到达nalloc 时，如果再往此数组中存储元素，则会引发数组的扩容。数组的容量将会扩展到原有容量的2倍大小。实际上是分配新的一块内存，新的一块内存的大小是原有内存大小的2倍。原有的数据会被拷贝到新的一块内存中。 |
| **pool:** | 该数组用来分配内存的内存池。 |

下面介绍ngx\_array\_t相关操作函数。

ngx\_array\_t \*ngx\_array\_create(ngx\_pool\_t \*p, ngx\_uint\_t n, size\_t size);

创建一个新的数组对象，并返回这个对象。

|  |  |
| --- | --- |
| **p:** | 数组分配内存使用的内存池； |
| **n:** | 数组的初始容量大小，即在不扩容的情况下最多可以容纳的元素个数。 |
| **size:** | 单个元素的大小，单位是字节。 |

void ngx\_array\_destroy(ngx\_array\_t \*a);

销毁该数组对象，并释放其分配的内存回内存池。

void \*ngx\_array\_push(ngx\_array\_t \*a);

在数组a上新追加一个元素，并返回指向新元素的指针。需要把返回的指针使用类型转换，转换为具体的类型，然后再给新元素本身或者是各字段（如果数组的元素是复杂类型）赋值。

void \*ngx\_array\_push\_n(ngx\_array\_t \*a, ngx\_uint\_t n);

在数组a上追加n个元素，并返回指向这些追加元素的首个元素的位置的指针。

static ngx\_inline ngx\_int\_t ngx\_array\_init(ngx\_array\_t \*array, ngx\_pool\_t \*pool, ngx\_uint\_t n, size\_t size);

如果一个数组对象是被分配在堆上的，那么当调用ngx\_array\_destroy销毁以后，如果想再次使用，就可以调用此函数。

如果一个数组对象是被分配在栈上的，那么就需要调用此函数，进行初始化的工作以后，才可以使用。

**注意事项:** 由于使用ngx\_palloc分配内存，数组在扩容时，旧的内存不会被释放，会造成内存的浪费。因此，最好能提前规划好数组的容量，在创建或者初始化的时候一次搞定，避免多次扩容，造成内存浪费。

### 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。

配置例子：

user nobody;

worker\_processes 1;

error\_log logs/error.log info;

events {

worker\_connections 1024;

}

http {

server {

listen 80;

server\_name www.linuxidc.com;

access\_log logs/linuxidc.access.log main;

location / {

index index.html;

root /var/www/linuxidc.com/htdocs;

}

}

server {

listen 80;

server\_name www.Androidj.com;

access\_log logs/androidj.access.log main;

location / {

index index.html;

root /var/www/androidj.com/htdocs;

}

}

}

mail {

auth\_http 127.0.0.1:80/auth.php;

pop3\_capabilities "TOP" "USER";

imap\_capabilities "IMAP4rev1" "UIDPLUS";

server {

listen 110;

protocol pop3;

proxy on;

}

server {

listen 25;

protocol smtp;

proxy on;

smtp\_auth login plain;

xclient off;

}

}

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：

操作系统提供的机制（例如epoll, kqueue等）产生相关的事件。

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。

处理request的header和body。

产生响应，并发送回客户端。

完成request的处理。

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。

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）。

处理请求头。

处理请求体。

如果有的话，调用与此请求（URL或者Location）关联的handler。

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：

获取location配置。

产生适当的响应。

发送response header。

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）。

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。

如果使用[kqueue](http://nginx.org/en/docs/events.html#kqueue) 连接处理办法的办法，那么这个指令会被忽略，因为它需要报告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。

### Nginx变量

### 第三方模块

使用了[ngx\_echo](http://wiki.nginx.org/HttpEchoModule)的 [echo](http://wiki.nginx.org/HttpEchoModule#echo) 的配置指令。

为省去麻烦，之间使用OpenResty的运行环境。

安装：

<http://openresty.org/#Installation>

## 源码分析

### Nginx模块概览

参见：<http://www.evanmiller.org/nginx-modules-guide.html>

编译时候nginx模块要跟主模块一起编译

我们讲提到三种类型的nginx模块：

* 处理器（handler），用来处理请求和加工输出
* 过滤器（filter），处理handler加工出来的输出数据。
* 负载均衡器（load-balancer)，如果有多个合法的后端服务器，那么就选择一个后端服务器，并且把请求转发过去。

跟apache的模块不同，nginx的模块不是动态加载的，也就是说nginx的模块必须和主程序一起编译进去。

多个handler要挂在同一个location，那只有一个能成功挂上，即运行

#### 模块是如何被调用的呢？

一般来说，在服务启动的时候，每一个handler都会根据配置文件找到属于他的location，然后挂上去；如果多个handler要挂在同一个location，那只有一个能成功挂上（不要让多个handler一起竞争）handler有三种返回方式：一切正常，发生错误，或这拒绝处理请求并转向默认handler。

nginx带了两个负载均衡模块：轮询；还有一种是：IP哈希，保证来自同一个IP的请求都能发到同一台机器。

如果handler不报错，就会调用filter。每个location都可以挂n个filter，所以一个响应可以先被压缩，然后分块。这些filter的执行顺序在编译阶段就决定了filter采用了经典的“CHAIN OF RESPONSIBILITY”设计模式：一个filter被调用，做属于它的工作，然后调用下一个，直到最后一个filter被调用，nginx完成这次响应。

多个handler要挂在同一个location，那只有一个能成功挂上，即运行

filter chain真正牛的地方是，每一个filter可以不用等他之前的filter完全结束，就开始对前一个filter的输出进行操作，就像Unix pipeline一样（译注：整个协议是个流，而不是一个块，所以一个响应在这堆filter里不具有原子性，这样filter的效率大大提高）。filter一般在一个4K大小的一个缓冲区里进行操作，但可以在nginx.conf里改。也就是说，例如一个模块可以在完全接收到后端传来的响应之前，就开始对一部分数据进行压缩并传到客户端。

#### 一般的的操作周期：

客户端发了一个http请求 → nginx根据location配置挑了一个handler → （如果一切正常）load-balancer会挑一个后端服务器 → handler做它该做的事，并且把一个一个的输出缓冲发给第一个filter → 第一个filter → 第一个filter把输出传给第二个filter → 第二个给第三个 → 依次往后传 → 最后响应发还给客户端。

这里我所谓“一般的”，是因为nginx模块的调用是可以非常定制化的。他给模块开发这极大的自由去定义模块在何时如何去运行（有时候这个自由甚至显得太大了）。模块调用是通过很多的回调来实现的，也就是说，你要写一堆可以被调用的函数：

#### 一堆回调函数

* 在服务器载入配置文件之前
* 为location和server的每个配置指令
* 当nginx初始化main配置
* 当nginx初始化server配置
* 当nginx合并server和main配置
* 当nginx初始化location配置
* 当nginx合并location还有他上一级的server配置
* 当nginxmaster进程启动（译注：nginx是一个master进程带一堆对worker的工作模式）
* 当一个心的worker进程启动
* 当一个worker进程结束
* 当主进程结束
* 处理一个请求
* 过滤响应的headers
* 过滤响应的body
* 选择后端服务器
* 初始化发给后端服务器的请求
* 重新初始化发送给后端服务器的请求
* 处理后端服务器发来的响应
* 完成与后端服务器的合作

### 模块的各个组件

基本上作为第三方开发者最可能开发的就是三种类型的模块，即handler，filter和load-balancer。

#### 模块基本结构

用户可以通过配置来控制模块的行为，而模块配置结构用来存储这些配置信息。

##### 模块配置结构

Nginx配置信息分为几个作用域。包括main、server以及location。

每个模块的配置指令可以出现在这些作用域中，不用作用域中的配置信息的需要不同的数据结构去存储。

命名习惯：ngx\_http\_<module name>\_(main|srv|loc)\_conf\_t。

typedef struct {

ngx\_uint\_t methods;

ngx\_flag\_t create\_full\_put\_path;

ngx\_uint\_t access;

} ngx\_http\_dav\_loc\_conf\_t;

注意，nginx有一些特别的数据类型（ngx\_uint\_t 和 ngx\_flag\_t）；这些都是原始数据类型的别名（在 core/ngx\_config.h里可以找到这些别名的定义）。

配置结构里的这些成员会被模块的指令赋值。

##### 模块配置指令

一个模块的配置指令是定义在一个ngx\_command\_ts的静态数组中的。

static ngx\_command\_t ngx\_http\_circle\_gif\_commands[] = {

{ ngx\_string("circle\_gif"),

NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_NOARGS,

ngx\_http\_circle\_gif,

NGX\_HTTP\_LOC\_CONF\_OFFSET,

0,

NULL },

{ ngx\_string("circle\_gif\_min\_radius"),

NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,

ngx\_conf\_set\_num\_slot,

NGX\_HTTP\_LOC\_CONF\_OFFSET,

offsetof(ngx\_http\_circle\_gif\_loc\_conf\_t, min\_radius),

NULL },

...

ngx\_null\_command

};

ngx\_command\_t的定义，位于src/core/ngx\_conf\_file.h中

struct ngx\_command\_s {

ngx\_str\_t name;

ngx\_uint\_t type;

char \*(\*set)(ngx\_conf\_t \*cf, ngx\_command\_t \*cmd, void \*conf);

ngx\_uint\_t conf;

ngx\_uint\_t offset;

void \*post;

};

|  |  |
| --- | --- |
| name: | 配置指令的名称。这是个ngx\_str\_t类型的数据 |
| type: | 是一个flag的集合，用来标记指令在哪是合法的，有几个参数什么的。在使用过程中，通常是下边这些数值的按位或：  NGX\_HTTP\_MAIN\_CONF：指令在main配置中有效  NGX\_HTTP\_SRV\_CONF：指令在server配置中有效  NGX\_HTTP\_LOC\_CONF：指令在location配置中有效  NGX\_HTTP\_UPS\_CONF：指令在upstream配置中有效  NGX\_CONF\_NOARGS：指令没有参数  NGX\_CONF\_TAKE1：指令读入1个参数  NGX\_CONF\_TAKE2：指令读入2个参数  ...  NGX\_CONF\_TAKE7：指令读入带7个参数  NGX\_CONF\_FLAG：指令读入一个boolean型（“on”或“off”）  NGX\_CONF\_1MORE：指令至少读入1个参数  NGX\_CONF\_2MORE：指令至少读入2个参数  还有一些其他的选项，参见core/ngx\_conf\_file.h |

|  |  |
| --- | --- |
| set: | 结构成员是一个指向设定该模块配置函数的指针；一般来说这个函数用来处理该指令接收到的参数并把处理后的值存入相应的结构中。这个设定函数可以带三个参数：  来看一下这个函数指针要求的函数原型。 |

char \*(\*set)(ngx\_conf\_t \*cf, ngx\_command\_t \*cmd, void \*conf);

再看一下这个函数被调用的时候，传入的三个参数。

cf: 指向ngx\_conf\_t的指针，包含传给指令的参数。特别注意的是这个参数的args字段是一个ngx\_str\_t类型的数组，该数组的首个元素是这个配置指令本身，第二个元素是指令的第一个参数，第三个元素是第二个参数，依次类推。

cmd: 指向配置指令对应的ngx\_command\_t结构的指针。

conf: 指向模块自定义配置结构的指针。当检测到相应指令时，这些设置函数就会被调用。nginx提供了一堆把自定义的配置结构转换成标准类型的函数。这些函数包括：

* ngx\_conf\_set\_flag\_slot：把“on”和“off”转成1和0
* ngx\_conf\_set\_str\_slot：把字符串格式化为以ngx\_str\_t类型
* ngx\_conf\_set\_num\_slot：解析数字并转换为一个整型
* ngx\_conf\_set\_size\_slot：解析表示大小的值（“8k”，“1m”等）并格式化成size\_t格式
* 还有别的，不过都不复杂（参考：core/ngx\_conf\_file.h）。模块开发者也可以把对自定义函数的引用写在这里，用来替代内建函数。

|  |  |
| --- | --- |
| **conf:** | conf标示数据是放在main，server还是location（NGX\_HTTP\_MAIN\_CONF\_OFFSET、NGX\_HTTP\_SRV\_CONF\_OFFSET、或NGX\_HTTP\_LOC\_CONF\_OFFSET）。 |
| **offset:** | 指定该配置项值的精确存放位置，一般指定为某一个结构体变量的字段偏移。因为对于配置信息的存储，一般我们都是定义个结构体来存储的。那么比如我们定义了一个结构体A，该项配置的值需要存储到该结构体的b字段。那么在这里就可以填写为offsetof(A, b)。对于有些配置项，它的值不需要保存或者是需要保存到更为复杂的结构中时，这里可以设置为0。 |
| **post:** | 该字段存储一个指针。可以指向任何一个在读取配置过程中需要的数据，以便于进行配置读取的处理。大多数时候，都不需要，所以简单地设为0即可。 |

**需要注意的是，就是在ngx\_http\_hello\_commands这个数组定义的最后，都要加一个ngx\_null\_command作为结尾。**

##### 模块上下文

这是一个静态的ngx\_http\_module\_t结构，由一系列负责创建配置和合并配置的函数引用组成。命名规则是：ngx\_http\_<模块名>\_module\_ctx。函数引用依次为：

* **preconfiguration：**在创建和读取该模块的配置信息之前被调用。
* **postconfiguration：**在创建和读取该模块的配置信息之后被调用。
* **create\_main\_conf:** 创建main配置时调用（例如：分配内存空间和设置默认值）
* **init\_main\_conf:** 初始化main配置时调用（例如：用nginx.conf中的值覆盖默认值）
* **create\_srv\_conf:** 创建server配置时调用
* **init\_main\_conf:** 与main配置合并时调用
* **merge\_srv\_conf:** 创建location配置时调用
* **merge\_loc\_conf:** 与server配置合并时调用

