**预研 nginx gzip/https/cache/proxy 配置、运行细节（什么时候压缩，header需要加什么，header有什么变化，内容长度，根据header变化）、优化（本身的优化，对那些开启）、性能测试**

# nginx gzip

## 1.gzip配置

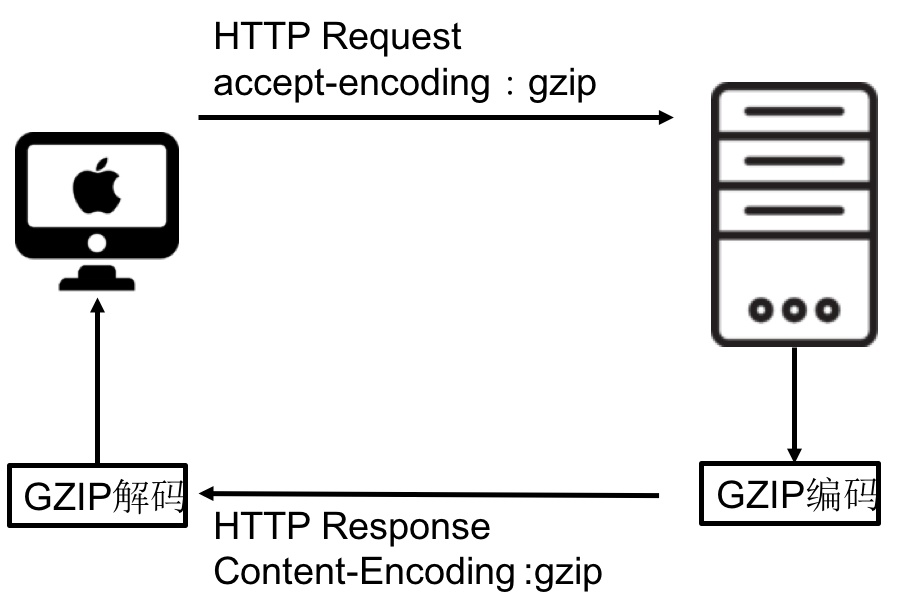
gzip是GNUzip的缩写，最早用于UNIX系统的文件压缩。HTTP协议上的gzip编码是一种用来改进web应用程序性能的技术，web服务器和客户端（浏览器）必须共同支持gzip。目前主流的浏览器，Chrome,firefox,IE等都支持该协议。常见的服务器如Apache，Nginx，IIS同样支持gzip。

gzip压缩比率在3到10倍左右，可以大大节省服务器的网络带宽。而在实际应用中，并不是对所有文件进行压缩，通常只是压缩静态文件。

gzip来源于nginx提供的HttpGzip模块(准备地说是ngx\_http\_gzip\_module模块)。这个模块支持在线实时压缩输出数据流。

Gzip压缩作用：将响应报⽂发送⾄客户端之前可以启⽤压缩功能，这能够有效地节约带宽，并提⾼响应⾄客户端的速度。

注意：使用SSL / TLS协议时，压缩的响应可能会受到[BREACH](https://en.wikipedia.org/wiki/BREACH)攻击。



gzip工作原理图

1)浏览器请求url，并在request header中设置属性accept-encoding:gzip。表明浏览器支持gzip。

2)服务器收到浏览器发送的请求之后，判断浏览器是否支持gzip，如果支持gzip，则向浏览器传送压缩过的内容，不支持则向浏览器发送未经压缩的内容。一般情况下，浏览器和服务器都支持gzip，response headers返回包含content-encoding:gzip。

3)浏览器接收到服务器的响应之后判断内容是否被压缩，如果被压缩则解压缩显示页面内容。

**使用范例：**

gzip on;

gzip\_min\_length 1k;

gzip\_comp\_level 6;

gzip\_proxied expired no-cache no-store private auth;

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

内置变量$gzip\_ratio可以获取到gzip的压缩比率

**指令：**

[#gzip gzip]

[#gzip\_buffers gzip\_buffers]

[#gzip\_comp\_level gzip\_comp\_level]

[#gzip\_min\_length gzip\_min\_length]

[#gzip\_http\_version gzip\_http\_version]

[#gzip\_proxied gzip\_proxied]

[#gzip\_types gzip\_types]

### gzip

语法:gzip on|off

默认值:gzip off

作用域: http, server, location, if (x) location

开启或者关闭gzip模块

### gzip\_buffers

语法:gzip\_buffers number size

默认值:gzip\_buffers 4 4k/8k

作用域: http, server, location

设置系统获取几个单位的缓存用于存储gzip的压缩结果数据流。 例如:

4 4k 代表以4k为单位，按照原始数据大小以4k为单位的4倍申请内存。

4 8k 代表以8k为单位，按照原始数据大小以8k为单位的4倍申请内存。

如果没有设置，默认值是申请跟原始数据相同大小的内存空间去存储gzip压缩结果。

### gzip\_comp\_level

语法:gzip\_comp\_level 1..9

默认值:gzip\_comp\_level 1

作用域: http, server, location

gzip压缩比，1 压缩比最小处理速度最快，9 压缩比最大但处理最慢（传输快但比较消耗cpu）。

### gzip\_min\_length

语法:gzip\_min\_length length

默认值:gzip\_min\_length 0

作用域: http, server, location

设置允许压缩的页面最小字节数，页面字节数从header头中的Content-Length中进行获取。

默认值是0，不管页面多大都压缩。

建议设置成大于1k的字节数，小于1k可能会越压越大。 即: gzip\_min\_length 1024

### gzip\_http\_version

语法:gzip\_http\_version 1.0|1.1

默认值:gzip\_http\_version 1.1

作用域: http, server, location

识别http的协议版本。由于早期的一些浏览器或者http客户端，可能不支持gzip自解压，用户就会看到乱码，所以做一些判断还是有必要的。 注：21世纪都来了，现在除了类似于百度的蜘蛛之类的东西不支持自解压，99.99%的浏览器基本上都支持gzip解压了，所以可以不用设这个值,保持系统默认即可。

### gzip\_proxied

语法:gzip\_proxied [off|expired|no-cache|no-store|private|no\_last\_modified|no\_etag|auth|any] ...

默认值:gzip\_proxied off

作用域: http, server, location

Nginx作为反向代理的时候启用，开启或者关闭后端服务器返回的结果，匹配的前提是后端服务器必须要返回包含"Via"的 header头。

**off**- 关闭所有的代理结果数据的压缩

**expired**- 启用压缩，如果header头中包含 "Expires" 头信息

**no-cache**- 启用压缩，如果header头中包含 "Cache-Control:no-cache" 头信息

**no-store**- 启用压缩，如果header头中包含 "Cache-Control:no-store" 头信息

**private**- 启用压缩，如果header头中包含 "Cache-Control:private" 头信息

**no\_last\_modified**- 启用压缩,如果header头中不包含 "Last-Modified" 头信息

**no\_etag**- 启用压缩 ,如果header头中不包含 "ETag" 头信息

**auth**- 启用压缩 , 如果header头中包含 "Authorization" 头信息

**any**- 无条件启用压缩

### gzip\_types

语法:gzip\_types mime-type [mime-type ...]

