1. 安装

下载nginx-1.8.0.tar.gz

解压

cd nginx-1.8.0

./configure

make & make install

默认安装路径在/usr/local/nginx

luaJIT

wget -c http://luajit.org/download/LuaJIT-2.0.2.tar.gz

tar xzvf LuaJIT-2.0.2.tar.gz

cd LuaJIT-2.0.2

make install PREFIX=/usr/local/luajit

echo "/usr/local/luajit/lib" > /etc/ld.so.conf.d/usr\_local\_luajit\_lib.conf

ldconfig

#注意环境变量!

export LUAJIT\_LIB=/usr/local/luajit/lib

export LUAJIT\_INC=/usr/local/luajit/include/luajit-2.0

安装mudole

tar -xvf echo-nginx-module-0.59.tar.gz

tar -xvf lua-nginx-module-0.10.2.tar.gz

./configure --add-module=/usr/local/nginx/module/lua-nginx-module-0.10.2 --add-module=/usr/local/nginx/module/echo-nginx-module-0.59

cp /usr/local/nginx/sbin/nginx /usr/local/nginx/sbin/nginx-bak

cp objs/nginx /usr/local/nginx/sbin/

测试配置nginx.conf

location /test {

default\_type 'text/plain';

set $a 32;

echo $a;

set $a 56;

echo $a;

}

location = /lua-version {

default\_type 'text/plain';

content\_by\_lua ' if jit then

ngx.say(jit.version)

else

ngx.say(\_VERSION)

end

';

}

**Nginx顺序**

Nginx 处理每一个用户请求时，都是按照若干个不同阶段（phase）依次处理的，而不是根据配置文件上的顺序。

Nginx 处理请求的过程一共划分为 11 个阶段，按照执行顺序依次是

**NGX\_HTTP\_POST\_READ\_PHASE**, //0读取请求phase

**NGX\_HTTP\_SERVER\_REWRITE\_PHASE**,//1这个阶段主要是处理全局的(server block)的rewrite

**NGX\_HTTP\_FIND\_CONFIG\_PHASE**, //2这个阶段主要是通过uri来查找对应的location，然后根据loc\_conf设置r的相应变量

**NGX\_HTTP\_REWRITE\_PHASE**, //3这个主要处理location的rewrite

**NGX\_HTTP\_POST\_REWRITE\_PHASE**, //4postrewrite，这个主要是进行一些校验以及收尾工作，以便于交给后面的模块。

**NGX\_HTTP\_PREACCESS\_PHASE**, //5比如流控这种类型的access就放在这个phase，也就是说它主要是进行一些比较粗粒度的access。

**NGX\_HTTP\_ACCESS\_PHASE**, //6这个比如存取控制，权限验证就放在这个phase，一般来说处理动作是交给下面的模块做的.这个主要是做一些细粒度的access

**NGX\_HTTP\_POST\_ACCESS\_PHASE**, //7一般来说当上面的access模块得到access\_code之后就会由这个模块根据access\_code来进行操作

**NGX\_HTTP\_TRY\_FILES\_PHASE,** //8try\_file模块，就是对应配置文件中的try\_files指令，可接收多个路径作为参数，当前一个路径的资源无法找到，则自动查找下一个路径

**NGX\_HTTP\_CONTENT\_PHASE**, //9内容处理模块

**NGX\_HTTP\_LOG\_PHASE** //10log模块

**post-read:**

读取请求内容阶段

Nginx读取并解析完请求头之后就立即开始运行

例如模块 ngx\_realip 就在 post-read 阶段注册了处理程序，它的功能是迫使 Nginx 认为当前请求的来源地址是指定的某一个请求头的值。

**server-rewrite**

Server请求地址重写阶段

当 ngx\_rewrite 模块的set配置指令直接书写在 server 配置块中时，基本上都是运行在 server-rewrite 阶段

**find-config**

配置查找阶段

这个阶段并不支持 Nginx 模块注册处理程序，而是由 Nginx 核心来完成当前请求与 location 配置块之间的配对工作。

**rewrite**

Location请求地址重写阶段

当 ngx\_rewrite 模块的指令用于 location 块中时，便是运行在这个 rewrite 阶段。

另外，ngx\_set\_misc(设置md5、encode\_base64等) 模块的指令，还有 ngx\_lua 模块的 set\_by\_lua 指令和 rewrite\_by\_lua 指令也在此阶段。