不同的参数取决于不同的功能。这里有一个摘自http/ngx\_http\_config.h的结构定义，可以看出不同的回调函数之间的区别：

typedef struct {

    ngx\_int\_t   (\*preconfiguration)(ngx\_conf\_t \*cf);

  ngx\_int\_t   (\*postconfiguration)(ngx\_conf\_t \*cf);

    void       \*(\*create\_main\_conf)(ngx\_conf\_t \*cf);

    char       \*(\*init\_main\_conf)(ngx\_conf\_t \*cf, void \*conf);

    void       \*(\*create\_srv\_conf)(ngx\_conf\_t \*cf);

    char       \*(\*merge\_srv\_conf)(ngx\_conf\_t \*cf, void \*prev, void \*conf);

    void       \*(\*create\_loc\_conf)(ngx\_conf\_t \*cf);

    char       \*(\*merge\_loc\_conf)(ngx\_conf\_t \*cf, void \*prev, void \*conf);

} ngx\_http\_module\_t;

你可以把不需要的函数就设成NULL，nginx会搞定它。

大多数的handler都只用最后两个：一个用来给特定的配置分配内存（叫作：ngx\_http\_<模块名>\_create\_loc\_conf），另一个用来设置默认值，和合并继承来的配置（叫作：ngx\_http\_<模块名>\_merge\_loc\_conf）。合并函数还负责检测到非法配置后的报错；这些错误会导致服务终止。

这里是一个示例模块的context结构：

static ngx\_http\_module\_t  ngx\_http\_circle\_gif\_module\_ctx = {

    NULL,                          /\* preconfiguration \*/

    NULL,                          /\* postconfiguration \*/

    NULL,                          /\* create main configuration \*/

    NULL,                          /\* init main configuration \*/

    NULL,                          /\* create server configuration \*/

    NULL,                          /\* merge server configuration \*/

    ngx\_http\_circle\_gif\_create\_loc\_conf,  /\* create location configuration \*/

    ngx\_http\_circle\_gif\_merge\_loc\_conf /\* merge location configuration \*/

};

###### create\_loc\_conf

这是一个最精简的create\_loc\_conf函数，出自circle\_gif模块（[查看原码](https://www.google.com/url?q=http://www.google.com/url?q%3Dhttp%253A%252F%252Fwww.evanmiller.org%252Fnginx%252Fngx_http_circle_gif_module.c.txt%26sa%3DD%26sntz%3D1%26usg%3DAFQjCNGAD-yvHdnvskeKDqBsoJWqh3OMxg&sa=D&ust=1457488504342000&usg=AFQjCNG3qd6hEtIvzMWANdGyBkDXChIHSA)）。它输入一个指令结构（ngx\_conf\_t），返回一个新建模块的配置结构（在这个例子中是：ngx\_http\_circle\_gif\_loc\_conf\_t）。

static void \*

ngx\_http\_circle\_gif\_create\_loc\_conf(ngx\_conf\_t \*cf)

{

ngx\_http\_circle\_gif\_loc\_conf\_t \*conf;

conf = ngx\_pcalloc(cf->pool, sizeof(ngx\_http\_circle\_gif\_loc\_conf\_t));

if (conf == NULL) {

return NGX\_CONF\_ERROR;

分配内存，返回新建模块的配置结构

}

conf->min\_radius = NGX\_CONF\_UNSET\_UINT;

conf->max\_radius = NGX\_CONF\_UNSET\_UINT;

return conf;

}

首先要注意nginx的内存分配，只要用ngx\_palloc（malloc的封装）或ngx\_palloc（calloc的封装），系统会自动释放内存。  
UNSET常量有可能是NGX\_CONF\_UNSET\_UINT、NGX\_CONF\_UNSET\_PTR、NGX\_CONF\_UNSET\_SIZE或NGX\_CONF\_UNSET\_MSEC，并且表示全部的NGX\_CONF\_UNSET。UNSET常量告诉合并函数哪些值应该被覆盖。

###### merge\_loc\_conf

这里是circle\_gif模块中用到的合并函数：

static char \*

ngx\_http\_circle\_gif\_merge\_loc\_conf(ngx\_conf\_t \*cf, void \*parent, void \*child)

{

ngx\_http\_circle\_gif\_loc\_conf\_t \*prev = parent;

ngx\_http\_circle\_gif\_loc\_conf\_t \*conf = child;

ngx\_conf\_merge\_uint\_value(conf->min\_radius, prev->min\_radius, 10);

ngx\_conf\_merge\_uint\_value(conf->max\_radius, prev->max\_radius, 20);

if (conf->min\_radius < 1) {

ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,

"min\_radius must be equal or more than 1");

return NGX\_CONF\_ERROR;

}

if (conf->max\_radius < conf->min\_radius) {

ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,

"max\_radius must be equal or more than min\_radius");

return NGX\_CONF\_ERROR;

}

return NGX\_CONF\_OK;

}

首先要知道nginx提供了很好用的合并不同数据类型的函数（ngx\_conf\_merge\_<数据类型>\_value）；这些参数是：

1. *this* location's value

合并函数用来合并配置文件参数，获取的顺序如下，结果放在第一个参数中

      1、当前location的值

1. the value to inherit if #1 is not set

      2、如果第一个参数没有赋值

1. the default if neither #1 nor #2 is set

      3、如果前两个参数都没有赋值

结果放在第一个参数中，合并函数包括ngx\_conf\_merge\_size\_value和ngx\_conf\_merge\_msed\_value，还有其他的参考：core/ngx\_conf\_file.h。

有一个问题：第一个参数是用来付值的，那么这些默认值是如何被写进去的呢？

回答：这些函数其实都是预处理命令（在编译之前，他们就会被扩展成一些“if”语句什么的）。

还有要注意的就是错误是如何产生的；函数去写日志文件，并返回NGX\_CONF\_ERROR。返回代码会终止服务。（如果这些消息被标记了NGX\_LOG\_EMERG，那么消息也会打印到标准输出；core/nginx\_log.h有完整的log级别定义。）

Nginx里面的配置信息都是上下一层层的嵌套的，对于具体某个location的话，对于同一个配置，如果当前层次没有定义，那么就使用上层的配置，否则使用当前层次的配置。

这些配置信息一般默认都应该设为一个未初始化的值，针对这个需求，Nginx定义了一系列的宏定义来代表各种配置所对应数据类型的未初始化值，如下：

#define NGX\_CONF\_UNSET -1

#define NGX\_CONF\_UNSET\_UINT (ngx\_uint\_t) -1

#define NGX\_CONF\_UNSET\_PTR (void \*) -1

#define NGX\_CONF\_UNSET\_SIZE (size\_t) -1

#define NGX\_CONF\_UNSET\_MSEC (ngx\_msec\_t) -1

又因为对于配置项的合并，逻辑都类似，也就是前面已经说过的，如果在本层次已经配置了，也就是配置项的值已经被读取进来了（那么这些配置项的值就不会等于上面已经定义的那些UNSET的值），就使用本层次的值作为定义合并的结果，否则，使用上层的值，如果上层的值也是这些UNSET类的值，那就赋值为默认值，否则就使用上层的值作为合并的结果。对于这样类似的操作，Nginx定义了一些宏操作来做这些事情，我们来看其中一个的定义。

#define ngx\_conf\_merge\_uint\_value(conf, prev, default) \

if (conf == NGX\_CONF\_UNSET\_UINT) { \

conf = (prev == NGX\_CONF\_UNSET\_UINT) ? default : prev; \

}

显而易见，这个逻辑确实比较简单，所以其它的宏定义也类似，我们就列具其中的一部分吧。

ngx\_conf\_merge\_value

ngx\_conf\_merge\_ptr\_value

ngx\_conf\_merge\_uint\_value

ngx\_conf\_merge\_msec\_value

ngx\_conf\_merge\_sec\_value

等等。

##### 模块定义

我们需要定义一个nax\_module\_t类型的变量来说明模块本身的信息，包括定义的配置信息，上下文信息，都是通过这个模块来告诉nginx。

我们先来看下ngx\_module\_t的定义

typedef struct ngx\_module\_s ngx\_module\_t;

struct ngx\_module\_s {

ngx\_uint\_t ctx\_index;

ngx\_uint\_t index;

ngx\_uint\_t spare0;

ngx\_uint\_t spare1;

ngx\_uint\_t abi\_compatibility;

ngx\_uint\_t major\_version;

ngx\_uint\_t minor\_version;

void \*ctx;

ngx\_command\_t \*commands;

ngx\_uint\_t type;

ngx\_int\_t (\*init\_master)(ngx\_log\_t \*log);

ngx\_int\_t (\*init\_module)(ngx\_cycle\_t \*cycle);

ngx\_int\_t (\*init\_process)(ngx\_cycle\_t \*cycle);

ngx\_int\_t (\*init\_thread)(ngx\_cycle\_t \*cycle);

void (\*exit\_thread)(ngx\_cycle\_t \*cycle);

void (\*exit\_process)(ngx\_cycle\_t \*cycle);

void (\*exit\_master)(ngx\_cycle\_t \*cycle);

uintptr\_t spare\_hook0;

uintptr\_t spare\_hook1;

uintptr\_t spare\_hook2;

uintptr\_t spare\_hook3;

uintptr\_t spare\_hook4;

uintptr\_t spare\_hook5;

uintptr\_t spare\_hook6;

uintptr\_t spare\_hook7;

};

#define NGX\_NUMBER\_MAJOR 3

#define NGX\_NUMBER\_MINOR 1

#define NGX\_MODULE\_V1 0, 0, 0, 0, \

NGX\_DSO\_ABI\_COMPATIBILITY, NGX\_NUMBER\_MAJOR, NGX\_NUMBER\_MINOR

#define NGX\_MODULE\_V1\_PADDING 0, 0, 0, 0, 0, 0, 0, 0

下边我们增加一层，ngx\_module\_t结构。他的变量命名为ngx\_http\_<模块名>\_module。之前说道的context引用和指令执行都包含其中，另外还有回调函数（退出线程/进程，等等）。模块定义有时候就像是一个用来查找与某一特定模块相关的数据的键。模块定义通常就像这样：

ngx\_module\_t  ngx\_http\_<module name>\_module = {

    NGX\_MODULE\_V1,

    &ngx\_http\_<module name>\_module\_ctx, /\* module context \*/

    ngx\_http\_<module name>\_commands,   /\* module directives \*/

    NGX\_HTTP\_MODULE,               /\* module type \*/

    NULL,                          /\* init master \*/

    NULL,                          /\* init module \*/

    NULL,                          /\* init process \*/

    NULL,                          /\* init thread \*/

    NULL,                          /\* exit thread \*/

    NULL,                          /\* exit process \*/

    NULL,                          /\* exit master \*/

    NGX\_MODULE\_V1\_PADDING

};

<module name>的位置就填相应的模块名。模块可以为进程/线程创建和销毁定义回调函数，但是大多数模块都不这么干，保持一个简洁的定义。（参数详情参见：core/ngx\_conf\_file.h）

#### handler模块

Handler模块就是接受来自客户端的请求并产生输出的模块。有些地方说upstream模块实际上也是一种handler模块，只不过它产生的内容来自于从后端服务器获取的，而非在本机产生的。

配置文件中使用location指令可以配置content handler模块，当Nginx系统启动的时候，每个handler模块都有一次机会把自己关联到对应的location上。如果有多个handler模块都关联了同一个location，那么实际上只有一个handler模块真正会起作用。当然大多数情况下，模块开发人员都会避免出现这种情况

处理结果：

处理成功，处理失败（处理的时候发生了错误）或者是拒绝去处理。在拒绝处理的情况下，这个location的处理就会由默认的handler模块来进行处理。例如，当请求一个静态文件的时候，如果关联到这个location上的一个handler模块拒绝处理，就会由默认的ngx\_http\_static\_module模块进行处理，该模块是一个典型的handler模块。

##### 一个处理程序的剖析(非代理)

处理程序通常做四件事:获得location配置,生成一个适当的回应,发送数据头,发送数据体。一个处理程序有一个参数,即请求结构体。一个请求结构体有很多关于客户请求的有用信息,如请求方式,URI,和headers。我们将逐步经历这些步骤。

###### 获取location配置

这部分很简单. 你只需要传入正确的参数和模块定义调用ngx\_http\_get\_module\_loc\_conf即可. 下面是我circle gif 处理的相关部分配置:

static ngx\_int\_t

ngx\_http\_circle\_gif\_handler(ngx\_http\_request\_t \*r)

{

ngx\_http\_circle\_gif\_loc\_conf\_t \*circle\_gif\_config;

circle\_gif\_config = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_circle\_gif\_module);

...

现在可以使用合并函数中我设置的任何变量了.

###### 生成响应

这部分很有趣,是模块真正意义上工作运行的部分.

下面的请求结构一目了然:

typedef struct {

...

/\* the memory pool, used in the ngx\_palloc functions \*/

ngx\_pool\_t \*pool;

ngx\_str\_t uri;

ngx\_str\_t args;

ngx\_http\_headers\_in\_t headers\_in;

...

} ngx\_http\_request\_t;

uri是请求的地址, 例如. "/query.cgi".

args请求参数中,问号后面的(如. "name=john").

headers\_in 包含很多有用的参数,比如cookies和浏览器的信息, 不过很多模块都不需要.有兴趣可以看下[http/ngx\_http\_request.h](http://lxr.evanmiller.org/http/source/http/ngx_http_request.h#L158)

有了这些参数就足够输出很多有用的信息了.全部ngx\_http\_request\_t结构在这里可以找到 [http/ngx\_http\_request.h](http://lxr.evanmiller.org/http/source/http/ngx_http_request.h#L316).

###### 发送头信息

在结构中响应的头信息称为headers\_out, 并在请求结构中引用. 处理器设置后这些头信息后调用ngx\_http\_send\_header(r). 下面是头信息中一些有用的:

typedef stuct {

...

ngx\_uint\_t status;

size\_t content\_type\_len;

ngx\_str\_t content\_type;

ngx\_table\_elt\_t \*content\_encoding;

off\_t content\_length\_n;

time\_t date\_time;

time\_t last\_modified\_time;

..

} ngx\_http\_headers\_out\_t;

(可以在这里查看结果 [http/ngx\_http\_request.h](http://lxr.evanmiller.org/http/source/http/ngx_http_request.h#L220).)