默认值:gzip\_types text/html

作用域: http, server, location

匹配MIME类型进行压缩，（无论是否指定）**"text/html"类型总是会被压缩的**。

注意：如果作为http server来使用，主配置文件中要包含文件类型配置文件

http

{

include conf/mime.types;

......

}

如果你希望压缩常规的文件类型，可以写成这个样子

http

{

include conf/mime.types;

gzip on;

gzip\_min\_length 1000;

gzip\_buffers 4 8k;

gzip\_http\_version 1.1;

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

......

}

Gzip压缩可以配置http,[server](https://www.centos.bz/tag/server/)和[location](https://www.centos.bz/tag/location/)模块下。Nginx开启Gzip压缩功能的配置如下:

#修改nginx配置文件 /usr/local/nginx/conf/nginx.conf

[root@localhost ~]# vim /usr/local/nginx/conf/nginx.conf #将以下配置放到nginx.conf的http{ ... }节点中

#修改配置为

gzip on; #开启gzip压缩功能

gzip\_min\_length 10k; #设置允许压缩的页面最小字节数; 这里表示如果文件小于10个字节，就不用压缩，因为没有意义，本来就很小.

gzip\_buffers 4 16k; #设置压缩缓冲区大小，此处设置为4个16K内存作为压缩结果流缓存

gzip\_http\_version 1.1; #压缩版本

gzip\_comp\_level 2; #设置压缩比率，最小为1，处理速度快，传输速度慢；9为最大压缩比，处理速度慢，传输速度快; 这里表示压缩级别，可以是0到9中的任一个，级别越高，压缩就越小，节省了带宽资源，但同时也消耗CPU资源，所以一般折中为6

gzip types text/css text/xml application/javascript; #制定压缩的类型,线上配置时尽可能配置多的压缩类型!

gzip\_disable "MSIE [1-6]\."; #配置禁用gzip条件，支持正则。此处表示ie6及以下不启用gzip（因为ie低版本不支持）

gzip vary on; #选择支持vary header；改选项可以让前端的缓存服务器缓存经过gzip压缩的页面; 这个可以不写，表示在传送数据时，给客户端说明我使用了gzip压缩

线上使用的Gzip压缩配置

[root@external-lb02 ~]# cat /data/nginx/conf/nginx.conf

........

http {

.......

gzip on;

gzip\_min\_length 1k;

gzip\_buffers 4 16k;

gzip\_http\_version 1.1;

gzip\_comp\_level 9;

gzip\_types text/plain application/x-javascript text/css application/xml text/javascript application/x-[httpd](https://www.centos.bz/tag/httpd/)-php application/javascript application/json;

gzip\_disable "MSIE [1-6]\.";

gzip\_vary on;

}

## 运行细节（什么时候压缩，header需要加什么，header有什么变化，内容长度，根据header变化）

什么时候压缩？

本次测试使用的浏览器为Chrome,打开控制台查看网络信息可以看到request headers中包含：accept-encoding:gzip, deflate, sdch，表明chrome浏览器支持这三种压缩。这里值得一提的是accept-encoding中添加的另外两个压缩方式deflate和sdch。deflate与gzip使用的压缩算法几乎相同，这里不再赘叙。sdch是Shared Dictionary Compression over HTTP的缩写，即通过字典压缩算法对各个页面中相同的内容进行压缩，减少相同的内容的传输。sdch是Google推出的，目前只在Google Chrome, Chromium 和Android中支持。图2为浏览器发送的request header。图3为服务器返回的response header。

图2 淘宝request header

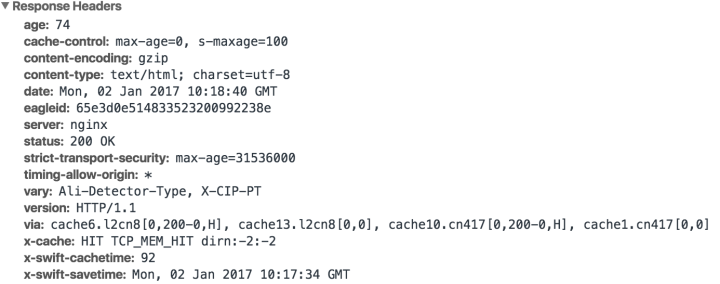


图3 淘宝response header

通过图2以图3很明显可以看出网站支持gzip，那么当支持gzip之后，压缩效率如何体现呢？通常浏览器都有现成的插件检测gzip压缩效率，如firefoxd的YSlow插件，我这里使用了网站[http://gzip.zzbaike.com/](https://link.zhihu.com/?target=http%3A//gzip.zzbaike.com/)做了检测。检测结果如图4所示：

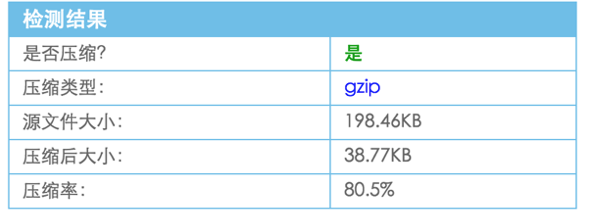


图4 淘宝gzip检测结果

很明显可以看出，通过使用gzip，静态文件被压缩了80.5%，极大的节省了服务器的网络带宽，这对于访问量巨大的淘宝来讲节约的流量非常可观。

在企业级应用中，通常被使用到的服务器有nginx，Apache等。nginx是一个高性能的HTTP和反向代理服务器，本文接下来的内容会介绍一下在Nginx中如何开启gzip。

response headers中返回了content-encoding:gzip，表明gzip开启成功。

## 性能测试（压测，进程本身的资源占用，cpu，内存的变化，gzip等级不同压缩比和速度）

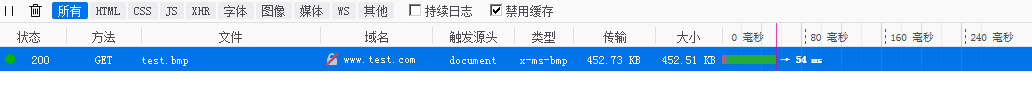
### 单个测试：

如果不开启Gzip压缩功能(即注释掉Gzip的相关配置), 查看某个图片大小

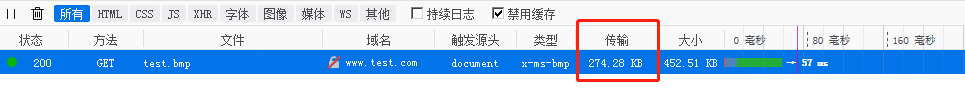
[root@external-lb02 ~]# ll -h /data/web//www/test.bmp

-rw-r--r-- 1 root root 453K 3月 14 18:43 /data/web//www/test.bmp

如下可知, 文件没有被压缩,文件传输大小还是400多K



如果开启Nginx的Gzip压缩功能(即打开Gzip的相关配置), 然后再次访问test.bmp图片, 发现压缩后的该图片文件传输大小只有200多K !