**post-rewrite**

请求地址重写提交阶段

由 Nginx 核心完成 rewrite 阶段所要求的“内部跳转”操作,如果 rewrite 阶段有此要求的话。

**preaccess**

访问权限检查准备阶段

标准模块 ngx\_limit\_req 和 ngx\_limit\_zone 就运行在此阶段，前者可以控制请求的访问频度，而后者可以限制访问的并发度。

**access**

访问权限检查阶段

标准模块 ngx\_access、第三方模块 ngx\_auth\_request 以及第三方模块 ngx\_lua 的 access\_by\_lua 指令就运行在这个阶段。

配置指令多是执行访问控制性质的任务，比如检查用户的访问权限，检查用户的来源 IP 地址是否合法

**post-access**

访问权限检查提交阶段

主要用于配合 access 阶段实现标准 ngx\_http\_core 模块提供的配置指令 satisfy 的功能。

satisfy all(与关系)

satisfy any(或关系)

try-files

配置项try\_files处理阶段

专门用于实现标准配置指令 try\_files 的功能

如果前 N-1 个参数所对应的文件系统对象都不存在，try-files 阶段就会立即发起“内部跳转”到最后一个参数（即第 N 个参数）所指定的 URI.

**content**

内容产生阶段

Nginx 的 content 阶段是所有请求处理阶段中最为重要的一个，因为运行在这个阶段的配置指令一般都肩负着生成“内容”

并输出 HTTP 响应的使命。

log

日志模块处理阶段

记录日志

**子请求（subrequest）**

其实在Nginx 世界里有两种类型的“请求”，一种叫做“主请求”（main request），而另一种则叫做“子请求”（subrequest）。

所谓“主请求”，就是由 HTTP 客户端从 Nginx 外部发起的请求。比如，从浏览器访问Nginx就是一个“主请求”。

而“子请求”则是由 Nginx 正在处理的请求在 Nginx 内部发起的一种级联请求。“子请求”在外观上很像 HTTP 请求，但实现上却和 HTTP 协议乃至网络通信一点儿关系都没有。它是 Nginx 内部的一种抽象调用，目的是为了方便用户把“主请求”的任务分解为多个较小粒度的“内部请求”，并发或串行地访问多个 location 接口，然后由这些 location 接口通力协作，共同完成整个“主请求”。当然，“子请求”的概念是相对的，任何一个“子请求”也可以再发起更多的“子子请求”，甚至可以玩递归调用（即自己调用自己）。当一个请求发起一个“子请求”的时候，按照 Nginx 的术语，习惯把前者称为后者的“父请求”（parent request）。

[plain] view plain copy print?

location /main {

echo\_location /foo; # echo\_location发送子请求到指定的location

echo\_location /bar;

}

location /foo {

echo foo;

}

location /bar {

echo bar;

}

输出：

[plain] view plain copy print?

$ curl location/main

$ foo

bar

这里，main location就是发送2个子请求，分别到foo和bar，这就类似一种函数调用。

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

**协程（Coroutine）**

http://blog.csdn.net/chosen0ne/article/details/7304192

协程类似一种多线程，与多线程的区别有：

1. 协程并非os线程，所以创建、切换开销比线程相对要小。

2. 协程与线程一样有自己的栈、局部变量等，但是协程的栈是在用户进程空间模拟的，所以创建、切换开销很小。

3. 多线程程序是多个线程并发执行，也就是说在一瞬间有多个控制流在执行。而协程强调的是一种多个协程间协作的关系，只有当一个协程主动放弃执行权，另一个协程才能获得执行权，所以在某一瞬间，多个协程间只有一个在运行。