举例来说, 假设某个模块把头信息的Content-Type 设为 "image/gif", Content-Length为 100, 返回200状态码, 那么就是这个样子:

r->headers\_out.status = NGX\_HTTP\_OK;

r->headers\_out.content\_length\_n = 100;

r->headers\_out.content\_type.len = sizeof("image/gif") - 1;

r->headers\_out.content\_type.data = (u\_char \*) "image/gif";

ngx\_http\_send\_header(r);

大多合法的HTTP 头信息都是有效的. 不过, 有些设置起来就不像上面那么简单了;如,含有(ngx\_table\_elt\_t\*)的content\_encoding, 此时模块还需要给它分配内存.可以通过调用ngx\_list\_push函数来实现, 此函数需要ngx\_list\_t(类似数组)参数会返回一组新创建的成员列表(输入ngx\_table\_elt\_t的成员). 下面的代码就是把Content-Encoding 设为"deflate"并发送头信息:

r->headers\_out.content\_encoding = ngx\_list\_push(&r->headers\_out.headers);

if (r->headers\_out.content\_encoding == NULL) {

return NGX\_ERROR;

}

r->headers\_out.content\_encoding->hash = 1;

r->headers\_out.content\_encoding->key.len = sizeof("Content-Encoding") - 1;

r->headers\_out.content\_encoding->key.data = (u\_char \*) "Content-Encoding";

r->headers\_out.content\_encoding->value.len = sizeof("deflate") - 1;

r->headers\_out.content\_encoding->value.data = (u\_char \*) "deflate";

ngx\_http\_send\_header(r);

上面这种情况一般是在头部信息会同时有多个值的时候; 理论上来说对于过滤模块的添加和删除某些值保留其他值会变得容易些,因为这可以省去对他们排序为字符串的操作.

###### 发送响应体

到这里模块就已经生成好了响应并保存到内存中去了, 只需要把它分配到特定的buffer缓冲里面去就行了, 然后把这个缓冲分配到一个"链"中去,然后调用"链"中的"发送响应体"函数.

上面说的"链"有什么用呢? Nginx让处理模块(还有过滤模块的处理方式)一次性在缓冲里面生成响应信息 ; 在链表中每个元素都会指向下一个指针(最后一个指向的是NULL). 简单起见,我们每次只呈现一个缓冲.

首先, 模块需要申明缓冲和连表:

ngx\_buf\_t \*b;

ngx\_chain\_t out;

接下来就是分配缓冲,并把我们的响应数据指向分配的缓冲:

b = ngx\_pcalloc(r->pool, sizeof(ngx\_buf\_t));

if (b == NULL) {

ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,

"Failed to allocate response buffer.");

return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;

}

b->pos = some\_bytes; /\* first position in memory of the data \*/

b->last = some\_bytes + some\_bytes\_length; /\* last position \*/

b->memory = 1; /\* content is in read-only memory \*/

/\* (i.e., filters should copy it rather than rewrite in place) \*/

b->last\_buf = 1; /\* there will be no more buffers in the request \*/

那么这样模块就把它附到链去了:

out.buf = b;

out.next = NULL;

最后就是, 就是发送响应体了, 返回一次性过滤链所输出的状态码:

return ngx\_http\_output\_filter(r, &out);

缓冲链是Nginx IO模块中很重要的部分,所以你需要注意下它们的工作方式.

细节考虑: 为什么缓冲还有一个"last\_buf"变量?

通过判断"下一个"是否为空,我们怎么知道已经到达链的最后了?

回答: 链有时候可能不是完整的,比如,有很多缓冲的时候并不是所有的缓冲都在请求和响应中的.所以有的缓冲在链的末尾但并没有在请求的尾部,这些就引入了下面的...

##### 解剖upstream(又称代理) 处理

在前面我说了有关处理(handler)生成相应的知识点. 有时候一段C语言也能帮你实现得到相应的目的, 不过你经常需要跟另一台服务器交互 (比如, 你写的是一个实现另一个网络协议的模块). 你可以自己写完所有的网络程序,但要是你收到部分相应怎么办?我想你也不愿意看到主循环还要等待自己循环接收余下响应而受阻的情况吧.幸运的是Nginx支持你对它处理后台服务的机制加上自己的钩子hook(称"上游"), 这样一来你写的模块就可以直接跟另一服务器交互而避免要接收其他响应的弊端. 这部分描述的是模块怎么跟上游(Memcached, FastCGI, 或者其他HTTP服务)交互的.

###### ****upstream回调简介****

上游模块处理函数不像其他模块的处理函数，它很少在实际中工作。它不调用ngx＿http\_output\_filter。当上游服务器准备被编写和读取时，它只是设置可调用的回调函数。可实际上只有六个可用的钩子：

create\_request  探测会被发送到upstream的请求缓存或者缓存链。

reinit＿request 如果连接后端被重置(正如第二次create\_request被调用之前。)

process\_headers 处理upstream响应的第一个字节，通常保持上游有效负载的一个指针。

abort＿requests当客服端终止请求时调用

finalize＿request 当Nginx停止在upstream读取时被调用

input＿filter 它是一个响应体过滤器，可以在响应体中被调用 。

那么这些是怎么实现相互关联的呢?下面是一个代码模块处理的简单版例子:

static ngx\_int\_t

ngx\_http\_proxy\_handler(ngx\_http\_request\_t \*r)

{

ngx\_int\_t rc;

ngx\_http\_upstream\_t \*u;

ngx\_http\_proxy\_loc\_conf\_t \*plcf;

plcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_proxy\_module);

/\* set up our upstream struct \*/

u = ngx\_pcalloc(r->pool, sizeof(ngx\_http\_upstream\_t));

if (u == NULL) {

return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;

}

u->peer.log = r->connection->log;

u->peer.log\_error = NGX\_ERROR\_ERR;

u->output.tag = (ngx\_buf\_tag\_t) &ngx\_http\_proxy\_module;

u->conf = &plcf->upstream;

/\* attach the callback functions \*/

u->create\_request = ngx\_http\_proxy\_create\_request;

u->reinit\_request = ngx\_http\_proxy\_reinit\_request;

u->process\_header = ngx\_http\_proxy\_process\_status\_line;

u->abort\_request = ngx\_http\_proxy\_abort\_request;

u->finalize\_request = ngx\_http\_proxy\_finalize\_request;

r->upstream = u;

rc = ngx\_http\_read\_client\_request\_body(r, ngx\_http\_upstream\_init);

if (rc >= NGX\_HTTP\_SPECIAL\_RESPONSE) {

return rc;

}

return NGX\_DONE;

}

看起来都是很常规的功能,但最重要的部分还是回调函数. 同时还要注意gx\_http\_read\_client\_request\_body这块. 这部分主要是设置当Nginx从客户端完成读取周时的回调函数.

那么每个回调函数都有什么功能呢? reinit\_request,abort\_request, 和finalize\_request 通常是设置或重置内部某些部分的状态,自然代码就简短了. 正儿八经做大事的是create\_request和process\_header.

###### create\_request回调函数

为了简单起见, 假设我现在有一个读一个字符而能输出二个字符的反向代理服务器. 函数怎么写?

create\_request需要给单字符请求分配一个缓冲, 给这个缓冲分配一个链, 再把反向代理结构指向这个链. 就是这个样子了:

static ngx\_int\_t

ngx\_http\_character\_server\_create\_request(ngx\_http\_request\_t \*r)

{

/\* make a buffer and chain \*/

ngx\_buf\_t \*b;

ngx\_chain\_t \*cl;

b = ngx\_create\_temp\_buf(r->pool, sizeof("a") - 1);

if (b == NULL)

return NGX\_ERROR;

cl = ngx\_alloc\_chain\_link(r->pool);

if (cl == NULL)

return NGX\_ERROR;

/\* hook the buffer to the chain \*/

cl->buf = b;

/\* chain to the upstream \*/

r->upstream->request\_bufs = cl;

/\* now write to the buffer \*/

b->pos = "a";

b->last = b->pos + sizeof("a") - 1;

return NGX\_OK;

}

看起来不特别差吧? 当然, 实际中你可能想使用URI , ngx\_str\_t存在于r->uri, r->args可以获得GET参数, 不要忘记你还可以获得请求头部信息和cookie哦.

###### process\_header 回调函数

现在该说说process\_header了. 跟create\_request一样给响应体增加指针,process\_header会自动把响应的指针指向客户端将要接收的那部分去. 它还会读取反向代理的头部信息并相应的设置客户端响应的头信息.

下面有一个小例子, 主要是来读取上面的那些两子杰的响应. 假设第一个字符是有关状态的字节. 如果为问号, 我们想要给客户端返回的是一个404找不到文件状态码,那么就要忽略掉其他的字符.如果是一个空格, 我们想要给客户端返回的是包含其他字符的202状态. 好了, 这还不是最有用的协议, 只是一个很好的例子而已. 那么我们怎么来写process\_header函数呢?

static ngx\_int\_t

ngx\_http\_character\_server\_process\_header(ngx\_http\_request\_t \*r)

{

ngx\_http\_upstream\_t \*u;

u = r->upstream;

/\* read the first character \*/

switch(u->buffer.pos[0]) {

case '?':

r->header\_only; /\* suppress this buffer from the client \*/

u->headers\_in.status\_n = 404;

break;

case ' ':

u->buffer.pos++; /\* move the buffer to point to the next character \*/

u->headers\_in.status\_n = 200;

break;

}

return NGX\_OK;

}

就是这样. 操纵头部, 改变指针就ok了. 你有木有发现headers\_in实际上是一个跟我们在之前说的([http/ngx\_http\_request.h](http://lxr.evanmiller.org/http/source/http/ngx_http_request.h#L158))差不多的头部响应结构, 不过它可以用来自反向代理头部的信息填充而已. 一个真正的代理模块往往比头部处理做的事情要多的多, 不仅仅是提到那些错误处理部分,但重要的是你理解了就行.

但是.. 要是我们没有获取到来自反向代理缓冲的整个头部信息会怎么样呢?

###### 保持状态

好了,还记得我层说过的abort\_request,reinit\_request, 和 finalize\_request是怎么重置内部状态的吗? 那是因为许多反向代理模块都有内部状态. 模块还要定义一个自定义上下文结构来跟踪它从某个反向代理所获取到的东东.这跟之前提到的"模块上下文"是不同的. 毕竟它是预定义的类型,然而自定义上下文需要有你自己需要的元素和数据 (这个要看你自己的结构来定). 自定义上下文结构需要在create\_request函数中初始化, 或许会是这样的:

ngx\_http\_character\_server\_ctx\_t \*p; /\* my custom context struct \*/

p = ngx\_pcalloc(r->pool, sizeof(ngx\_http\_character\_server\_ctx\_t));

if (p == NULL) {

return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;

}

ngx\_http\_set\_ctx(r, p, ngx\_http\_character\_server\_module);

最后一行实际上是用来注册自定义上下文结构,为了方便检索特殊请求和模块名的东西的. 在你要使用自定义上下文结构的时候 (在其他回调函数中), 只需这样调用即可:

ngx\_http\_proxy\_ctx\_t \*p;

p = ngx\_http\_get\_module\_ctx(r, ngx\_http\_proxy\_module);

p将会存储当前的状态. 设置,重置,增加和减少数据...所有你想要的数据, 可以显示任何你想要的数据. 对于持久状态机大量读取反向代理数据来说是很棒的,自然也不会破坏主循环,很酷!

###### 安装handler

通过给回调函数增加启用模块的指令代码来安装处理.比如,我自己写的circle gif模块 ngx\_command\_t是这样:

{ ngx\_string("circle\_gif"),

NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_NOARGS,

ngx\_http\_circle\_gif,

0,

0,

NULL }

第三个参数是回调函数, 在这个例子中ngx\_http\_circle\_gif就是处理. 指令结构(ngx\_conf\_t, 含有用户自己的参数),相应ngx\_command\_t结构和自定义配置的指针指的是本回调函数所回调的参数.对我的circle gif 模块函数就是这样:

static char \*

ngx\_http\_circle\_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\_circle\_gif\_handler;

return NGX\_CONF\_OK;

}

可以分为两步来分析: 第一, 得到本地的核心结构, 然后,把它分配给处理. 相当简单,是吧?

我把自己对模块处理所知道的都写出来了. 下面该说说存在于过滤链中的过滤模块了.

###### handler模块的挂载

两种方式挂载到处理过程，1是按处理阶段挂载，2是按需挂载

按处理阶段挂载（content phase handlers）

为了更精细地控制对于客户端请求的处理过程，nginx把这个处理过程划分成了11个阶段。他们从前到后，依次列举如下：

|  |  |
| --- | --- |
| **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 |

一般情况下，我们自定义的模块，大多数是挂载在NGX\_HTTP\_CONTENT\_PHASE阶段的。挂载的动作一般是在模块上下文调用的postconfiguration函数中。

**注意：有几个阶段是特例，它不调用挂载地任何的handler，也就是你就不用挂载到这几个阶段了：**

* NGX\_HTTP\_FIND\_CONFIG\_PHASE
* NGX\_HTTP\_POST\_ACCESS\_PHASE
* NGX\_HTTP\_POST\_REWRITE\_PHASE
* NGX\_HTTP\_TRY\_FILES\_PHASE

所以其实真正是有7个phase你可以去挂载handler。

挂载的代码如下（摘自hello module）:

static ngx\_int\_t

ngx\_http\_hello\_init(ngx\_conf\_t \*cf)

{

ngx\_http\_handler\_pt \*h;

ngx\_http\_core\_main\_conf\_t \*cmcf;

cmcf = ngx\_http\_conf\_get\_module\_main\_conf(cf, ngx\_http\_core\_module);

h = ngx\_array\_push(&cmcf->phases[NGX\_HTTP\_CONTENT\_PHASE].handlers);

if (h == NULL) {

return NGX\_ERROR;

}

\*h = ngx\_http\_hello\_handler;

return NGX\_OK;

}

使用这种方式挂载的handler也被称为 **content phase handlers**。

当某个location处理到NGX\_HTTP\_CONTENT\_PHASE阶段时，如果有content handler模块，那么NGX\_HTTP\_CONTENT\_PHASE挂载的所有content phase handlers都不会被执行了。

必须在NGX\_HTTP\_CONTENT\_PHASE阶段才能执行到

按需挂载（**content handler）**

当一个请求进来以后，nginx从NGX\_HTTP\_POST\_READ\_PHASE阶段开始依次执行每个阶段中所有handler。执行到 NGX\_HTTP\_CONTENT\_PHASE阶段的时候，如果这个location有一个对应的content handler模块，那么就去执行这个content handler模块真正的处理函数。否则继续依次执行NGX\_HTTP\_CONTENT\_PHASE阶段中所有content phase handlers，直到某个函数处理返回NGX\_OK或者NGX\_ERROR。