通过上面测试对比, 发现Nginx开启Gzip压缩功能后, 定义的gzip type的文件在传输时的大小明显变小, 这样这会大大提高nginx访问性能.

直接用curl测试命令:

[root@fvtlb02 ~]# curl -I -H "Accept-Encoding: gzip, deflate" "http://127.0.0.1:80"

HTTP/1.1 200 OK

Server: nginx/1.12.2

Date: Mon, 26 Nov 2018 02:19:16 GMT

Content-Type: application/javascript; charset=utf-8

Connection: keep-alive

Vary: Accept-Encoding

Last-Modified: Sun, 25 Nov 2018 22:28:15 GMT

Vary: Accept-Encoding

ETag: W/"5bfb21ff-40be"

Content-Encoding: gzip

如上,response header头信息中出现"Conten\_Encoding: gzip" , 就说明Nginx已开启了压缩 (在浏览器访问, 通过F12看请求的响应头部 也是一样)

Nginx的Gzip压缩功能虽然好用，但是下面两类文件资源不太建议启用此压缩功能。

### 压测：

## 优化

以下资源不需要用gzip：

### **1) 图片类型资源 (还有视频文件)**

原因：图片如jpg、png文件本身就会有压缩，所以就算开启gzip后，压缩前和压缩后大小没有多大区别，所以开启了反而会白白的浪费资源。（可以试试将一张jpg图片压缩为zip，观察大小并没有多大的变化。虽然zip和gzip算法不一样，但是可以看出压缩图片的价值并不大）

### **2) 大文件资源**

原因：会消耗大量的cpu资源，且不一定有明显的效果。

## 需要和不需要压缩的对象

**纯文本内容压缩比很高，因此纯文本的内容最好进行压缩，例如html css js xml shtml等格式的文件。**

**被压缩的纯文本文件必须要大于1KB，由于压缩算法的特殊原因，极小的文件压缩后反而会变大。**

**图片、视频等文件尽量不要压缩，因为这些文件大多是经过压缩的，如果再压缩就可能会减小很少，或者有可能增大，同时压缩时还会消耗大量的CPU、内存资源。**

# nginx HTTPS

## 背景：

网络就是实现不同主机之间的通讯，网络出现之初利用TCP/IP协议簇的相关协议概念，已经满足了互连两台主机之间可以进行通汛的目的，虽然看似简简单单几句话，就描述了网络概念与网络出现的目的，但是为了真正实现两台主机之间的稳定可靠通讯，其实是一件非常困难的事情了，如果还要再通讯的基础上保证数据传输的安全性，可想而知，绝对是难上加难，因此，网络发明之初，并没有太关注TCP/IP互联协议中的安全问题。

对于默认的两台主机而言，早期传输数据信息并没有通过加密方式传输数据，设备两端传输的数据本身实际是明文的，只要能截取到传输的数据包，就可以直接看到传输的数据信息，所以根本没有安全性可言。

目前已存在一些提供认证(基本认证和摘要认证)和报文完整性检查(摘要 qop="auth-int")的轻量级方法。对很多网络事务来说，这些方法都是很好用的，但对大规模的购物、银行事务，或者对访问机密数据来说，并不足够强大。这些更 为重要的事务需要将 HTTP 和数字加密技术结合起来使用，才能确保安全。

HTTP 的安全版本要高效、可移植且易于管理，不但能够适应不断变化的情况而且还应 该能满足社会和政府的各项要求。我们需要一种能够提供下列功能的 HTTP 安全技术。

服务器认证 (客户端知道它们是在与真正的而不是伪造的服务器通话)。

客户端认证 (服务器知道它们是在与真正的而不是伪造的客户端通话)。

完整性 (客户端和服务器的数据不会被修改)。

加密 (客户端和服务器的对话是私密的，无需担心被窃听)。

效率 (一个运行的足够快的算法，以便低端的客户端和服务器使用)。

普适性 (基本上所有的客户端和服务器都支持这些协议)。

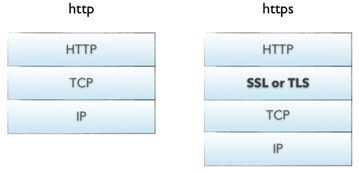
管理的可扩展性 (在任何地方的任何人都可以立即进行安全通信)。

适应性 (能够支持当前最知名的安全方法)。

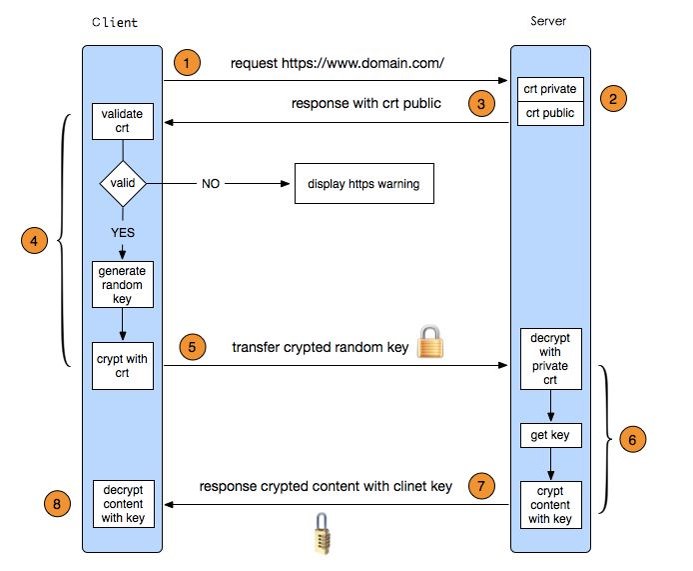
在社会上的可行性 (满足社会的政治文化需要)。

1、HTTP 协议（HyperText Transfer Protocol，超文本传输协议）：是客户端浏览器或其他程序与Web服务器之间的应用层通信协议 。

2、HTTPS 协议（HyperText Transfer Protocol over Secure Socket Layer）：可以理解为HTTP+SSL/TLS， 即 HTTP 下加入 SSL 层，HTTPS 的安全基础是 SSL，因此加密的详细内容就需要 SSL，用于安全的 HTTP 数据传输。



**ngx\_http\_ssl\_module模块为实现 HTTPS提供了必要的支持。此模块不是在默认情况下生成的, 在安装nginx时使用--with-http\_ssl\_module配置参数启用它。此模块需要OpenSSL库。**



##### 解读SSL的工作流程

1.浏览器发送一个https的请求给服务器；

2.服务器要有一套数字证书，可以自己制作（后面的操作就是阿铭自己制作的证书），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，这套证书其实就是一对公钥和私钥；