4. 由于多个协程时只有一个在运行，所以对于临界区的访问不需要加锁，而多线程的情况则必须加锁。

5. 多线程程序由于有多个控制流，所以程序的行为不可控，而多个协程的执行是由开发者定义的所以是可控的。

Nginx的每个Worker进程都是在epoll或kqueue这样的事件模型之上，封装成协程，每个请求都有一个协程进行处理。这正好与Lua内建协程的模型是一致的，所以即使ngx\_lua需要执行Lua，相对C有一定的开销，但依然能保证高并发能力。

**ngx\_lua原理**

ngx\_lua将Lua嵌入Nginx，可以让Nginx执行Lua脚本，并且高并发、非阻塞的处理各种请求。Lua内建协程，这样就可以很好的将异步回调转换成顺序调用的形式。ngx\_lua在Lua中进行的IO操作都会委托给Nginx的事件模型，从而实现非阻塞调用。开发者可以采用串行的方式编写程序，ngx\_lua会自动的在进行阻塞的IO操作时中断，保存上下文；然后将IO操作委托给Nginx事件处理机制，在IO操作完成后，ngx\_lua会恢复上下文，程序继续执行，这些操作都是对用户程序透明的。

每个NginxWorker进程持有一个Lua解释器或者LuaJIT实例，被这个Worker处理的所有请求共享这个实例。每个请求的Context会被Lua轻量级的协程分割，从而保证各个请求是独立的。

ngx\_lua采用“one-coroutine-per-request”的处理模型，对于每个用户请求，ngx\_lua会唤醒一个协程用于执行用户代码处理请求，当请求处理完成这个协程会被销毁。每个协程都有一个独立的全局环境（变量空间），继承于全局共享的、只读的“comman data”。所以，被用户代码注入全局空间的任何变量都不会影响其他请求的处理，并且这些变量在请求处理完成后会被释放，这样就保证所有的用户代码都运行在一个“sandbox”（沙箱），这个沙箱与请求具有相同的生命周期。

得益于Lua协程的支持，ngx\_lua在处理10000个并发请求时只需要很少的内存。根据测试，ngx\_lua处理每个请求只需要2KB的内存，如果使用LuaJIT则会更少。所以ngx\_lua非常适合用于实现可扩展的、高并发的服务

**Nginx配置指令：**

init\_by\_lua http

set\_by\_lua server, server if, location, location if

rewrite\_by\_lua http, server, location, location if

access\_by\_lua http, server, location, location if

content\_by\_lua location, location if

header\_filter\_by\_lua http, server, location, location if

body\_filter\_by\_lua http, server, location, location if

log\_by\_lua http, server, location, location if

Timer

init\_by\_lua:

在nginx重新加载配置文件时，运行里面lua脚本，常用于全局变量的申请。

例如lua\_shared\_dict共享内存的申请，只有当nginx重起后，共享内存数据才清空，这常用于统计。

**1、set\_by\_lua和set\_by\_lua\_file**

和set指令一样用于设置Nginx变量并且在rewrite阶段执行，只不过这个变量是由lua脚本计算并返回的

例如：

配置：

location = /adder {

set\_by\_lua $res "

local a = tonumber(ngx.arg[1])

local b = tonumber(ngx.arg[2])

return a + b" $arg\_a $arg\_b;

echo $res;

}

输出：

$ curl 'localhost/adder?a=25&b=75'

$ 100

set\_by\_lua\_file执行Nginx外部的lua脚本，可以避免在配置文件中使用大量的转义。

配置：

location = /fib {

set\_by\_lua\_file $res "conf/adder.lua" $arg\_n;

echo $res;

}

adder.lua:

local a = tonumber(ngx.arg[1])

local b = tonumber(ngx.arg[2])

return a + b

输出：

$ curl 'localhost/adder?a=25&b=75

$ 100

rewrite\_by\_lua:

在access阶段前运行，主要用于rewrite

**2、access\_by\_lua和access\_by\_lua\_file**

运行在access阶段，用于访问控制。Nginx原生的allow和deny是基于ip的，通过access\_by\_lua能完成复杂的访问控制，比如，访问数据库进行用户名、密码验证等。

例如、配置：

location /auth {

access\_by\_lua '

if ngx.var.arg\_user == "ntes" then

return

else

Ngx.exit(ngx.HTTP\_FORBIDDEN)

end

';

echo 'welcome ntes';

}

输出：

$ curl 'localhost/auth?user=sohu'

$ Welcome ntes

$ curl 'localhost/auth?user=ntes'

$ <html>

<head><title>403 Forbidden</title></heda>

<body bgcolor="white">

<center><h1>403 Forbidden</h1></center>

<hr><center>ngx\_openresty/1.0.10.48</center>

</body>

</html>

3、**rewrite\_by\_lua和rewrite\_by\_lua\_file**

实现url重写，在rewrite阶段执行。

例如、配置：

location = /foo {

rewrite\_by\_lua 'ngx.exec("/bar")';

echo 'in foo';

}

location = /bar {

echo 'in bar';

}

输出：

$ curl 'localhost/lua'

$ in bar

**4、content\_by\_lua和content\_by\_lua\_file**

Content handler在content阶段执行，生成内容（content）并输出HTTP响应。由于content阶段只能有一个handler，所以在与echo模块使用时，不能同时生效，我测试的结果是content\_by\_lua会覆盖echo。这和之前的hello world的例子是类似的。

例如 配置（直接响应）：

location = /lua {

content\_by\_lua 'ngx.say("Hello, Lua!")';

}

输出：

$ curl 'localhost/lua'

$ Hello, Lua!

配置（在Lua中访问Nginx变量）：

location = /hello {

content\_by\_lua 'local who = ngx.var.arg\_who

ngx.say("Hello, ", who, "!")';

}

输出：

$ curl 'localhost/hello?who=world

$ Hello, world!

**header\_filter\_by\_lua:**

一般只用于设置Cookie和Headers等

该阶段不能运行Output API、Control API、Subrequest API、Cosocket API

**body\_filter\_by\_lua:**

一般会在一次请求中被调用多次, 因为这是实现基于 HTTP 1.1 chunked 编码的所谓“流式输出”的。

该阶段不能运行Output API、Control API、Subrequest API、Cosocket API

**log\_by\_lua:**

该阶段总是运行在请求结束的时候，用于请求的后续操作，如在共享内存中进行统计数据,如果要高精确的数据统计，应该使用body\_filter\_by\_lua。

该阶段不能运行Output API、Control API、Subrequest API、Cosocket API

****Nginx全局内存****

使用过如Java的朋友可能知道如Ehcache等这种进程内本地缓存，Nginx是一个Master进程多个Worker进程的工作方式，因此我们可能需要在多个Worker进程中共享数据，那么此时就可以使用ngx.shared.DICT来实现全局内存共享。

首先在nginx.conf的http部分分配内存大小

#共享全局变量，在所有worker间共享

lua\_shared\_dict shared\_data 1m;

****Nginx操作****

**1、验证nginx配置文件是否正确**

方法一：进入nginx安装目录sbin下，输入命令./nginx -t

看到如下显示nginx.conf syntax is ok

nginx.conf test is successful

说明配置文件正确！

方法二：在启动命令-c前加-t

**2、重启Nginx服务**

方法一：进入nginx可执行目录sbin下，输入命令./nginx -s reload 即可

**Openresty安装**

tar -xvf ngx\_openresty-1.7.4.1.tar.gz

cd ngx\_openresty-1.7.4.1

./configure --prefix=/usr/local/test\_ngx\_openresty

make && make install

**Nginx.conf配置**

2核CPU，开启2个进程

worker\_processes 2;

worker\_cpu\_affinity 01 10;