例子：

static char \*

ngx\_http\_circle\_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\_circle\_gif\_handler;

return NGX\_CONF\_OK;

}

##### 编写步骤

1. 编写模块基本结构。包括模块的定义，模块上下文结构，模块的配置结构等。
2. 实现handler的挂载函数。根据模块的需求选择正确的挂载方式。
3. 编写handler处理函数。模块的功能主要通过这个函数来完成。

##### Hello handler完整代码分析

下面来完整的给出ngx\_http\_hello\_module模块的完整代码。

#include <ngx\_config.h>

#include <ngx\_core.h>

#include <ngx\_http.h>

//指令配置结构，用来存储用户配置模块行为的配置信息

typedef struct

{

ngx\_str\_t hello\_string;

ngx\_int\_t hello\_counter;

}ngx\_http\_hello\_loc\_conf\_t;

//挂载代码

static ngx\_int\_t ngx\_http\_hello\_init(ngx\_conf\_t \*cf);

//调用该函数创建本模块位于location block的配置信息存储结构。每个在配置中指明的location创建一个。该函数执行成功，返回创建的配置对象。失败的话，返回NULL

static void \*ngx\_http\_hello\_create\_loc\_conf(ngx\_conf\_t \*cf);

//string处理函数

static char \*ngx\_http\_hello\_string(ngx\_conf\_t \*cf, ngx\_command\_t \*cmd,

void \*conf);

//counter处理函数

static char \*ngx\_http\_hello\_counter(ngx\_conf\_t \*cf, ngx\_command\_t \*cmd,

void \*conf);

//模块配置指令

static ngx\_command\_t ngx\_http\_hello\_commands[] = {

{

ngx\_string("hello\_string"),//指令名称

/\*\*可以出现在http server块的location配置指令里|配置指令不接受任何参数。这里的意思是指令可以不填参数，也可以接收1个参数。 \*\*/

NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_NOARGS|NGX\_CONF\_TAKE1,//指令类型

ngx\_http\_hello\_string,//处理方法函数

NGX\_HTTP\_LOC\_CONF\_OFFSET,// 该字段指定当前配置项存储的内存位置

offsetof(ngx\_http\_hello\_loc\_conf\_t, hello\_string),// 指定该配置项值的精确存放位置

NULL },//可以指向任何一个在读取配置过程中需要的数据

{

ngx\_string("hello\_counter"),

NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_FLAG,

ngx\_http\_hello\_counter,

NGX\_HTTP\_LOC\_CONF\_OFFSET,

offsetof(ngx\_http\_hello\_loc\_conf\_t, hello\_counter),

NULL },

ngx\_null\_command

};

/\*

static u\_char ngx\_hello\_default\_string[] = "Default String: Hello, world!";

\*/

static int ngx\_hello\_visited\_times = 0;

//指令上下文结构，这是一组回调函数

static ngx\_http\_module\_t ngx\_http\_hello\_module\_ctx = {

NULL, /\* preconfiguration \*/

ngx\_http\_hello\_init, /\* postconfiguration在创建和读取该模块的配置信息之后被调用。\*/

NULL, /\* create main configuration \*/

NULL, /\* init main configuration \*/

NULL, /\* create server configuration \*/

NULL, /\* merge server configuration \*/

//读取位于location block的配置信息存储结构

ngx\_http\_hello\_create\_loc\_conf, /\* create location configuration \*/

NULL /\* merge location configuration \*/

};

//模块定义，告诉nginx各种配置信息

ngx\_module\_t ngx\_http\_hello\_module = {

NGX\_MODULE\_V1,

&ngx\_http\_hello\_module\_ctx, /\* module context \*/

ngx\_http\_hello\_commands, /\* module directives \*/

NGX\_HTTP\_MODULE, /\* module type \*/

NULL, /\* init master \*/

NULL, /\* init module \*/

NULL, /\* init process \*/

NULL, /\* init thread \*/

NULL, /\* exit thread \*/

NULL, /\* exit process \*/

NULL, /\* exit master \*/

NGX\_MODULE\_V1\_PADDING

};

static ngx\_int\_t

ngx\_http\_hello\_handler(ngx\_http\_request\_t \*r)

{

ngx\_int\_t rc;

ngx\_buf\_t \*b;

ngx\_chain\_t out;

ngx\_http\_hello\_loc\_conf\_t\* my\_conf;

u\_char ngx\_hello\_string[1024] = {0};

ngx\_uint\_t content\_length = 0;

ngx\_log\_error(NGX\_LOG\_EMERG, r->connection->log, 0, "ngx\_http\_hello\_handler is called!");

my\_conf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_hello\_module);

if (my\_conf->hello\_string.len == 0 )

{

ngx\_log\_error(NGX\_LOG\_EMERG, r->connection->log, 0, "hello\_string is empty!");

return NGX\_DECLINED;

}

if (my\_conf->hello\_counter == NGX\_CONF\_UNSET

|| my\_conf->hello\_counter == 0)

{

ngx\_sprintf(ngx\_hello\_string, "%s", my\_conf->hello\_string.data);

}

else

{

ngx\_sprintf(ngx\_hello\_string, "%s Visited Times:%d", my\_conf->hello\_string.data,

++ngx\_hello\_visited\_times);

}

ngx\_log\_error(NGX\_LOG\_EMERG, r->connection->log, 0, "hello\_string:%s", ngx\_hello\_string);

content\_length = ngx\_strlen(ngx\_hello\_string);

/\* we response to 'GET' and 'HEAD' requests only \*/

if (!(r->method & (NGX\_HTTP\_GET|NGX\_HTTP\_HEAD))) {

return NGX\_HTTP\_NOT\_ALLOWED;

}

/\* discard request body, since we don't need it here \*/

rc = ngx\_http\_discard\_request\_body(r);

if (rc != NGX\_OK) {

return rc;

}

/\* set the 'Content-type' header \*/

/\*

\*r->headers\_out.content\_type.len = sizeof("text/html") - 1;

\*r->headers\_out.content\_type.data = (u\_char \*)"text/html";

\*/

ngx\_str\_set(&r->headers\_out.content\_type, "text/html");

/\* send the header only, if the request type is http 'HEAD' \*/

if (r->method == NGX\_HTTP\_HEAD) {

r->headers\_out.status = NGX\_HTTP\_OK;

r->headers\_out.content\_length\_n = content\_length;

return ngx\_http\_send\_header(r);

}

/\* allocate a buffer for your response body \*/

b = ngx\_pcalloc(r->pool, sizeof(ngx\_buf\_t));

if (b == NULL) {

return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;

}

/\* attach this buffer to the buffer chain \*/

out.buf = b;

out.next = NULL;

/\* adjust the pointers of the buffer \*/

b->pos = ngx\_hello\_string;

b->last = ngx\_hello\_string + content\_length;

b->memory = 1; /\* this buffer is in memory \*/

b->last\_buf = 1; /\* this is the last buffer in the buffer chain \*/

/\* set the status line \*/

r->headers\_out.status = NGX\_HTTP\_OK;

r->headers\_out.content\_length\_n = content\_length;

/\* send the headers of your response \*/

rc = ngx\_http\_send\_header(r);

if (rc == NGX\_ERROR || rc > NGX\_OK || r->header\_only) {

return rc;

}

/\* send the buffer chain of your response \*/

return ngx\_http\_output\_filter(r, &out);

}

static void \*ngx\_http\_hello\_create\_loc\_conf(ngx\_conf\_t \*cf)

{

ngx\_http\_hello\_loc\_conf\_t\* local\_conf = NULL;

local\_conf = ngx\_pcalloc(cf->pool, sizeof(ngx\_http\_hello\_loc\_conf\_t));

if (local\_conf == NULL)

{

return NULL;

}

ngx\_str\_null(&local\_conf->hello\_string);

local\_conf->hello\_counter = NGX\_CONF\_UNSET;

return local\_conf;

}

/\*

static char \*ngx\_http\_hello\_merge\_loc\_conf(ngx\_conf\_t \*cf, void \*parent, void \*child)

{

ngx\_http\_hello\_loc\_conf\_t\* prev = parent;

ngx\_http\_hello\_loc\_conf\_t\* conf = child;

ngx\_conf\_merge\_str\_value(conf->hello\_string, prev->hello\_string, ngx\_hello\_default\_string);

ngx\_conf\_merge\_value(conf->hello\_counter, prev->hello\_counter, 0);

return NGX\_CONF\_OK;

}\*/

static char \*

ngx\_http\_hello\_string(ngx\_conf\_t \*cf, ngx\_command\_t \*cmd, void \*conf)

{

ngx\_http\_hello\_loc\_conf\_t\* local\_conf;

local\_conf = conf;

char\* rv = ngx\_conf\_set\_str\_slot(cf, cmd, conf);

ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0, "hello\_string:%s", local\_conf->hello\_string.data);

return rv;

}

//cf:该参数里面保存从配置文件读取到的原始字符串以及相关的一些信息

//cmd 这个配置指令对应的ngx\_command结构 conf 就是定义这个配置值的结构体

static char \*

ngx\_http\_hello\_counter(ngx\_conf\_t \*cf, ngx\_command\_t \*cmd,void \*conf)

{

ngx\_http\_hello\_loc\_conf\_t\* local\_conf;

local\_conf = conf;

char\* rv = NULL;

rv = ngx\_conf\_set\_flag\_slot(cf, cmd, conf);

ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0, "hello\_counter:%d", local\_conf->hello\_counter);

return rv;

}

//挂载代码，按处理阶段挂载，把handler挂载到NGX\_HTTP\_CONTENT\_PHASE阶段，把处理

static ngx\_int\_t

ngx\_http\_hello\_init(ngx\_conf\_t \*cf)

{

ngx\_http\_handler\_pt \*h;

ngx\_http\_core\_main\_conf\_t \*cmcf;

cmcf = ngx\_http\_conf\_get\_module\_main\_conf(cf, ngx\_http\_core\_module);

h = ngx\_array\_push(&cmcf->phases[NGX\_HTTP\_CONTENT\_PHASE].handlers);//获取NGX\_HTTP\_CONTENT\_PHASE的处理模块地址

if (h == NULL) {

return NGX\_ERROR;

}

\*h = ngx\_http\_hello\_handler;//将处理办法设为ngx\_http\_hello\_handler

return NGX\_OK;

}

#### Filters模块

Filter处理生成的响应。Hearder filter操作HTTP头，响应体filter操作响应的内容.

##### Header Filter剖析

头部Filter由三个步骤组成:

1. 决定何时操作响应
2. 操作相应
3. 调用下一个filter

举个例子，比如有一个简化版本的未修改的头部filter：如果客户请求头中的If-Modified-Since和响应头中的Last-Modified相符，它把响应状态设置成304。注意这个头部filter只读入一个参数：ngx\_http\_request\_t结构体，而我们可以通过它操作到客户请求头和一会将被发送的响应消息头

static

ngx\_int\_t ngx\_http\_not\_modified\_header\_filter(ngx\_http\_request\_t \*r)

{

time\_t if\_modified\_since;

if\_modified\_since = ngx\_http\_parse\_time(r->headers\_in.if\_modified\_since->value.data,

r->headers\_in.if\_modified\_since->value.len);

/\* step 1: decide whether to operate \*/

if (if\_modified\_since != NGX\_ERROR &&

if\_modified\_since == r->headers\_out.last\_modified\_time) {

/\* step 2: operate on the header \*/

r->headers\_out.status = NGX\_HTTP\_NOT\_MODIFIED;

r->headers\_out.content\_type.len = 0;

ngx\_http\_clear\_content\_length(r);

ngx\_http\_clear\_accept\_ranges(r);

}

/\* step 3: call the next filter \*/

return ngx\_http\_next\_header\_filter(r);

}

结构headers\_out和我们在hander那一节中看到的是一样的（参考::[http/ngx\_http\_request.h](http://lxr.evanmiller.org/http/source/http/ngx_http_request.h#L220)), 可以无禁止的随意操作.

##### Body Filter剖析

由于主体filter一次只能处理一个缓冲,所以要用缓冲链来写主体filter就有点点麻烦. 模块需要判断是用新分配的缓冲替换和覆盖输入的缓冲,还是在我们讨论的缓冲之前或之后插入新的缓冲. 更复杂的是, 有的时候模块会接收到许多缓冲,这样就造成了有许多待完成的缓冲链. 不幸的是, Nginx 根本就没有提供高等级的API来处理这些缓冲链,造成主体filter很难理解和写入了. 但是,有些操作在实际中你还是能发现的.

一个标准的主体filter应该是这样的 (来源于Nginx代码中的"chunked" filter):

static ngx\_int\_t ngx\_http\_chunked\_body\_filter(ngx\_http\_request\_t \*r, ngx\_chain\_t \*in);

第一个参数是我们熟悉的请求结构. 第二个参数是指向当前链的指针(可能包含0, 1, 或者更多的缓冲).

再来举个简单的例子. 假设我们想把 "<l!-- Served by Nginx -->" 插入到每个请求的最后面去. 首先, 我们要做的就是判断响终端缓冲是否包含在所给出的缓冲链里面,像我之前说的, 没有一个实用的API, 所以还得自己写循环:

ngx\_chain\_t \*chain\_link;

int chain\_contains\_last\_buffer = 0;

chain\_link = in;

for ( ; ; ) {

if (chain\_link->buf->last\_buf)

chain\_contains\_last\_buffer = 1;

if (chain\_link->next == NULL)

break;

chain\_link = chain\_link->next;

}

现在保证在没有最后缓冲的时候就这样:

if (!chain\_contains\_last\_buffer)

return ngx\_http\_next\_body\_filter(r, in);

很好, 现在最后的缓冲就存到chain\_link里面去了. 该分配一个新的缓冲了:

ngx\_buf\_t \*b;

b = ngx\_calloc\_buf(r->pool);

if (b == NULL) {

return NGX\_ERROR;

}

并写入数据:

b->pos = (u\_char \*) "<!-- Served by Nginx -->";

b->last = b->pos + sizeof("<!-- Served by Nginx -->") - 1;

把这个缓冲hook到一个新的链表去:

ngx\_chain\_t \*added\_link;

added\_link = ngx\_alloc\_chain\_link(r->pool);

if (added\_link == NULL)

return NGX\_ERROR;

added\_link->buf = b;

added\_link->next = NULL;

最后,把刚新建的链表hook到我们之前找到的那个终端链表去:

chain\_link->next = added\_link;

并根据实际情况重置"last\_buf"变量:

chain\_link->buf->last\_buf = 0;

added\_link->buf->last\_buf = 1;

把修改后的链传给下一个过滤函数的返回值:

return ngx\_http\_next\_body\_filter(r, in);

作为最后结果的函数往往会比像mod\_perl ($response->body =~ s/$/<!-- Served by mod\_perl -->/)处理起来要麻烦些.但缓冲链确实是一个很有用的结构体,它允许程序员循序渐进地处理数据,这样客户端就能及时获取到新的数据了. 不过, 在我看来, 对于缓冲链来说它极度需要一个更为简洁的接口,这样来能保证程序员不会使链出现前后不一致状态的情况. 目前, 所有的操作都由自己负责了.