3.服务器会把公钥传输给客户端；

4.客户端（浏览器）收到公钥后，会验证其是否合法有效，无效会有警告提醒，有效则会生成一串随机数，并用收到的公钥加密；

5.客户端把加密后的随机字符串传输给服务器；

6.服务器收到加密随机字符串后，先用私钥解密（公钥加密，私钥解密），获取到这一串随机数后，再用这串随机字符串加密传输的数据（该加密为对称加密，所谓对称加密，就是将数据和私钥也就是这个随机字符串>通过某种算法混合在一起，这样除非知道私钥，否则无法获取数据内容）；

7.服务器把加密后的数据传输给客户端；

8.客户端收到数据后，再用自己的私钥（也就是那个随机字符串）解密；

**openssl是由三部分组成：**

libcrpto :通用加密库

libssl:TSL/SSL功能的实现，基于会话的，实现了身份认证，数据机密性和会话完整性的 TSL/SSL 库

openssl:提供的命令行工具，多用途命令工具，横拟实现私有证书颁发机构；命令行工具是通过多种子命令实现openssl的相应功能

openssl version/ rpm -qa openssl 检查版本

**创建私钥与公钥信息、生成自签署的证书、相关配置文件参数设定**

### 示例配置

为了减轻处理器负载，建议

1)设置工作进程的数量等于处理器的数量；

2)启用保持连接；

3)启用共享会话缓存；

4)禁用内置的会话缓存；

5)建议增加会话超时时间 （默认为5分钟）。

worker\_processes auto;

http {

server {

listen 443 ssl;

keepalive\_timeout 70;

ssl\_protocols TLSv1 TLSv1.1 TLSv1.2;

ssl\_ciphers AES128-SHA:AES256-SHA:RC4-SHA:DES-CBC3-SHA:RC4-MD5;

ssl\_certificate /usr/local/nginx/conf/cert.pem;

ssl\_certificate\_key /usr/local/nginx/conf/cert.key;

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

ssl\_session\_timeout 10m;

}

### 1.7.2 SSL模块指令说明

#### ****1.7.2.1 ssl****

Syntax: ssl on | off;

Default: ssl off;

Context: http, server

为指定的虚拟服务器启用HTTPS协议，默认关闭。

#### ****1.7.2.2 ssl\_buffer\_size****

Syntax: ssl\_buffer\_size size;

Default: ssl\_buffer\_size 16k;

Context: http, server

# This directive appeared in version 1.5.9.

该指令用于设置用于发送数据的缓冲区的大小。

默认情况下，缓冲区大小为16k，这对应于发送大响应时的最小开销。为了将发送第一个字节的时间减少，可以使用较小的值，例如：

ssl\_buffer\_size 4k;

#### ****1.7.2.3 ssl\_certificate****

Syntax: ssl\_certificate file;

Default: —

Context: http, server

指定虚拟主机的 PEM 格式的**证书文件**路径 。如果除了主证书外,还要指定中间证书, 则应按以下顺序在同一文件中指定它们: 主证书首先出现, 然后是中间证书。PEM 格式的密钥可以放在同一个文件中。

从1.11.0版本开始，可以多次使用该指令来加载不同类型的证书，例如RSA和ECDSA：

server {

listen 443 ssl;

server\_name example.com;

ssl\_certificate example.com.rsa.crt;

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

ssl\_certificate example.com.ecdsa.crt;

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

...

}

**注意：**只有OpenSSL 1.0.2及以上版本，支持加载不同类型的证书。对于较旧的版本，只能使用同一类型的单个证书。

**注意：使用nginx配置HTTPS多虚拟主机时，不同的主机要监听不同的地址，否则在初次访问时，SSL连接在浏览器发送HTTPs请求之前建立，而nginx不知道请求的服务器的名称。因此，它只能提供默认的服务器证书。将会导致业务的异常。详情参照：**<http://nginx.org/en/docs/http/configuring_https_servers.html#name_based_https_servers>

#### ****1.7.2.4 ssl\_certificate\_key****

Syntax: ssl\_certificate\_key file;

Default: —

Context: http, server

指定虚拟主机的 PEM 格式的**密钥文件**存放路径 。

在1.7.9版本engine:name:id 可以指定替代密钥文件，该指令能够从OpenSSL中加载指定ID的密钥 name。

#### ****1.7.2.5 ssl\_ciphers****

Syntax: ssl\_ciphers ciphers;

Default: ssl\_ciphers HIGH:!aNULL:!MD5;

Context: http, server

指定证书的加密方式。加密方式要是openssl可以识别的方式，例如：

ssl\_ciphers ALL:!aNULL:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP;

所有的加密方式可以在系统中通过openssl ciphers命令查看。

选用加密方式时要注意不同版本之间的兼容性问题，详情参考<http://nginx.org/en/docs/http/configuring_https_servers.html#compatibility>

#### ****1.7.2.6 ssl\_client\_certificate****

Syntax: ssl\_client\_certificate file;

Default: —

Context: http, server

如果启用了ssl\_stapling，则需要指定PEM格式的可信CA证书文件路径，用于验证客户端证书和OCSP(Online Certificate Status Protocol，在线证书状态协议)响应。

注意:使用该参数时证书列表将被发送给客户。如果不需要，可以使用ssl\_trusted\_certificate 指令。

#### ****1.7.2.7 ssl\_crl****

Syntax: ssl\_crl file;

Default: —

Context: http, server

# This directive appeared in version 0.8.7.

指定用于验证客户端证书的 PEM 格式指定具有吊销证书 (CRL) 的文件 。

#### ****1.7.2.8 ssl\_dhparam****

Syntax: ssl\_dhparam file;

Default: —

Context: http, server

# This directive appeared in version 0.7.2.

指定DHE加密方式的DH证书文件位置。

#### ****1.7.2.9 ssl\_ecdh\_curve****

Syntax: ssl\_ecdh\_curve curve;

Default:

ssl\_ecdh\_curve auto;

Context: http, server

# This directive appeared in versions 1.1.0 and 1.0.6.

指定一个curve用于ECDHE密码。

当使用的OpenSSL 1.0.2或更高版本时，可以指定多curve（1.11.0），例如：

ssl\_ecdh\_curve prime256v1:secp384r1;

特殊值auto（1.11.0）指示nginx在使用OpenSSL 1.0.2或更高版本时使用OpenSSL库中内置的列表为prime256v1，或使用旧版本。

在版本1.11.0 之前, 默认curve为prime256v1。

#### ****1.7.2.10**** ****ssl\_password\_file****

Syntax: ssl\_password\_file file;

Default: —

Context: http, server

# This directive appeared in version 1.7.3.

为密钥指定一个密码文件 , 其中每个口令都在单独的行上指定密码。密码在加载密钥时依次尝试。

示例：

http {

ssl\_password\_file /etc/keys/global.pass;