01表示启用第一个CPU内核，10表示启用第二个CPU内核

worker\_cpu\_affinity 01 10;表示开启两个进程，第一个进程对应着第一个CPU内核，第二个进程对应着第二个CPU内核。

worker\_cpu\_affinity配置是写在/etc/nginx/nginx.conf里面的。

2核是 01，四核是0001，8核是00000001，有多少个核，就有几位数，1表示该内核开启，0表示该内核关闭。

#指定一个nginx进程可以打开的最多文件描述符数目

worker\_rlimit\_nofile 65535;

worker\_rlimit\_nofile:理论值应该是最多打开文件数（ulimit -n）与nginx 进程数相除，但是nginx 分配请求并不是那么均匀，所以最好与ulimit -n 的值保持一致。

#设定nginx的工作模式及连接数上限

**events**{

use epoll; #epoll是多路复用IO(I/O Multiplexing)中的一种方式,但是仅用于linux2.6以上内核,可以大大提高nginx的性能

worker\_connections 65536;#设定worker的最大连接数

}

**http内指令配置**

**lua\_code\_cache on | off**

作用：lua\_code\_cache是nginx\_lua模块的一条指令。它为 \*\_by\_lua\_file(如 set\_by\_lua\_file 和 content\_by\_lua\_file) 这些指令以及Lua模块, 开启或关闭Lua代码缓存.（nginx\_lua模块说明：http://wiki.nginx.org/HttpLuaModule#lua\_code\_cache）

如果**关闭，每个ngx\_lua处理的请求将运行在一个独立的Lua VM实例**里，0.9.3版本后有效. 所以 set\_by\_lua\_file, content\_by\_lua\_file, access\_by\_lua\_file, 等等指令引用的Lua文件将不再缓存到内存， 并且所有Lua模块每次都会从头重新加载. 这样开发者就可以避免改代码然后重启nginx的操作.

但是, 那些直接写在 nginx.conf 里的代码比如由 set\_by\_lua, content\_by\_lua, access\_by\_lua, and rewrite\_by\_lua 指定的代码不会在你编辑他们时实时更新，因为只有发送HUP信号通知Nginx才会正确重新加载Nginx的config文件。

即使打开了代码缓存, 那些\*\_by\_lua\_file指令里由 dofile 或 loadfile 加载的Lua文件也无法被缓存 (除非你自己手动缓存了他们). 通常你可以用 init\_by\_lua 或 init\_by\_lua\_file 其中一种指令来加载所有这些无法被缓存 的文件，或把这些文件做成真正的Lua模块并用require加载.

生产环境下千万别关闭Lua代码缓存，只能用在开发模式下，因为对性能有十分大的影响（每次IO读取和编译Lua代码消耗很大， 简单的hello world都会慢一个数量级

lua\_code\_cache on 时，全局变量的生命期是每请求的

**lua\_package\_path**

语法：lua\_package\_path <lua-style-path-str>

默认：由lua的环境变量决定

适用上下文：http

设置lua代码的寻找目录。

例如：lua\_package\_path "/opt/nginx/conf/www/?.lua;;";

具体的路径设置要参考lua的模块机制

**resolver指令**

语法: resolver address ... [valid=time];

默认值: —

配置段: http, server, location

配置DNS服务器IP地址。可以指定多个，以轮询方式请求。

nginx会缓存解析的结果。默认情况下，缓存时间是名字解析响应中的TTL字段的值，可以通过valid参数更改。

**log\_format指令：**

语法 log\_format name string ...

默认值 combined "..."