##### Filter的安装

在post-configuration那一步的时候Filters其实就安装了.头部filters和主体filters都是在那时同步安装的.

那我们就以chunked filter模块为例来分析下. 本模块的上下文大概是这样的:

static ngx\_http\_module\_t ngx\_http\_chunked\_filter\_module\_ctx = {

NULL, /\* preconfiguration \*/

ngx\_http\_chunked\_filter\_init, /\* postconfiguration \*/

...

};

下面的发生在ngx\_http\_chunked\_filter\_init中的:

static ngx\_int\_t

ngx\_http\_chunked\_filter\_init(ngx\_conf\_t \*cf)

{

ngx\_http\_next\_header\_filter = ngx\_http\_top\_header\_filter;

ngx\_http\_top\_header\_filter = ngx\_http\_chunked\_header\_filter;

ngx\_http\_next\_body\_filter = ngx\_http\_top\_body\_filter;

ngx\_http\_top\_body\_filter = ngx\_http\_chunked\_body\_filter;

return NGX\_OK;

}

这里都发生了什么呢? 嗯,如果你记得的话, filters设置的时候会有一些列的功能的. 当处理生成响应的时候, 就需要调用两个函数:一个是ngx\_http\_output\_filter,主是为了调用引用函数ngx\_http\_top\_body\_filter; 另一个是ngx\_http\_send\_header, 主要是调用引用函数ngx\_top\_header\_filter.

ngx\_http\_top\_body\_filter和ngx\_http\_top\_header\_filter分别是主体的头部和链表的头部. 连表上的每个链都有一个引用函数对应链表中的下一个链(引用分别是gx\_http\_next\_body\_filter和ngx\_http\_next\_header\_filter).当某个filter完成一项执行命令后,就会调用下一个filter, 直到遇到一个定义为"写入"(它包含HTPP相应)的filter的时候. 你在filter\_init函数中所看到的就是模块把自己加到filter链的代码; 在"下"一个变量中它会在旧的"顶部"filter设置一个参照的东东,并把自己的函数变为新的"头部"filter. (如此, 带安装的最后一来链将会第一个被执行.ps:这句话我没想到一个比较好点的翻译,唉)

旁注: 这是怎么运行的呢?

每个filter要么会返回错误.要么使用ngx\_http\_next\_body\_filter()作为返回值:

如此, 要是filter链能运行到那个特殊定义链的最后, 就会返回"ok", 要是, 中途有错误的话链就会终止运行,Nginx会显示响应的错误信息. 它是一个单相的,易返回错误的,只有引用函数的表.很牛X的.

#### 负载均衡模块

负载均衡是决定哪个后端服务器将接收特殊请求的一种方式; 具体实施的是打断循环性的请求或是对请求信息进行hash. 本节会介绍负载均衡的安装和调用,举例使用upstream\_hash模块([全部资源](http://www.evanmiller.org/nginx/ngx_http_upstream_hash_module.c.txt)). upstream\_hash会按照规定hash配置文件nginx.conf中的变量来选择后端服务器.

负载均衡模块有6部分:

1. 授权配置指令(如hash)将要调用注册函数
2. 注册函数会定义规范的服务选项(比如设置高度),并实例化*upstream函数*
3. 配置只要合格,初始化upstream函数就会自动调用的,初始化upstream函数将会:
   * 把服务器名字解析到指定IP
   * 给sockets分配空间
   * 给*初*始化*构造函数设置回调*
4. 每次请求都会调用初始化构造函数, 它主要是设置*负载均衡函数*能使用和操作的数据结构 ;
5. 负载均衡函数会分配请求通道; 客户端每次(要是后端服务请求失败的话不止调用一次的)请求至少会调用一次. 有意思的事情在这里.
6. 最后,对端析构函数在跟某个后端服务器通信(不管失败还是成功)之后会更新统计.

这些有复杂,我会分块来解说的.

##### 授权指令

指令的声明, 回调, 既指定了它们在哪里是合法,又确定了该调用的时候使用什么函数. 加载平衡的指令要有NGX\_HTTP\_UPS\_CONF设置标志, 这样Nginx才会识别该指令只在反向代理代码快内部才是合法的.它应该给注册的函数分配一个指针. 下面是来自upstream\_hash 模块的指令:

{ ngx\_string("hash"),

NGX\_HTTP\_UPS\_CONF|NGX\_CONF\_NOARGS,

ngx\_http\_upstream\_hash,

0,

0,

NULL },

都是很熟悉的知识点.

##### 注册函数

上面说到的ngx\_http\_upstream\_hash函数就是一个注册函数, 我这样命名那是因为它是一个把周边的反向代理配置信息和自己一起注册的反向代理初始化函数. 而且, 在特定的反向代理块(高度,失败时间)里面,这个注册函数会定义哪些选项对于服务器指令是规范的. 下面是upstream\_hash模块的注册函数:

ngx\_http\_upstream\_hash(ngx\_conf\_t \*cf, ngx\_command\_t \*cmd, void \*conf)

{

ngx\_http\_upstream\_srv\_conf\_t \*uscf;

ngx\_http\_script\_compile\_t sc;

ngx\_str\_t \*value;

ngx\_array\_t \*vars\_lengths, \*vars\_values;

value = cf->args->elts;

/\* the following is necessary to evaluate the argument to "hash" as a $variable \*/

ngx\_memzero(&sc, sizeof(ngx\_http\_script\_compile\_t));

vars\_lengths = NULL;

vars\_values = NULL;

sc.cf = cf;

sc.source = &value[1];

sc.lengths = &vars\_lengths;

sc.values = &vars\_values;

sc.complete\_lengths = 1;

sc.complete\_values = 1;

if (ngx\_http\_script\_compile(&sc) != NGX\_OK) {

return NGX\_CONF\_ERROR;

}

/\* end of $variable stuff \*/

uscf = ngx\_http\_conf\_get\_module\_srv\_conf(cf, ngx\_http\_upstream\_module);

/\* the upstream initialization function \*/

uscf->peer.init\_upstream = ngx\_http\_upstream\_init\_hash;

uscf->flags = NGX\_HTTP\_UPSTREAM\_CREATE;

/\* OK, more $variable stuff \*/

uscf->values = vars\_values->elts;

uscf->lengths = vars\_lengths->elts;

/\* set a default value for "hash\_method" \*/

if (uscf->hash\_function == NULL) {

uscf->hash\_function = ngx\_hash\_key;

}

return NGX\_CONF\_OK;

}

除跳过以外唯命是从外,我们还可以稍后计算$variable, 真的相当直观; 分配一个回调函数, 设置标记. 有写什么标记呢?

* NGX\_HTTP\_UPSTREAM\_CREATE: 反向代理中有server指令. 我实在想不出什么场合你不会使用.
* NGX\_HTTP\_UPSTREAM\_WEIGHT: 让服务指令获取weight=选项
* NGX\_HTTP\_UPSTREAM\_MAX\_FAILS: 允许max\_fail选项
* NGX\_HTTP\_UPSTREAM\_FAIL\_TIMEOUT: 允许fail\_timeout选项
* NGX\_HTTP\_UPSTREAM\_DOWN: 允许down 选项
* NGX\_HTTP\_UPSTREAM\_BACKUP:允许backup选项

每个模块都能访问获取这些配置的. 一切取决于模块怎么用了*.* 也就是说,max\_fails不会自动执行; 所以的错误逻辑都是模块的作者来决定的. 更多的信息留在以后说. 现在, 我们仍还没有完成跟踪回调. 接下来, 我们来看下反向代理初始化函数 (就是之前说的init\_upstream中的回调).

##### 反向代理初始化函数

反向代理初始化函数的主要功能是解析主机,给socket分配内存, 分配 (另一个分配功能) 回调函数. 下面是upstream\_hash具体情况:

ngx\_int\_t

ngx\_http\_upstream\_init\_hash(ngx\_conf\_t \*cf, ngx\_http\_upstream\_srv\_conf\_t \*us)

{

ngx\_uint\_t i, j, n;

ngx\_http\_upstream\_server\_t \*server;

ngx\_http\_upstream\_hash\_peers\_t \*peers;

/\* set the callback \*/

us->peer.init = ngx\_http\_upstream\_init\_upstream\_hash\_peer;

if (!us->servers) {

return NGX\_ERROR;

}

server = us->servers->elts;

/\* figure out how many IP addresses are in this upstream block. \*/

/\* remember a domain name can resolve to multiple IP addresses. \*/

for (n = 0, i = 0; i < us->servers->nelts; i++) {

n += server[i].naddrs;

}

/\* allocate space for sockets, etc \*/

peers = ngx\_pcalloc(cf->pool, sizeof(ngx\_http\_upstream\_hash\_peers\_t)

+ sizeof(ngx\_peer\_addr\_t) \* (n - 1));

if (peers == NULL) {

return NGX\_ERROR;

}

peers->number = n;

/\* one port/IP address per peer \*/

for (n = 0, i = 0; i < us->servers->nelts; i++) {

for (j = 0; j < server[i].naddrs; j++, n++) {

peers->peer[n].sockaddr = server[i].addrs[j].sockaddr;

peers->peer[n].socklen = server[i].addrs[j].socklen;

peers->peer[n].name = server[i].addrs[j].name;

}

}

/\* save a pointer to our peers for later \*/

us->peer.data = peers;

return NGX\_OK;

}

这个函数比你期望的难些. 大多数的功能ms都有点抽象, 但实际不是这样的, 那才是我们能接收它的原因. 倒是有个简单的策略可以解决,那就是调用另一个模块的反向代理初始化函数, 让它来完成这些冗杂的功能(监听分配什么的), 然后覆盖us->peer.init回调即可.看个例子:http/modules/ngx\_http\_upstream\_ip\_hash\_module.c.

在我们观点中最重要的一点就是给监听初始化函数设置一个指针, 也就像在这个例子中的http\_upstream\_init\_upstream\_hash\_peer.

##### 对端初始化

每次请求都会调用对端初始化函数.它的功能主要是构造一组数据结构供模块在尝试找适合能服务于该请求所需要的后端服务器时使用; 这个结构会一值保存从后端服务器上通信的重复记录,所以通过它可以方便跟踪链接失败的次数, 或者是计算好的hash值. 按照惯例, 这个结构称为ngx\_http\_upstream\_<模块名>\_peer\_data\_t.

此外, 它还会生成两个回调函数:

* get: 加载均衡方法
* free: 对端析构函数 (通常是更新一些通信结束后的统计)

好像那些还不够,它还会初始化一个名叫tries的变量. 只要tries是正数, nginx就会一直重试这个平衡加载器. 当tries是0的时候, nginx就会终止. 这一切都要看get和free这两个函数是怎么设置tries的了.

下面是来自upstream\_hash的对端初始化函数:

static ngx\_int\_t

ngx\_http\_upstream\_init\_hash\_peer(ngx\_http\_request\_t \*r,

ngx\_http\_upstream\_srv\_conf\_t \*us)

{

ngx\_http\_upstream\_hash\_peer\_data\_t \*uhpd;

ngx\_str\_t val;

/\* evaluate the argument to "hash" \*/

if (ngx\_http\_script\_run(r, &val, us->lengths, 0, us->values) == NULL) {

return NGX\_ERROR;

}

/\* data persistent through the request \*/

uhpd = ngx\_pcalloc(r->pool, sizeof(ngx\_http\_upstream\_hash\_peer\_data\_t)

+ sizeof(uintptr\_t)

\* ((ngx\_http\_upstream\_hash\_peers\_t \*)us->peer.data)->number

/ (8 \* sizeof(uintptr\_t)));

if (uhpd == NULL) {

return NGX\_ERROR;

}

/\* save our struct for later \*/

r->upstream->peer.data = uhpd;

uhpd->peers = us->peer.data;

/\* set the callbacks and initialize "tries" to "hash\_again" + 1\*/

r->upstream->peer.free = ngx\_http\_upstream\_free\_hash\_peer;

r->upstream->peer.get = ngx\_http\_upstream\_get\_hash\_peer;

r->upstream->peer.tries = us->retries + 1;

/\* do the hash and save the result \*/

uhpd->hash = us->hash\_function(val.data, val.len);

return NGX\_OK;

}

也没有想象的那么糟糕. 现在我们就该选择反向代理服务器了.

##### 负载均衡函数

该上主角了. 这才是重点. 也是模块选择负载均衡的地方.标准的函数是:

static ngx\_int\_t

ngx\_http\_upstream\_get\_<module\_name>\_peer(ngx\_peer\_connection\_t \*pc, void \*data);

上面的data就是我们关注客户交互的有用信息结构体. pc会获取我们将要链接的服务器信息. 负载均衡函数的功能是传forpc->sockaddr,pc->socklen, andpc->name等数据. 那是你有网络编程知识的话, 这些变量你可能见过; 但实际上跟我们手边的工作比起来还是没那么重要.我们并不需要知道它们代表什么; 我们只需要知道在哪里去给它们找到合适的数据就行了.