...

server {

server\_name www1.example.com;

ssl\_certificate\_key /etc/keys/first.key;

}

server {

server\_name www2.example.com;

# named pipe can also be used instead of a file

ssl\_password\_file /etc/keys/fifo;

ssl\_certificate\_key /etc/keys/second.key;

}

}

#### ****1.7.2.11**** ****ssl\_prefer\_server\_ciphers****

Syntax: ssl\_prefer\_server\_ciphers on | off;

Default: ssl\_prefer\_server\_ciphers off;

Context: http, server

指定在使用 SSLv3 和 TLS 协议时, 服务器密码应优先于客户端密码。

#### ****1.7.2.12**** ****ssl\_protocols****

Syntax: ssl\_protocols [SSLv2] [SSLv3] [TLSv1] [TLSv1.1] [TLSv1.2] [TLSv1.3];

Default: ssl\_protocols TLSv1 TLSv1.1 TLSv1.2;

Context: http, server

启用指定的SSL协议。

说明：

TLSv1.1和 TLSv1.2参数 (1.1.13、1.0.12) 仅在使用 OpenSSL 1.0.1 或更高时才起作用。

TLSv1.3参数 (1.13.0) 仅在使用用 TLSv1.3 仅在使用 OpenSSL 1.1.1 时有效。

#### ****1.7.2.13**** ****ssl\_session\_cache****

Syntax: ssl\_session\_cache off | none | [builtin[:size]] [shared:name:size];

Default: ssl\_session\_cache none;

Context: http, server

设置存储会话参数的高速缓存的类型和大小。缓存可以是以下任何一种类型：

| 类型 | 类型说明 |
| --- | --- |
| off | 严格禁止使用会话缓存: nginx 会明确告诉客户端会话可能无法重用。 |
| none | 不允许使用会话缓存: nginx 告诉客户端会话可以重用, 但实际上不会将会话参数存储在缓存中。 |
| builtin | 建立 OpenSSL 的缓存：仅由一个工作进程使用。缓存大小在会话中指定。如果未给定大小, 则默认为20480个会话。 注意：使用内置缓存会导致产生内存碎片。 |
| shared | 所有工作进程之间共享的缓存：缓存大小以字节为单位指定，一兆字节可以存储大约4000个会话。每个共享缓存都应具有名称。可以在多个虚拟服务器中使用同名的缓存。 |

两种不同的缓存类型都可以同时使用, 例如:

ssl\_session\_cache builtin:1000 shared:SSL:10m;

但是, 在没有内置缓存的情况下使用共享缓存才会更有效。

#### ****1.7.2.14**** ****ssl\_session\_ticket\_key****

Syntax: ssl\_session\_ticket\_key file;

Default: —

Context: http, server

# This directive appeared in version 1.5.7.

指定用于加密和解密TLS会话令牌的密钥文件存放位置。如果需要在多个服务器之间共享相同的密钥,则需要使用该指令。默认情况下, 使用随机生成的密钥。

如果指定了多个密钥, 则仅使用第一个密钥对 TLS 会话进行加密。密钥对时可以进行轮换的，例如：

ssl\_session\_ticket\_key current.key;

ssl\_session\_ticket\_key previous.key;

该文件必须包含80或48字节的随机数, 可以使用以下命令进行创建：

openssl rand 80 > ticket.key

根据文件大小的不同, AES256 (对于80字节密钥、1.11.8) 或 AES128 (对于48字节密钥) 用于加密。

#### ****1.7.2.15**** ****ssl\_session\_tickets****

Syntax: ssl\_session\_tickets on | off;

Default: ssl\_session\_tickets on;

Context: http, server

# This directive appeared in version 1.5.9.

通过TLS session tickets启用或禁用会话恢复 。详情参考：<https://tools.ietf.org/html/rfc5077>

#### ****1.7.2.16**** ****ssl\_session\_timeout****

Syntax: ssl\_session\_timeout time;

Default: ssl\_session\_timeout 5m;

Context: http, server

指定客户端可以重用会话参数的时间。

#### ****1.7.2.17**** ****ssl\_stapling****

Syntax: ssl\_stapling on | off;

Default: ssl\_stapling off;

Context: http, server

# This directive appeared in version 1.3.7.

启用或禁用服务器对OSCP的响应，详情参考：<https://tools.ietf.org/html/rfc4366#section-3.6>

配置示例：

ssl\_stapling on;

resolver 192.0.2.1;

要使 OCSP 正常工作, 需要知道服务器证书颁发者的证书

如果ssl\_certificate文件不包含中间证书, 则应在ssl\_trusted\_certificate文件中显示服务器证书颁发者的证书。

对于 OCSP 响应方主机名的解析, 还应指定解析程序指令。

#### ****1.7.2.18**** ****ssl\_stapling\_file****

Syntax: ssl\_stapling\_file file;

Default: —

Context: http, server

# This directive appeared in version 1.3.7.

设置后，将从指定的file中采取订阅的 ocsp 响应, 而不是查询在服务器证书中指定的 ocsp 应答器。

该文件应该是由“ openssl ocsp”命令产生的DER格式。

#### ****1.7.2.19**** ****ssl\_stapling\_responder****

Syntax: ssl\_stapling\_responder url;

Default: —

Context: http, server

# This directive appeared in version 1.3.7.

重写在 "颁发机构信息访问" 证书扩展中指定的 OCSP 响应程序的 URL。

参考文献：<https://tools.ietf.org/html/rfc5280#section-4.2.2.1>

仅支持 "http://" OCSP 响应程序，例如:

ssl\_stapling\_responder http://ocsp.example.com/;

#### ****1.7.2.20**** ****ssl\_stapling\_verify****

Syntax: ssl\_stapling\_verify on | off;

Default: ssl\_stapling\_verify off;

Context: http, server

# This directive appeared in version 1.3.7.

启用或禁用服务器对 OCSP 响应的验证。

要进行验证, 应使用ssl\_trusted\_certificate指令将服务器证书颁发者、根证书和所有中间证书的证书配置为受信任。

#### ****1.7.2.21**** ****ssl\_trusted\_certificate****

Syntax: ssl\_trusted\_certificate file;

Default: —

Context: http, server

# This directive appeared in version 1.3.7.

指定一个 PEM 格式的文件 , 其中带有用于验证 ssl\_stapling启用时使用的校验证书和 OCSP 进行验证。

与ssl\_client\_certificate设置的证书正好相反, 这些证书的列表将不会发送到客户端。

#### ****1.7.2.22**** ****ssl\_verify\_client****

Syntax: ssl\_verify\_client on | off | optional | optional\_no\_ca;

Default: ssl\_verify\_client off;