使用字段 http

参数说明：

$remote\_addr 和 $http\_x\_forwarded\_for：记录客户端的ip地址。

$remote\_user：记录客户端用户名称。

$time\_local：记录访问时间与时区。

$request：记录请求的URL与HTTP协议。

$status：记录请求状态；成功是200。

$body\_bytes\_sent：记录发送给客户端文件主体内容大小。

$http\_referer：记录从那个页面链接访问过来的。

$http\_user\_agent：记录客户端浏览器的相关信息

**access\_log**  设置nginx是否将存储访问日志。关闭这个选项可以让读取磁盘IO操作更快

**error\_log**  告诉nginx只能记录严重的错误

**Timeout(超时)设置**

**使用上下文：**http server location

**1、keepalive\_timeout**

语法 keepalive\_timeout timeout [ header\_timeout ]  
默认值 75s

说明:第一个参数指定了与client的keep-alive连接超时时间，服务器将在这个超时时间过后关闭链接。我们将它设置低些可以让ngnix持续工作的时间更长。可选的第二个参数指定了在响应头Keep-Alive: timeout=time中的time值。这个头能够让一些浏览器主动关闭连接，这样服务器就不必要去关闭连接了。

**2、lingering\_timeout**

语法 lingering\_timeout time  
默认值 5s  
上下文 http server location  
说明 lingering\_close生效后，在关闭连接前，会检测是否有用户发送的数据到达服务器，如果超过lingering\_timeout时间后还没有数据可读，就直接关闭连接；否则，必须在读取完连接缓冲区上的数据并丢弃掉后才会关闭连接。

**3、resolver\_timeout指令**

语法: resolver\_timeout time;

默认值: resolver\_timeout 30s;

配置段: http, server, location

DNS解析超时时间。

**4、proxy\_connect\_timeout**  
语法 proxy\_connect\_timeout time   
默认值 60s  
上下文 http server location  
说明 该指令设置与upstream server的连接超时时间，有必要记住，这个超时不能超过75秒。  
这个不是等待后端返回页面的时间，那是由proxy\_read\_timeout声明的。如果你的upstream服务器起来了，但是hanging住了（例如，没有足够的线程处理请求，所以把你的请求放到请求池里稍后处理），那么这个声明是没有用的，由于与upstream服务器的连接已经建立了。

**5、proxy\_read\_timeout**  
语法 proxy\_read\_timeout time   
默认值 60s  
上下文 http server location  
说明 该指令设置与代理服务器的读超时时间。它决定了nginx会等待多长时间来获得请求的响应。这个时间不是获得整个response的时间，而是两次reading操作的时间。  
6、**proxy\_send\_timeout**  
语法 proxy\_send\_timeout time   
默认值 60s  
上下文 http server location  
说明 这个指定设置了发送请求给upstream服务器的超时时间。超时设置不是为了整个发送期间，而是在两次write操作期间。如果超时后，upstream没有收到新的数据，nginx会关闭连接  
7、**proxy\_upstream\_fail\_timeout（fail\_timeout）**  
语法 server address [fail\_timeout=30s]  
默认值 10s  
上下文 upstream  
说明 Upstream模块下 server指令的参数，设置了某一个upstream后端失败了指定次数（max\_fails）后，该后端不可操作的时间，默认为10秒

**8、client\_header\_timeout 和client\_body\_timeout**

设置请求头和请求体(各自)的超时时间。我们也可以把这个设置低些。

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

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

**sendfile**

#sendfile 指令指定 nginx 是否调用 sendfile 函数（zero copy 方式）来输出文件，对于普通应用，  
    #必须设为 on,如果用来进行下载等应用磁盘IO重负载应用，可设置为 off，以平衡磁盘与网络I/O处理速度，降低系统的uptime.  
 sendfile        on;

**tcp\_nodelay**

指令tcp\_nodelay作用于socket参数TCP\_NODELAY。

在这之前，我们先说说**nagle**缓存算法，有些应用程序在网络通讯的时候会发送很少的字节，比如说一个字节，那么再加TCP协议本身，实际上发的要41个字节，这样的效率是很低的。这时候nagle算法就应运而生了，它将要发送的数据存放在缓存里，当积累到一定量或一定时间，再将它们发送出去。

这里TCP\_NODELAY就是nagle启用与否的开关，所以下面的指令的效果就是禁用nagle算法，也即不缓存数据。

tcp\_nodelay on;