本函数需要找到一些列可用的服务, 选择其中的一个, 把值分配给pc. 看下upstream\_hash 是怎么实现的.

upstream\_hash会预先把服务列表存到ngx\_http\_upstream\_hash\_peer\_data\_t回到之前ngx\_http\_upstream\_init\_hash结果中.现在结构体就会以data的形式获取了:

ngx\_http\_upstream\_hash\_peer\_data\_t \*uhpd = data;

对端列表就存在uhpd->peers->peer中去了. 下面我就从对端数组中取出一个元素,按照服务数把这个计算好的hash值分割出来:

ngx\_peer\_addr\_t \*peer = &uhpd->peers->peer[uhpd->hash % uhpd->peers->number];

看下结果:

pc->sockaddr = peer->sockaddr;

pc->socklen = peer->socklen;

pc->name = &peer->name;

return NGX\_OK;

就这样了! 要是load-balancer返回NGX\_OK的话,那就意味着, "干吧,再试". 要是返回NGX\_BUSY的话,那就意味着后端服务不可用, Nginx 必须再次尝试.

但是… 怎么跟踪那些到底是什么不可用呢? 要是我们不再次尝试的话怎样呢?

##### 对端析构函数

当有反向代理链接发生后对端析构函数就会执行; 主要是为了跟踪失败记录下面是函数原型:

void

ngx\_http\_upstream\_free\_<module name>\_peer(ngx\_peer\_connection\_t \*pc, void \*data,

ngx\_uint\_t state);

前两个函数跟我们之前看到的load-balancer函数中是一样的. 第三个参数是一个反映连接是否成功的状态变量. 它可能包含的是NGX\_PEER\_FAILED(连接失败的) 和NGX\_PEER\_NEXT(不管是连接失败还是成功应用都有返回值)两个位操作的或值. 0就是成功.

这都取决于模块开发者想怎么处理这些失败的事件. 如果真要用的话就会把结果存在data里面去,是一个指向每个自定义请求数据的结构体.

不过本函数最重要的目的是如果在请求中想终止Nginx保持尝试请求load-balancer的话就把pc->tries这只为0. 所以最简单可以是这样写:

pc->tries = 0;

这样就能保证要是后端有错误的话就能给客户端返回一个502代理错误.

### 编写并编译一个新的 Nginx 模块

现在，您应该准备好去查看一个 Nginx 模块，并尝试去理解发生了什么事(当然您知道可以在哪里寻求帮助）。在 [src/http/modules/](http://lxr.evanmiller.org/http/source/http/modules/) 上查看可用的模块。挑选一个一个和您试图去完成的模块相似的模块并仔细查看它。似曾相识的东西？它应当是。参阅这份指南和模块源代码以便理解发生了什么。

但是 Emiller 并没有写一本 阅读 Nginx 模块的进球指南(Balls-In Guide to Reading Nginx Modules)。 该死的。这是一份 出球指南(Balls-Out Guide)。我们没有在阅读。我们在写作。在创造。在与世界分享。

首先，你需要一个放置你的模块的地方。在硬盘的任何地方为你的模块建立一个文件夹，但要和 Nginx 源代码（请确保有来自于 nginx.net 的最新副本）分开。文件夹中应当包含两个以下列字符串开头的文件：

"config"

"ngx\_http\_<your module>\_module.c"

"config" 文件将被包含在 ./configure 中，它的内容依赖于模块的类型。

filter 模块的 "config":

ngx\_addon\_name=ngx\_http\_<your module>\_module

HTTP\_AUX\_FILTER\_MODULES="$HTTP\_AUX\_FILTER\_MODULES ngx\_http\_<your module>\_module"

NGX\_ADDON\_SRCS="$NGX\_ADDON\_SRCS $ngx\_addon\_dir/ngx\_http\_<your module>\_module.c"

其他模块的 "config" :

ngx\_addon\_name=ngx\_http\_<your module>\_module

HTTP\_MODULES="$HTTP\_MODULES ngx\_http\_<your module>\_module"

NGX\_ADDON\_SRCS="$NGX\_ADDON\_SRCS $ngx\_addon\_dir/ngx\_http\_<your module>\_module.c"

现在就到 C 文件了。我建议拷贝一个和你想做的模块要做的事类似的已有模块，但以 "ngx\_http\_<your module>\_module.c" 重命名它。像这样让它成为你的模块，就像你改变行为适应自己的需要一样，当要理解和重塑不同的部分时，可参考这篇指南。

当您准备好编译时，仅需进入 Nginx 目录键入

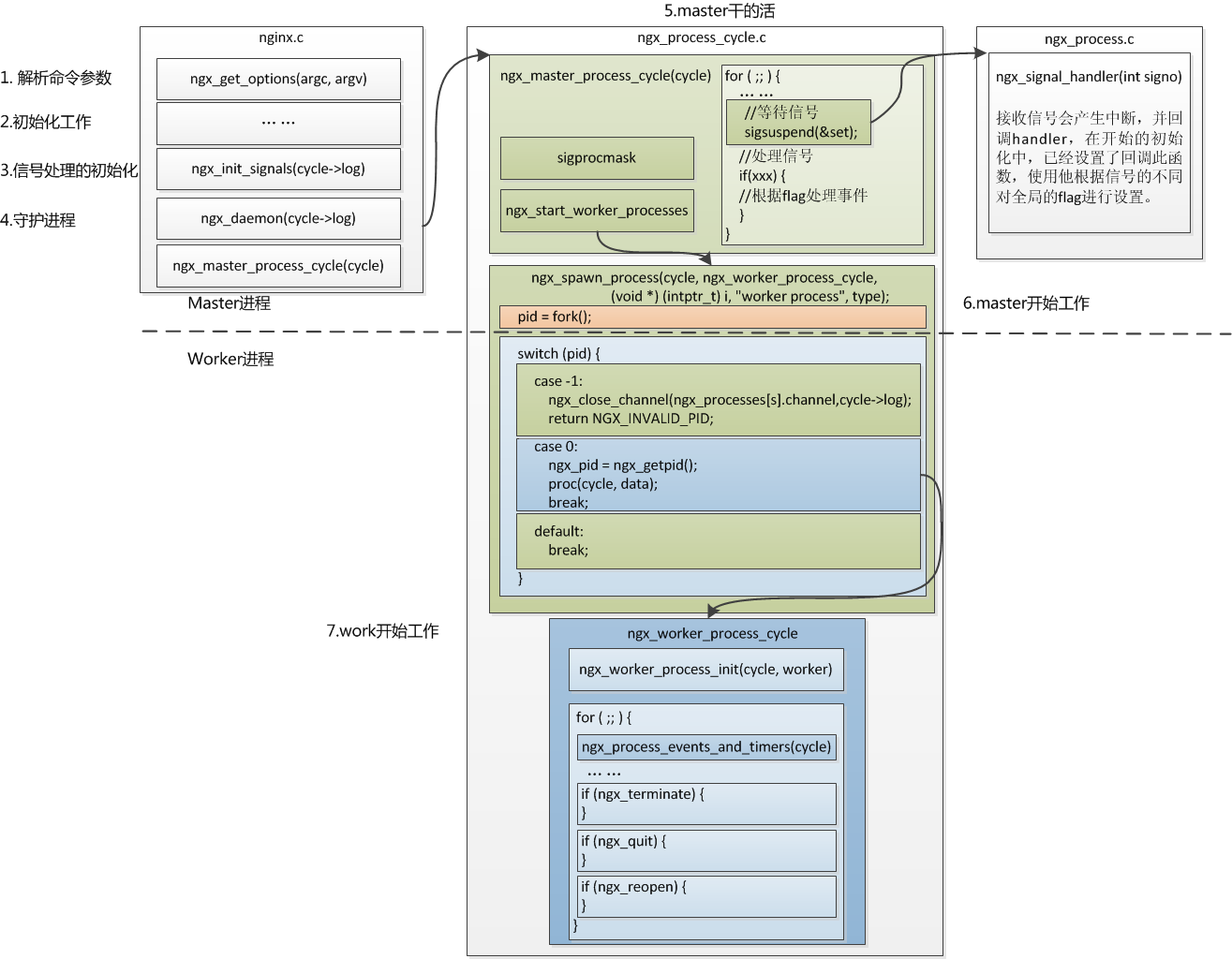
./configure --add-module=path/to/your/new/module/directory

接着和平常一样 make make install。如果一切都好，您的模块将正好编译完成。很漂亮，对吧？没必要去弄糟 Nginx 源码，同时将您的模块加到一个新版本的 Nginx 上是非常容易的，仅需使用相同的 ./configure 命令。顺便说一下，如果您的模块需要某些动态链接库，您可以将这个添加到您的 "config" 文件：

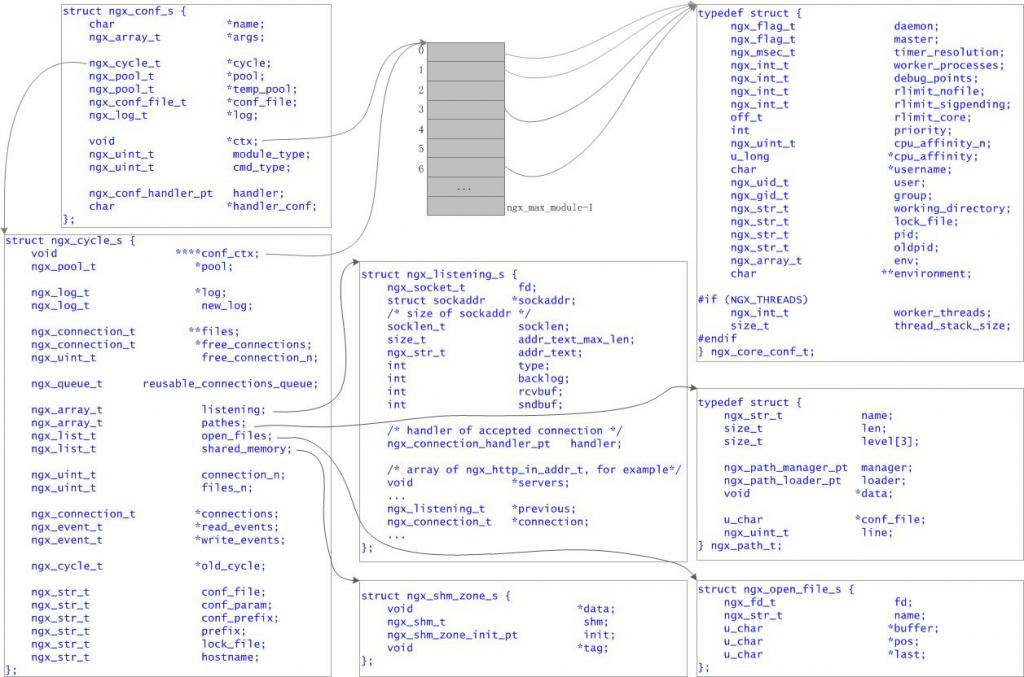
CORE\_LIBS="$CORE\_LIBS -lfoo"

这里 foo 是您需要的库。如果您创建了一个酷的或者有用的模块，请确保发送一条记录到 Nginx 邮件列表 同时分享您的工作。

### Nginx启动流程分析



#### 各种结构体之间关系图



#### Main()流程分析

nginx启动过程如下。

调用ngx\_get\_options()解析命令参数；

调用ngx\_time\_init()初始化并更新时间，如全局变量ngx\_cached\_time；

调用ngx\_log\_init()初始化日志，如初始化全局变量ngx\_prefix，打开日志文件ngx\_log\_file.fd；

清零全局变量ngx\_cycle，并为ngx\_cycle.pool创建大小为1024B的内存池；

调用ngx\_save\_argv()保存命令行参数至全局变量ngx\_os\_argv、ngx\_argc、ngx\_argv中；

调用ngx\_process\_options()初始化ngx\_cycle的prefix, conf\_prefix, conf\_file, conf\_param等字段；

调用ngx\_os\_init()初始化系统相关变量，如内存页面大小ngx\_pagesize,ngx\_cacheline\_size,最大连接数ngx\_max\_sockets等；

调用ngx\_crc32\_table\_init()初始化CRC表(后续的CRC校验通过查表进行，效率高)；

调用ngx\_add\_inherited\_sockets()继承sockets；

解析环境变量NGINX\_VAR="NGINX"中的sockets，并保存至ngx\_cycle.listening数组；

设置ngx\_inherited=1；

调用ngx\_set\_inherited\_sockets()逐一对ngx\_cycle.listening数组中的sockets进行设置；

具体可参考<nginx源码分析—初始化过程中处理继承的sockets>

初始化每个module的index，并计算ngx\_max\_module；

具体可参考[<nginx源码分析—模块及其初始化>](http://blog.csdn.net/livelylittlefish/article/details/6571497)；

调用ngx\_init\_cycle()进行初始化；

该初始化主要对ngx\_cycle结构进行；

具体可参考<nginx源码分析—全局变量ngx\_cycle的初始化>；

若有信号，则进入ngx\_signal\_process()处理；

调用ngx\_init\_signals()初始化信号；主要完成信号处理程序的注册；

若无继承sockets，且设置了守护进程标识，则调用ngx\_daemon()创建守护进程；

调用ngx\_create\_pidfile()创建进程记录文件；(非NGX\_PROCESS\_MASTER=1进程，不创建该文件)

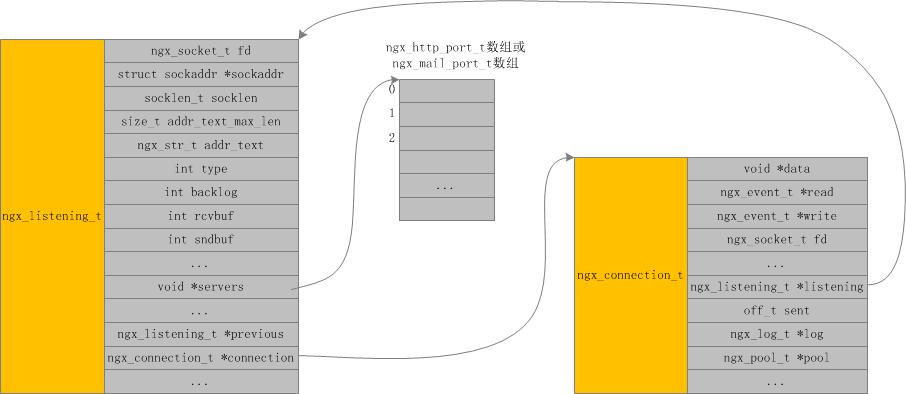
进入进程主循环；

若为NGX\_PROCESS\_SINGLE=1模式，则调用ngx\_single\_process\_cycle()进入进程循环；

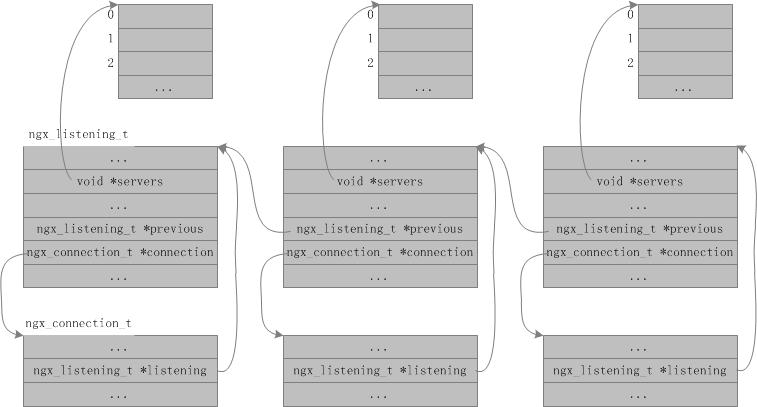
否则为master-worker模式，调用ngx\_master\_process\_cycle()进入进程循环；