Context: http, server

启用对客户端证书的验证。验证结果存储在 $ssl\_client\_verify变量中。

可选参数optional（0.8.7+）请求客户端证书、验证证书是否存在。

可选参数optional\_no\_ca (1.3.8, 1.2.5)请求客户端证书而不需要将签署由受信任的CA证书。这适用于nginx外部的服务运行实际证书验证的情况。证书的内容可以通过变量$ssl\_client\_cert进行访问 。

#### ****1.7.2.23**** ****ssl\_verify\_depth****

Syntax: ssl\_verify\_depth number;

Default: ssl\_verify\_depth 1;

Context: http, server

设置客户端证书的验证深度。

### 1.7.3 ngx\_http\_ssl\_module模块中常见错误处理

在ngx\_http\_ssl\_module模块支持几种非标准错误代码, 可用于使用error\_page指令进行重定向。

| 错误代码 | 产生原因 |
| --- | --- |
| 495 | an error has occurred during the client certificate verification; 在客户端证书验证过程中发生错误; |
| 496 | a client has not presented the required certificate; 客户未出示所需证书; |
| 497 | a regular request has been sent to the HTTPS port. 常规请求已发送到HTTPS端口。 |

在对请求进行了完全分析并且变量 (如$request\_uri、 $uri、 $args和其他项) 可用之后, 重定向发生。

### 1.7.4 ngx\_http\_ssl\_module模块中嵌入变量

ngx\_http\_ssl\_module模块支持多个嵌入变量：

| 变量 | 变量说明 |
| --- | --- |
| $ssl\_cipher | 返回用于已建立的 SSL 连接的密码字符串; |
| $ssl\_ciphers | 返回客户端支持的密码列表（1.11.7）。 |
| $ssl\_client\_escaped\_cert | 以 PEM 格式 (urlencoded) 返回已建立的 SSL 连接 (1.13.5) 的客户端证书。 |
| $ssl\_client\_cert | 以建立的SSL连接的PEM格式返回客户端证书，除第一行之外的每一行都加上制表符; 这是为了在 proxy\_set\_header指令中使用; 该变量已被弃用，$ssl\_client\_escaped\_cert应该使用该变量。 |
| $ssl\_client\_fingerprint | 为建立的SSL连接（1.7.1）返回客户端证书的SHA1指纹; |
| $ssl\_client\_i\_dn | 根据RFC 2253 (1.11.6) 返回已建立的 SSL 连接的客户端证书的 "颁发者 DN" 字符串; |
| $ssl\_client\_i\_dn\_legacy | 返回已建立的 SSL 连接的客户端证书的 "颁发者 DN" 字符串; 在版本1.11.6 之前, 变量名为$ssl\_client\_i\_dn. |
| $ssl\_client\_raw\_cert | 以 PEM 格式返回已建立的 SSL 连接的客户端证书; |
| $ssl\_client\_s\_dn | 根据RFC 2253（1.11.6），为建立的SSL连接返回客户端证书的“主题DN”字符串 ; |
| $ssl\_client\_s\_dn\_legacy | 为建立的SSL连接返回客户端证书的“主题DN”字符串; 在版本1.11.6之前，变量名是$ssl\_client\_s\_dn |
| $ssl\_client\_serial | 为建立的SSL连接返回客户端证书的序列号; |
| $ssl\_client\_v\_end | 返回客户端证书的结束日期（1.11.7）; |
| $ssl\_client\_v\_remain | 返回客户端证书过期的天数（1.11.7）; |
| $ssl\_client\_v\_start | 返回客户端证书的开始日期（1.11.7）; |
| $ssl\_client\_verify | 如果证书不存在，则 返回客户端证书验证的结果：“ SUCCESS”，“ FAILED:reason”和“ NONE”; 在版本1.11.7之前，“ FAILED”结果不包含reason字符串。 |
| $ssl\_curves | 返回由客户端 (1.11.7) 支持的curves 仅当使用 OpenSSL 版本1.0.2 或更高时, 才支持该变量。 |
| $ssl\_protocol | 返回已建立的 SSL 连接的协议; |
| $ssl\_server\_name | 通过SNI （1.7.0）返回请求的服务器名称 ; |
| $ssl\_session\_id | 返回建立的SSL连接的会话标识符; |
| $ssl\_session\_reused | 如果重新使用了 SSL 会话, 则返回 "r", 否则为 "." (1.5.11)。 |

**举例1：**

server {

listen 443;

server\_name localhost;

ssl on;

root html;

index index.html index.htm;

ssl\_certificate cert/21.pem;

ssl\_certificate\_key cert/21.key;

ssl\_session\_timeout 5m;

ssl\_ciphers ECDHE-RSA-AES128-GCM-SHA256:ECDHE:ECDH:AES:HIGH:!NULL:!aNULL:!MD5:!ADH:!RC4;

ssl\_protocols TLSv1 TLSv1.1 TLSv1.2;

ssl\_prefer\_server\_ciphers on;

location / {

root html;

index index.html index.htm;

}

}

**举例2：**

upstream www\_server\_pools {

server 10.0.0.8:443;

}

server {

listen 443;

server\_name xxx.com;

ssl on;

ssl\_certificate /application/nginx/conf/key/1\_blog.nmtui.com\_bundle.crt;

ssl\_certificate\_key /application/nginx/conf/key/2\_blog.nmtui.com.key;

ssl\_session\_timeout 5m;

ssl\_protocols TLSv1 TLSv1.1 TLSv1.2;

ssl\_ciphers ECDHE-RSA-AES128-GCM-SHA256:HIGH:!aNULL:!MD5:!RC4:!DHE;

ssl\_prefer\_server\_ciphers on;

ssl\_session\_cache shared:SSL:10m; ####添加缓存

location / {

proxy\_pass https://server\_pools;

}

}

server {

listen 80;

server\_name xxx.com;

rewrite ^(.\*)$ https://$host$1 permanent;

}

# nginx proxy cache

缓存服务器（通常）是“前端”;它从客户端收到初始HTTP请求。然后它会处理请求本身（如果它具有所请求资源的新缓存副本），或者将请求传递给Origin Server来实现。

如果请求发送到Origin Server，则由Cache Server读取源服务器的响应头，以确定响应是缓存还是简单传递。

**一些较大的Web应用程序除了缓存服务器之外还使用负载平衡器，从而导致高度分层的基础架构。**

缓存服务器的职责：

确定HTTP请求是否接受缓存响应，并且缓存中有一个新项目可以响应

如果请求不应被缓存，或者缓存的项目是否过期，则向源服务器发送HTTP请求

响应来自其缓存或源服务器的HTTP响应为适当的。

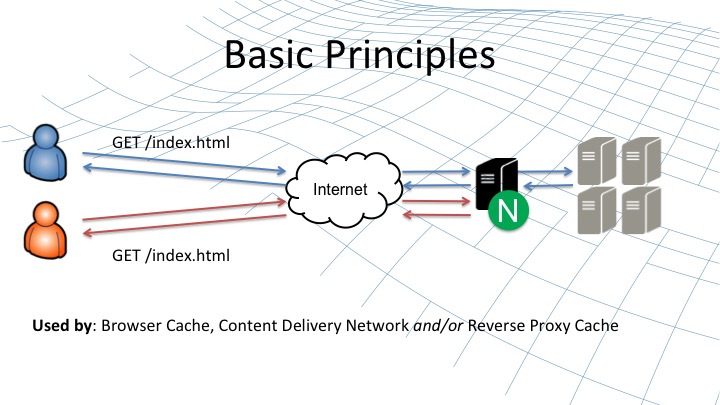
最后说下**客户端**。客户端可以拥有自己的本地（私有）缓存 - 例如每个浏览器都有一个本地缓存。我们的浏览器可能会缓存一个响应本身（通常是图像，CSS和JS文件），因此，如果静态文件在其本地缓存中已经有新版本，那么浏览器根本不会向缓存服务器发送请求。