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

**server\_tokens**并不会让nginx执行的速度更快，但它可以关闭在错误页面中的nginx版本数字，这样对于安全性是有好处的

**include** 只是一个在当前文件中包含另一个文件内容的指令。这里我们使用它来加载稍后会用到的一系列的MIME类型。

**default\_type** 设置文件使用的默认的MIME-type。

**charset** 设置我们的头文件中的默认的字符集

include /etc/nginx/mime.types;

default\_type text/html;

charset UTF-8;

**internal**

**语法：internal   
默认值：no   
使用字段： location   
internal指令指定某个location只能被“内部的”请求调用，外部的调用请求会返回”Not found” (404)**

**压缩设置**

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

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

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

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

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

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

**gzip\_type** 设置需要压缩的数据格式。上面例子中已经有一些了，你可以再添加更多的格式

gzip on;

gzip\_disable "msie6";

# gzip\_static on;

gzip\_proxied any;

gzip\_min\_length 1000;

gzip\_comp\_level 4;

gzip\_types text/plain text/css application/json application/x-javascript text/xml application/xml application/xml+rss text/javascript;

**负载均衡upstream目前支持5种方式的分配**

**1、轮询（默认）**

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

**2、weight**

指定轮询几率，weight和访问比率成正比，用于后端服务器性能不均的情况。

例如：

upstream bakend {

server 192.168.0.14 weight=10;

server 192.168.0.15 weight=10;

}

**3、ip\_hash**

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

例如：

upstream bakend {

ip\_hash;

server 192.168.0.14:88;

server 192.168.0.15:80;

}

**4、fair（第三方）**

按后端服务器的响应时间来分配请求，响应时间短的优先分配。

upstream backend {

server server1;

server server2;

fair;

}

**5、url\_hash（第三方）**

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

例：在upstream中加入hash语句，server语句中不能写入weight等其他的参数，hash\_method是使用的hash算法

upstream backend {

server squid1:3128;

server squid2:3128;

hash $request\_uri;

hash\_method crc32;

}

tips:

upstream bakend{#定义负载均衡设备的Ip及设备状态

ip\_hash;

server 127.0.0.1:9090 down;

server 127.0.0.1:8080 weight=2;

server 127.0.0.1:6060;

server 127.0.0.1:7070 backup;

}

在需要使用负载均衡的server中增加

proxy\_pass http://bakend/;

每个设备的状态设置为:

1.down 表示当前的server暂时不参与负载

2.weight 默认为1.weight越大，负载的权重就越大。

3.max\_fails ：允许请求失败的次数默认为1.当超过最大次数时，返回proxy\_next\_upstream 模块定义的错误

4.fail\_timeout:max\_fails次失败后，暂停的时间。

5.backup： 其它所有的非backup机器down或者忙的时候，请求backup机器。所以这台机器压力会最轻

openresty shared.dict

Ngx.location.capture

**nginx配置proxy\_pass**

需要注意转发的路径配置   
1、location /test/ {   
                proxy\_pass http://t6:8300;   
     }   
2、location /test/ {   
                proxy\_pass http://t6:8300/;   
     }   
上面两种配置，区别只在于proxy\_pass转发的路径后是否带 ****“/”****  
针对情况2，如果访问url = http://server/test/test.jsp，则被nginx代理后，请求路径会变为 http://proxy\_pass/test.jsp，直接访问server的根资源   
针对情况1，如果访问url = http://server/test/test.jsp，则被nginx代理后，请求路径会便问http://proxy\_pass/test/test.jsp，将test/ 作为根路径，请求test/路径下的资源

参考：

<https://www.nginx.com/resources/wiki/modules/lua/>

<http://my.oschina.net/u/586648/blog/146824>

<http://tengine.taobao.org/book/>

<http://www.buges.cn/post/39.html> openresty lua api

<http://www.cnblogs.com/whiteyun/archive/2009/08/12/1543184.html> Lua内置的函数库

<http://coolshell.cn/articles/10739.html> lua 语言基础

<http://www.tuicool.com/articles/na6fyaQ> mysql\redis

<http://wiki.jikexueyuan.com/project/openresty-best-practice/cache.html>