具体可参考<nginx源码分析—master/worker进程启动>；

#### 处理继承的Socket



ngx\_listening\_t与ngx\_connection\_t之间的关系如下图。



监听的结构

1. **struct** ngx\_listening\_s {
2. ngx\_socket\_t        fd;         /\* 文件描述符,即socket \*/
4. **struct** sockaddr    \*sockaddr;   /\* socket地址 \*/
5. socklen\_t           socklen;    /\* size of sockaddr \*/
6. **size\_t**              addr\_text\_max\_len;
7. ngx\_str\_t           addr\_text;
9. **int**                 type;
11. **int**                 backlog;
12. **int**                 rcvbuf;     /\* 接收缓冲区 \*/
13. **int**                 sndbuf;     /\* 发送缓冲区 \*/
15. /\* handler of accepted connection \*/
16. ngx\_connection\_handler\_pt   handler;
18. **void**               \*servers;  /\* array of ngx\_http\_in\_addr\_t, for example \*/
20. ngx\_log\_t           log;
21. ngx\_log\_t          \*logp;
23. **size\_t**              pool\_size;
24. /\* should be here because of the AcceptEx() preread \*/
25. **size\_t**              post\_accept\_buffer\_size;
26. /\* should be here because of the deferred accept \*/
27. ngx\_msec\_t          post\_accept\_timeout;
29. ngx\_listening\_t    \*previous;   /\* 指向前一个ngx\_listening\_t结构 \*/
30. ngx\_connection\_t   \*connection;
32. unsigned            open:1;
33. unsigned            remain:1;
34. unsigned            ignore:1;
36. unsigned            bound:1;       /\* already bound \*/
37. unsigned            inherited:1;   /\* inherited from previous process \*/
38. unsigned            nonblocking\_accept:1;
39. unsigned            listen:1;
40. unsigned            nonblocking:1;
41. unsigned            shared:1;    /\* shared between threads or processes \*/
42. unsigned            addr\_ntop:1;
44. #if (NGX\_HAVE\_INET6 && defined IPV6\_V6ONLY)
45. unsigned            ipv6only:2;
46. #endif
48. #if (NGX\_HAVE\_DEFERRED\_ACCEPT)
49. unsigned            deferred\_accept:1;
50. unsigned            delete\_deferred:1;
51. unsigned            add\_deferred:1;
52. #ifdef SO\_ACCEPTFILTER
53. **char**               \*accept\_filter;
54. #endif
55. #endif
56. #if (NGX\_HAVE\_SETFIB)
57. **int**                 setfib;
58. #endif
60. };
61. /\* 传入该参数的是init\_cycle,调用ngx\_init\_cycle()后全局变量ngx\_cycle会指向该结构 \*/
62. **static** ngx\_int\_t
63. ngx\_add\_inherited\_sockets(ngx\_cycle\_t \*cycle)
64. {
65. u\_char           \*p, \*v, \*inherited;
66. ngx\_int\_t         s;
67. ngx\_listening\_t  \*ls;
69. inherited = (u\_char \*) getenv(NGINX\_VAR);  /\* NGINX\_VAR为宏，值为"NGINX" \*/
71. **if** (inherited == NULL) {
72. **return** NGX\_OK;
73. }
75. ngx\_log\_error(NGX\_LOG\_NOTICE, cycle->log, 0,
76. "using inherited sockets from \"%s\"", inherited);
78. **if** (ngx\_array\_init(&cycle->listening, cycle->pool, 10,
79. **sizeof**(ngx\_listening\_t))
80. != NGX\_OK)  /\* 初始化ngx\_cycle.listening数组, 10个元素空间 \*/
81. {
82. **return** NGX\_ERROR;
83. }
85. **for** (p = inherited, v = p; \*p; p++) {
86. **if** (\*p == ':' || \*p == ';') {  /\* sockets以':'或者';'隔开 \*/
87. s = ngx\_atoi(v, p - v);    /\* sockets是10进制正整数 \*/
88. **if** (s == NGX\_ERROR) {
89. ngx\_log\_error(NGX\_LOG\_EMERG, cycle->log, 0,
90. "invalid socket number \"%s\" in " NGINX\_VAR
91. " environment variable, ignoring the rest"
92. " of the variable", v);
93. **break**;
94. }
96. /\* 第一次满足该条件时v和p分别指向如下值，此时解析出s=16000
97. (gdb) p v
98. $6 = (u\_char \*) 0x6b0139 "16000:16500:16600;"
99. (gdb) p p
100. $7 = (u\_char \*) 0x6b013e ":16500:16600;"
101. \*/
102. v = p + 1;
104. ls = ngx\_array\_push(&cycle->listening);  /\* 将合法的socket number加入该数组\*/
105. **if** (ls == NULL) {
106. **return** NGX\_ERROR;
107. }
109. ngx\_memzero(ls, **sizeof**(ngx\_listening\_t));
111. ls->fd = (ngx\_socket\_t) s;  /\* 保存该socket至listening数组元素的fd字段 \*/
112. }
113. }
115. ngx\_inherited = 1;
117. **return** ngx\_set\_inherited\_sockets(cycle);  /\* 该函数逐一设置cycle->listening数组每个元素 \*/
118. }

**ngx\_set\_inherited\_sockets()分析**

该函数从参数cycle(后续调用ngx\_init\_cycle()函数后全局变量ngx\_cycle会指向该参数)的listening数组中逐一对每个元素(ngx\_listening\_t结构)进行初始化，即初始化除fd字段外的其他的字段。

因此，ngx\_set\_inherited\_sockets()函数主要完成以下事情。

对全局变量ngx\_cycle的listening数组，逐一设置该数组每个元素的以下字段

* ls[i].sockaddr (调用getsockname())
* ls[i].addr\_text\_max\_len
* ls[i].addr\_text
* ls[i].backlog
* ls[i].rcvbuf (调用getsockopt())
* ls[i].sndbuf (调用getsockopt())
* ls[i].accept\_filter
* ls[i].deferred\_accept

涉及到的相关系统函数调用如下。

* getenv()获取环境变量，并返回指向该值字符串的指针；
* getsockname()获取socket名字(地址及长度)；
* getsockopt()获取socket选项；

#### 全局变量ngx\_cycle的初始化

**ngx\_cycle\_t结构**

该结构在./src/core/ngx\_cycle.h文件中定义，如下。

**[cpp]** [view plain](http://blog.csdn.net/livelylittlefish/article/details/7247080) [copy](http://blog.csdn.net/livelylittlefish/article/details/7247080)

1. **struct** ngx\_cycle\_s {
2. **void**                  \*\*\*\*conf\_ctx;  //配置上下文数组(含所有模块)
3. ngx\_pool\_t               \*pool;      //内存池
5. ngx\_log\_t                \*log;       //日志
6. ngx\_log\_t                 new\_log;
8. ngx\_connection\_t        \*\*files;     //连接文件
9. ngx\_connection\_t         \*free\_connections;  //空闲连接
10. ngx\_uint\_t                free\_connection\_n; //空闲连接个数
12. ngx\_queue\_t               reusable\_connections\_queue;  //再利用连接队列
14. ngx\_array\_t               listening;     //监听数组
15. ngx\_array\_t               pathes;        //路径数组
16. ngx\_list\_t                open\_files;    //打开文件链表
17. ngx\_list\_t                shared\_memory; //共享内存链表
19. ngx\_uint\_t                connection\_n;  //连接个数
20. ngx\_uint\_t                files\_n;       //打开文件个数
22. ngx\_connection\_t         \*connections;   //连接
23. ngx\_event\_t              \*read\_events;   //读事件
24. ngx\_event\_t              \*write\_events;  //写事件
26. ngx\_cycle\_t              \*old\_cycle;     //old cycle指针
28. ngx\_str\_t                 conf\_file;     //配置文件
29. ngx\_str\_t                 conf\_param;    //配置参数
30. ngx\_str\_t                 conf\_prefix;   //配置前缀
31. ngx\_str\_t                 prefix;        //前缀
32. ngx\_str\_t                 lock\_file;     //锁文件
33. ngx\_str\_t                 hostname;      //主机名
34. };

该结构体的大小是确定的，sizeof(ngx\_cycle\_t)=224。

其中，

pathes数组元素结构为ngx\_path\_t；

open\_files链表元素结构为ngx\_open\_file\_t；

shared\_memory链表元素结构为ngx\_shm\_zone\_t；

listening数组元素结构为ngx\_listening\_t，该数组用来存放监听套接字。

**ngx\_init\_cycle()分析**

初始化过程如下。

* 调用ngx\_timezone\_update()更新时区，调用ngx\_time\_update()更新时间
* 创建大小为NGX\_CYCLE\_POOL\_SIZE=16384B的内存池，并从中分配ngx\_cycle\_t结构
* 简单初始化，如记录pool指针、log指针
* 初始化配置前缀、前缀、配置文件、配置参数等字符串
* 初始化pathes数组
* 初始化open\_files链表
* 初始化shared\_memory链表
* 初始化listening数组
* 初始化resuable\_connections\_queue队列
* 从pool为conf\_ctx分配空间
* 初始化hostname字符串
* 调用core模块的create\_conf()
* 配置文件解析
* 调用core模块的init\_conf()
* 遍历open\_files链表中的每一个文件并打开
* 创建共享内存并初始化(新旧shared\_memory链表的比较，相同的共享内存保留，旧的不同的共享内存被释放，新的被创建)
* (尝试5遍)遍历listening数组并打开所有侦听sockets(socket()->setsockopt()->bind()->listen())
* 提交新的cycle配置，并调用所有模块的init\_module(实际上只有ngx\_event\_core\_module模块定义了该callback，即只有ngx\_event\_module\_init()被调用)
* 关闭或删除残留在old\_cycle中的资源
  + 释放多余的共享内存
  + 关闭多余的侦听sockets
  + 关闭多余的打开文件

具体请参考附录代码，不需要细究每一步实现，重要的是要搞清楚其初始化流程。

#### [ngx\_master\_process\_cycle 多进程](http://blog.csdn.net/lengzijian/article/details/7587740)

进程启动的过程中，有一些重要的全局数据会被设置，最重要的是进程表ngx\_processes，master每创建一个worker都会把一个设置好的ngx\_process\_t结构变量放入ngx\_processes中，进程表长度为1024，刚创建的进程存放在ngx\_process\_slot位置，ngx\_last\_process是进程表中最后一个存量进程的下一个位置，ngx\_process\_t是进程在nginx中的抽象：

1. **typedef** **void** (\*ngx\_spawn\_proc\_pt) (ngx\_cycle\_t \*cycle, **void** \*data);
3. **typedef** **struct** {
4. ngx\_pid\_t           pid;                 //进程id
5. **int**                 status;              //进程状态
6. ngx\_socket\_t        channel[2];          //socketpair创建的socket句柄
8. ngx\_spawn\_proc\_pt   proc;                //进程执行函数
9. **void**               \*data;                //执行函数的参数
10. **char**               \*name;                //名称
12. unsigned            respawn:1;           //重新创建
13. unsigned            just\_spawn:1;        //第一次创建的
14. unsigned            detached:1;          //分离的
15. unsigned            exiting:1;           //正在退出的
16. unsigned            exited:1;            //退出过的
17. } ngx\_process\_t;

进程之间是通过socketpair进行通信的，socketpair是一对全双工的无名socket

1. ypedef **struct** {
2. ngx\_uint\_t  command;
3. ngx\_pid\_t   pid;                        //发送方进程id
4. ngx\_int\_t   slot;                       //发送方进程表中偏移
5. ngx\_fd\_t    fd;                         //发送给对方的句柄
6. } ngx\_channel\_t;(src/os/unix/ngx\_channel.h)
8. /\*command是要发送的命令，有5种：
10. #define NGX\_CMD\_OPEN\_CHANNEL   1
11. #define NGX\_CMD\_CLOSE\_CHANNEL  2
12. #define NGX\_CMD\_QUIT           3
13. #define NGX\_CMD\_TERMINATE      4
14. #define NGX\_CMD\_REOPEN         5
15. \*/