实现本地缓存的客户端具有以下职责：

发送请求

缓存响应

决定从本地缓存中提取请求或发出HTTP请求以检索它们

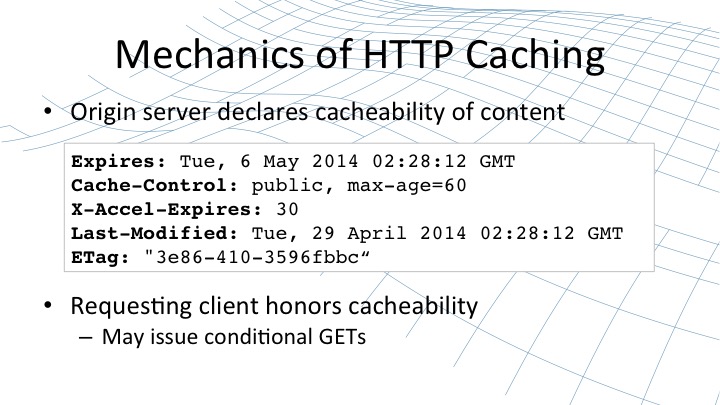


website流程图（nginx为proxy）

当第一个用户（蓝）访问网站，他的请求首先会到NGINX PROXY SERVER,随后NGNIX发往后端服务器（灰），后端会将请求的响应首先发往NGINX，由其返回给用户（蓝色），如果这个响应是可以缓存的，则NGINX会保留一份响应的副本，当其他用户（橙色）发起相同的request请求时，NGINX会根据request请求的内容是否存在于缓存中，来直接返回给用户，不经过后端。这个场景同样适用于浏览器，CDN，用于缓存静态资源。

## HTTP缓存机制

后端服务器会通过响应包头来定义缓存特性：



图二：Origin Server定义的缓存特性

当然，缓存服务器可以通过设置一些参数来忽略或者重写后端服务器的缓存特性，但后端服务器的缓存特性也是极其重要的。

**.Expires: 最原始的配置策略，即设置过期时间，但使用效率低下，目前绝大部分已经被Cache-Control（有兴趣的可以去看下http1.0和http1.1）；**

**.Cache-Control：定义缓存资源属性是private或者是public，并且设置缓存多久后过期，本例中，属性为public，60秒过期；**

**.X-Accel-Expires: 只有nginx能识别的缓存特性header，优先级大于上面两个header，可以设置此header，在nginx侧来重新定义缓存特性；**

**.Etag和Last-Modified是捆绑生成的: 有些场景下，你希望client端的浏览器长时间缓存，而缓存服务器只短时间缓存文件，以至于当后端服务器更新后，缓存服务器会及时同步，我们就可以使用最后两个header，Last-Modified表示最后修改时间，并声明一个ETag(哈希值)，做为缓存内容的标签，具有唯一性;客户端访问请求带有If‑Modified‑Since或者If‑None‑Match header，并申明自己的客户端带有静态缓存文件,以及文件修改日期和ETag值，如果服务器端的版本和Etag值与客户端一致，则服务端会直接返回304 not modified，这个验证流程是非常快的，并且节省网络带宽；**

**.如果Cache-Control设置为public，则客户端不会去验证资源的有效性，将会一直使用直到过期，同时public也代表资源可以被缓存在web proxy中；**

**.如果Cache-Control包含must-revalidate，则客户端每一次访问请求资源都会去验证缓存是否有更新；**

## NGINX都会缓存哪些资源

首先看下nginx proxy cache最基本的配置：

**proxy\_cache\_path** /path/to/cache levels=1:2 keys\_zone=my\_cache:10m max\_size=10g inactive=60m use\_temp\_path=off;  
server {  
 set $upstream [http://ip:port](https://link.jianshu.com?t=http:/ip:port)  
 location / {  
 **proxy\_cache** my\_cache;  
 **proxy\_pass** $upstream; }  
}

配置项说明：

1./path/to/cache : 本地路径，缓存文件存放地址；

2.levels : 默认所有缓存文件都放在同一个/path/to/cache下，从而影响缓存的性能，大部分场景推荐使用2级目录来存储缓存文件；

3.key\_zone : 在共享内存中设置一块存储区域来存放缓存的key和metadata（类似使用次数），这样nginx可以快速判断一个request是否命中或者未命中缓存，1m可以存储8000个key，10m可以存储80000个key；

4.max\_size : 最大cache空间，如果不指定，会使用掉所有disk space，当达到配额后，会删除最少使用的cache文件；

5.inactive : 未被访问文件在缓存中保留时间，本配置中如果60分钟未被访问则不论状态是否为expired，缓存控制程序会删掉文件，默认为10分钟；“需要注意的是，inactive和expired配置项的含义是不同的，expired只是缓存过期，但不会被删除，inactive是删除指定时间内未被访问的缓存文件”；

6.use\_temp\_path : 如果为off，则nginx会将缓存文件直接写入指定的cache文件中，而不是使用temp\_path存储，official建议为off，避免文件在不同文件系统中不必要的拷贝；

7.proxy\_cache : 启用proxy cache，指定key\_zone;

## 缓存和代理中常用的配置项

上文讲述了如何配置最基础的proxy cache，接下来，会对常用的高级配置项进行梳理。

**proxy\_no\_cache** string;

Default: —  
Context: http , server , location  
config example:  
**proxy\_no\_cache** $cookie\_nocache $arg\_nocache $arg\_comment;  
**proxy\_no\_cache** $http\_pragma $http\_authorization;

$cookie\_nocache $arg\_nocache...皆为变量，可以根据你访问的匹配策略来设置，其值只有2类，0和非0;  
访问匹配策略例如：  
if ($request\_uri ~ ^/(login|register|password\/reset)/) { set $cookie\_nocache 1; }  
如果在此链式配置中，只要有一个值不为0，则不会cache；例如：proxy\_no\_cache $cookie\_nocache(0) $arg\_nocache(1) $arg\_comment(0)，不会被cache。`  
注：一般会配合proxy\_cache\_bypass共同使用；