摘自：<http://blog.csdn.net/lengzijian/article/details/7587740>

**ngx\_master\_process\_cycle()**函数，这个函数会启动工作进程干活，并且会处理信号量，处理的过程中会杀死或者创建新的进程。

a)  阻塞所有nginx关心的信号；

b)  设置进程的title（如果你用ps –aux来查看就可以分清master与worker进程，这就是title的作用。）；

c)  按照ngx\_core\_conf\_t中worker\_processes数，启动若干个work进程；

d)  启动一个缓冲管理进程；

e)  初始化几个标志：ngx\_new\_binary = 0; delay = 0; live = 1;

f)  后面一个循环对不同的状态进行不同处理，而那些状态多数是进程收到的不同信号。下面是各个处理的详解：

a)   delay不为0，如果收到SIGALRM信号ngx\_sigalrm设为1，将delay时间乘以2；最后设置一个实时类型的计时器；

b)   挂起当前进程，等到有信号，就会从挂起状态退出，继续执行；

c)   退出挂起状态后，根据操作系统时间重新更新当前时间；

d)   ngx\_reap为1(收到SIGCHLD信号**，**有worker退出(ngx\_reap==1))，调用ngx\_reap\_children()回收子进程；

e)   如果子进程都退出了(!live)且当前进程收到ngx\_signal\_value(NGX\_SHUTDOWN\_SIGNAL)或ngx\_signal\_value(NGX\_TERMINATE\_SIGNAL)信号，本进程进行退出处理(ngx\_master\_process\_exit())；退出处理先删除pid文件，然后将调用所有模块的进程退出钩子，销毁内存池对象；

f)   如果ngx\_terminate为1，delay为0，就设成50；如果delay>1000，向work进程发送SIGKILL信号，否则向work进程发送ngx\_signal\_value(NGX\_TERMINATE\_SIGNAL)信号；

g)   如果ngx\_quit为1，向work进程发送ngx\_signal\_value(NGX\_SHUTDOWN\_SIGNAL)信号，然后将所有全局listening中的socket全关闭；continue；

h)   如果ngx\_reconfigure为1(ngx\_signal\_value(NGX\_RECONFIGURE\_SIGNAL)信号对应)，就重新读取config文件；重新创建并初始化ngx\_cycle对象，启动work进程，启动缓冲管理进程，将live设为1，调用ngx\_signal\_worker\_processes()发送ngx\_signal\_value(NGX\_SHUTDOWN\_SIGNAL)信号；

i)   ngx\_new\_binary为1(表示是新启动的一个进程)，启动work进程，启动缓冲管理进程，然后将ngx\_noaccepting设为0；continue；

j)   如果ngx\_restart为1(当ngx\_noaccepting=1的时候会把ngx\_restart设为1，重启worker)，启动work进程，启动缓冲管理进程，live设为1；

k)   如果ngx\_reopen为1(ngx\_signal\_value(NGX\_REOPEN\_SIGNAL)信号对应)，则重新找开log文件，调用ngx\_signal\_worker\_processes()发送ngx\_signal\_value(NGX\_REOPEN\_SIGNAL)信号；

l)   如果ngx\_change\_binary为1(ngx\_signal\_value(NGX\_CHANGEBIN\_SIGNAL)信号对应)，调用ngx\_exec\_new\_binary()执行新进程；

m)   如果ngx\_noaccept为1(ngx\_signal\_value(NGX\_NOACCEPT\_SIGNAL)对应)，设ngx\_noaccepting为1，调用ngx\_signal\_worker\_processes()发送ngx\_signal\_value(NGX\_SHUTDOWN\_SIGNAL)信号。

#### [ngx\_start\_worker\_processes子进程创建](http://blog.csdn.net/lengzijian/article/details/7589998)

真正创建worker子进程的函数是ngx\_start\_worker\_processes，这个函数本身很简单：

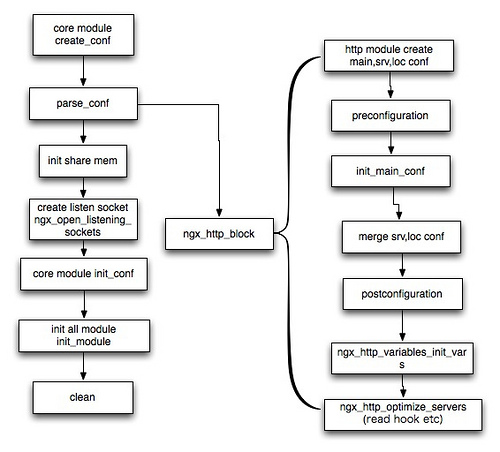
1. src/os/unix/ngx\_process\_cycle.c
3. **static** **void**
4. ngx\_start\_worker\_processes(ngx\_cycle\_t \*cycle, ngx\_int\_t n, ngx\_int\_t type)
5. {
6. ngx\_int\_t      i;
7. ngx\_channel\_t  ch;
9. ngx\_log\_error(NGX\_LOG\_NOTICE, cycle->log, 0, "start worker processes");
10. //传递给其他worker子进程的命令，打开通信管道
11. ch.command = NGX\_CMD\_OPEN\_CHANNEL;
12. //创建n个worker子进程
13. **for** (i = 0; i < n; i++) {
15. cpu\_affinity = ngx\_get\_cpu\_affinity(i);
17. //ngx\_spawn\_process创建worker子进程并初始化相关资源和属性
18. //然后执行子进程的执行函数ngx\_worker\_process\_cycle
19. ngx\_spawn\_process(cycle, ngx\_worker\_process\_cycle, NULL,
20. "worker process", type);
22. //向已经创建的worker进程广播当前创建worker进程信息。。。
23. ch.pid = ngx\_processes[ngx\_process\_slot].pid;
24. ch.slot = ngx\_process\_slot;
25. ch.fd = ngx\_processes[ngx\_process\_slot].channel[0];
27. //接下来详细讲解此方法
28. ngx\_pass\_open\_channel(cycle, &ch);
29. }
30. }
32. **static** **void**
33. ngx\_pass\_open\_channel(ngx\_cycle\_t \*cycle, ngx\_channel\_t \*ch)
34. {
35. ngx\_int\_t  i;
37. **for** (i = 0; i < ngx\_last\_process; i++) {
39. //跳过自己和异常的worker
40. **if** (i == ngx\_process\_slot
41. || ngx\_processes[i].pid == -1
42. || ngx\_processes[i].channel[0] == -1)
43. {
44. **continue**;
45. }
47. ngx\_log\_debug6(NGX\_LOG\_DEBUG\_CORE, cycle->log, 0,
48. "pass channel s:%d pid:%P fd:%d to s:%i pid:%P fd:%d",
49. ch->slot, ch->pid, ch->fd,
50. i, ngx\_processes[i].pid,
51. ngx\_processes[i].channel[0]);
53. /\* TODO: NGX\_AGAIN \*/
54. //发送消息给其他的worker
55. ngx\_write\_channel(ngx\_processes[i].channel[0],
56. ch, **sizeof**(ngx\_channel\_t), cycle->log);
57. }
58. }

接下来要研究的函数是创建子进程的函数，上面方法中的ngx\_spawn\_process

1. src/os/unix/ngx\_process.c
2. ngx\_pid\_t
3. ngx\_spawn\_process(ngx\_cycle\_t \*cycle, ngx\_spawn\_proc\_pt proc, **void** \*data,
4. **char** \*name, ngx\_int\_t respawn)
5. {
6. //proc:是子进程的执行函数，
7. //data是参数，
8. //name是子进程的名字
9. u\_long     on;
10. ngx\_pid\_t  pid;
11. ngx\_int\_t  s;//将要创建的子进程在进程表中的位置
13. **if** (respawn >= 0) {
14. //替换进程ngx\_processes[respawn],可安全重用该进程表项
15. s = respawn;
17. } **else** {
18. //先找到一个被回收的进程表象
19. **for** (s = 0; s < ngx\_last\_process; s++) {
20. **if** (ngx\_processes[s].pid == -1) {
21. **break**;
22. }
23. }
24. //进程表已经满
25. **if** (s == NGX\_MAX\_PROCESSES) {
26. ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, 0,
27. "no more than %d processes can be spawned",
28. NGX\_MAX\_PROCESSES);
29. **return** NGX\_INVALID\_PID;
30. }
31. }
33. //不是分离的子进程
34. **if** (respawn != NGX\_PROCESS\_DETACHED) {
36. /\* Solaris 9 still has no AF\_LOCAL \*/
37. //创建一对已经连接的无名socket
38. **if** (socketpair(AF\_UNIX, SOCK\_STREAM, 0, ngx\_processes[s].channel) == -1)
39. {
40. ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, ngx\_errno,
41. "socketpair() failed while spawning \"%s\"", name);
42. **return** NGX\_INVALID\_PID;
43. }
45. ngx\_log\_debug2(NGX\_LOG\_DEBUG\_CORE, cycle->log, 0,
46. "channel %d:%d",
47. ngx\_processes[s].channel[0],
48. ngx\_processes[s].channel[1]);
49. //设置socket为非阻塞模式
50. **if** (ngx\_nonblocking(ngx\_processes[s].channel[0]) == -1) {
51. ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, ngx\_errno,
52. ngx\_nonblocking\_n " failed while spawning \"%s\"",
53. name);
54. ngx\_close\_channel(ngx\_processes[s].channel, cycle->log);
55. **return** NGX\_INVALID\_PID;
56. }
58. **if** (ngx\_nonblocking(ngx\_processes[s].channel[1]) == -1) {
59. ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, ngx\_errno,
60. ngx\_nonblocking\_n " failed while spawning \"%s\"",
61. name);
62. ngx\_close\_channel(ngx\_processes[s].channel, cycle->log);
63. **return** NGX\_INVALID\_PID;
64. }
66. //开启channel[0]的消息驱动IO
67. on = 1;
68. //FIOASYNC:设置/清楚信号驱动异步I/O标志
69. /\*根据iocl 的第三个参数指向一个0 值或非0 值
70. 分别清除或设置针对本套接口的信号驱动异步I/O 标志，
71. 它决定是否收取针对本套接口的异步I/O 信号（SIGIO ）。
72. 本请求和O\_ASYNC 文件状态标志等效，
73. 而该标志可以通过fcntl 的F\_SETFL 命令清除或设置。\*/
74. **if** (ioctl(ngx\_processes[s].channel[0], FIOASYNC, &on) == -1) {
75. ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, ngx\_errno,
76. "ioctl(FIOASYNC) failed while spawning \"%s\"", name);
77. ngx\_close\_channel(ngx\_processes[s].channel, cycle->log);
78. **return** NGX\_INVALID\_PID;
79. }
81. //设置channel[0]的宿主，控制channel[0]的SIGIO信号只发给这个进程
82. **if** (fcntl(ngx\_processes[s].channel[0], F\_SETOWN, ngx\_pid) == -1) {
83. ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, ngx\_errno,
84. "fcntl(F\_SETOWN) failed while spawning \"%s\"", name);
85. ngx\_close\_channel(ngx\_processes[s].channel, cycle->log);
86. **return** NGX\_INVALID\_PID;
87. }
88. //若进程执行了exec后，关闭socket
89. **if** (fcntl(ngx\_processes[s].channel[0], F\_SETFD, FD\_CLOEXEC) == -1) {
90. ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, ngx\_errno,
91. "fcntl(FD\_CLOEXEC) failed while spawning \"%s\"",
92. name);
93. ngx\_close\_channel(ngx\_processes[s].channel, cycle->log);
94. **return** NGX\_INVALID\_PID;
95. }
96. //同上
97. **if** (fcntl(ngx\_processes[s].channel[1], F\_SETFD, FD\_CLOEXEC) == -1) {
98. ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, ngx\_errno,
99. "fcntl(FD\_CLOEXEC) failed while spawning \"%s\"",
100. name);
101. ngx\_close\_channel(ngx\_processes[s].channel, cycle->log);
102. **return** NGX\_INVALID\_PID;
103. }
104. //用于监听可读时间的socket
105. ngx\_channel = ngx\_processes[s].channel[1];
107. } **else** {
108. ngx\_processes[s].channel[0] = -1;
109. ngx\_processes[s].channel[1] = -1;
110. }
111. //设置当前子进程的进程表索引值
112. ngx\_process\_slot = s;
114. //创建子进程
115. pid = fork();
117. **switch** (pid) {
119. **case** -1:
120. ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, ngx\_errno,
121. "fork() failed while spawning \"%s\"", name);
122. ngx\_close\_channel(ngx\_processes[s].channel, cycle->log);
123. **return** NGX\_INVALID\_PID;
125. **case** 0:
126. //设置当前子进程的进程id
127. ngx\_pid = ngx\_getpid();
128. //子进程运行执行函数
129. proc(cycle, data);
130. **break**;
132. **default**:
133. **break**;
134. }
136. ngx\_log\_error(NGX\_LOG\_NOTICE, cycle->log, 0, "start %s %P", name, pid);
137. //设置一些进程表项字段
138. ngx\_processes[s].pid = pid;
139. ngx\_processes[s].exited = 0;
141. //如果是重复创建，即为替换进程，不用设置其他进程表字段，直接返回。
142. **if** (respawn >= 0) {
143. **return** pid;
144. }
146. 设置其他进程表字段
147. ngx\_processes[s].proc = proc;
148. ngx\_processes[s].data = data;
149. ngx\_processes[s].name = name;
150. ngx\_processes[s].exiting = 0;
152. //设置进程表项的一些状态字
153. **switch** (respawn) {
155. **case** NGX\_PROCESS\_NORESPAWN:
156. ngx\_processes[s].respawn = 0;
157. ngx\_processes[s].just\_spawn = 0;
158. ngx\_processes[s].detached = 0;
159. **break**;
161. **case** NGX\_PROCESS\_JUST\_SPAWN:
162. ngx\_processes[s].respawn = 0;
163. ngx\_processes[s].just\_spawn = 1;
164. ngx\_processes[s].detached = 0;
165. **break**;
167. **case** NGX\_PROCESS\_RESPAWN:
168. ngx\_processes[s].respawn = 1;
169. ngx\_processes[s].just\_spawn = 0;
170. ngx\_processes[s].detached = 0;
171. **break**;
173. **case** NGX\_PROCESS\_JUST\_RESPAWN:
174. ngx\_processes[s].respawn = 1;
175. ngx\_processes[s].just\_spawn = 1;
176. ngx\_processes[s].detached = 0;
177. **break**;
179. **case** NGX\_PROCESS\_DETACHED:
180. ngx\_processes[s].respawn = 0;
181. ngx\_processes[s].just\_spawn = 0;
182. ngx\_processes[s].detached = 1;
183. **break**;
184. }
186. **if** (s == ngx\_last\_process) {
187. ngx\_last\_process++;
188. }
190. **return** pid;
191. }

#### [ngx\_worker\_process\_cycle子进程执行](http://blog.csdn.net/lengzijian/article/details/7591025)

Nginx启动过程中最重要的部分是ngx\_init\_cycle，这个函数的流程图如下：



首先先来看几个相关的数据结构。 在nginx中，模块的结构是这样子的，首先所有的模块都是用ngx\_module\_t来表示，而模块又分为三类，分别是ngx\_core\_module\_t和ngx\_http\_module\_t，而在ngx\_module\_t中会包含这两个结构，只不过不同类的模块包含不同的结构。一般来说这部分就叫做ctx，我们写模块都会先定义一个ctx，然后包含到ngx\_module\_t中。这里有个type域用来标识模块的类型。

## 问题记录