**proxy\_cache\_bypass** string;

Default: —  
Context: http , server , location  
config example:  
**proxy\_cache\_bypass** $cookie\_nocache $arg\_nocache$arg\_comment;  
**proxy\_cache\_bypass** $http\_pragma $http\_authorization;

定义在哪些情况下不从cache读取，直接从backend获取资源；配置方式同**proxy\_no\_cache**。

**proxy\_cache\_key** string;

Default: proxy\_cache\_key $scheme$proxy\_host$request\_uri;  
Context: http, server, location

自定义cache key，例如：  
proxy\_cache\_key "$host$request\_uri $cookie\_user";  
默认值为：  
proxy\_cache\_key $scheme$proxy\_host$uri$is\_args$args;

**proxy\_cache\_methods** GET| HEAD|POST...;

Default: proxy\_cache\_methods GET HEAD;  
Context: http, server, location

指定客户端那些方法被缓存，默认为GET|HEAD。

**proxy\_cache\_purge** string ...;

Default: —  
Context: http, server, location  
config example：  
proxy\_cache\_path /data/nginx/cache keys\_zone=cache\_zone:10m;  
map $request\_method $purge\_method {  
 PURGE 1;  
default 0;  
}  
server {  
 location / {  
 proxy\_pass [http://backend](https://link.jianshu.com?t=http:/backend);  
 proxy\_cache cache\_zone;  
 proxy\_cache\_key $uri;  
 proxy\_cache\_purge $purge\_method;  
}}

定义缓存清除场景，同**proxy\_no\_cache**，**proxy\_cache\_bypass**链式配置方式，只要又一个不为0，则清除对应的cache key则会被清除，并返回204 response。注意，这里是删除内存中的cache key，而不是disk上的cache文件！！！disk的cache文件是由inactive控制；  
当purege request的cache key以通配符＊结束时，所有匹配到通配符的cache入口的cachekey都会被删除。

**proxy\_cache\_valid** \*[code...] time \*;

Default: —  
Context: http, server, location

设置不同相应码的缓存时间，当不指定响应码的时候，例如  
proxy\_cache\_valid 5m;  
只对响应码为200，301，302的访问请求资源设置缓存时间，此外可以个性化定制，例如：  
proxy\_cache\_valid 200 302 10m; proxy\_cache\_valid 301 1h; proxy\_cache\_valid 404 1m; proxy\_cache\_valid any 1m;  
此外，还可以在相应header里设置优先级更高的缓存有效时间：

1. “X-Accel-Expires”,设置响应的缓存过期时间，以秒为单位；0为不缓存；
2. 如果没有设置“X-Accel-Expires” header，则关于缓存的配置策略可能会在“Expires”或者“Cache-Control” header中；
3. 如果header含有“Set-Cookie”,则响应不会被缓存，类似的配置可以在“proxy\_ignore\_header”中可见；
4. header包含“Vary”并且设置为“＊”，则请求不会被缓存，如果“Vary”有具体的值，则对应的请求会被缓存；

**proxy\_ignore\_headers** field；

Default: —  
Context: http, server, location

不缓存包含在field的响应header，可以设置的值有：“X-Accel-Redirect”, “X-Accel-Expires”, “X-Accel-Limit-Rate”，“X-Accel-Buffering”, “X-Accel-Charset”, “Expires”, “Cache-Control”, “Set-Cookie” (0.8.44), and “Vary”。  
如果上述的header field没有设置为忽略，则header filed中有“X-Accel-Expires”, “Expires”, “Cache-Control”, “Set-Cookie”, and “Vary”的话，响应会被缓存。

**proxy\_pass\_headers** field；

**proxy\_hide\_headers** field；

Default: —  
Context: http, server, location

举例1：

1. proxy\_cache\_path /path/to/cache levels=1:2 keys\_zone=my\_cache:10m max\_size=10g inactive=60m
2. use\_temp\_path=off;
3. server {
4. ...
5. location / {
6. proxy\_cache my\_cache;
7. proxy\_cache\_revalidate on;
8. proxy\_cache\_min\_uses 3;
9. proxy\_cache\_use\_stale error timeout updating http\_500 http\_502 http\_503 http\_504;
10. proxy\_cache\_lock on;
11. proxy\_pass http:*//my\_upstream;*
12. }
13. }

举例2

http{

proxy\_connect\_timeout 10;

proxy\_read\_timeout 180;

proxy\_send\_timeout 5;

proxy\_buffer\_size 16k;

proxy\_buffers 4 32k;

proxy\_busy\_buffers\_size 96k;

proxy\_temp\_file\_write\_size 96k;

proxy\_temp\_path /tmp/temp\_dir;

proxy\_cache\_path /tmp/cache levels=1:2 keys\_zone=cache\_one:100m inactive=1d max\_size=10g;

server {

listen 80 default\_server;

server\_name localhost;

root /mnt/blog/;

location / {

}

#要缓存文件的后缀，可以在以下设置。

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

proxy\_pass http://ip地址:90;

proxy\_redirect off;

proxy\_set\_header Host $host;

proxy\_cache cache\_one;

proxy\_cache\_valid 200 302 24h;

proxy\_cache\_valid 301 30d;

proxy\_cache\_valid any 5m;

expires 90d;

add\_header wall "hey!guys!give me a star.";

}

}

# 无nginx缓存的blog端口

server {

listen 90;

server\_name localhost;

root /mnt/blog/;

location / {

}

}

}

**proxy\_connect\_timeout 服务器连接的超时时间**

**proxy\_read\_timeout 连接成功后,等候后端服务器响应时间**

**proxy\_send\_timeout 后端服务器数据回传时间**

**proxy\_buffer\_size 缓冲区的大小**

**proxy\_buffers 每个连接设置缓冲区的数量为number，每块缓冲区的大小为size**

**proxy\_busy\_buffers\_size 开启缓冲响应的功能以后，在没有读到全部响应的情况下，写缓冲到达一定大小时，nginx一定会向客户端发送响应，直到缓冲小于此值。**

**proxy\_temp\_file\_write\_size 设置nginx每次写数据到临时文件的size(大小)限制**

**proxy\_temp\_path 从后端服务器接收的临时文件的存放路径**

**proxy\_cache\_path 设置缓存的路径和其他参数。被缓存的数据如果在inactive参数（当前为1天）指定的时间内未被访问，就会被从缓存中移除**

proxy\_pass nginx缓存里拿不到资源，向该地址转发请求，拿到新的资源，并进行缓存

proxy\_redirect 设置后端服务器“Location”响应头和“Refresh”响应头的替换文本

proxy\_set\_header 允许重新定义或者添加发往后端服务器的请求头

proxy\_cache 指定用于页面缓存的共享内存，对应http层设置的keys\_zone

proxy\_cache\_valid 为不同的响应状态码设置不同的缓存时间

expires 缓存时间

## 静态资源缓存

## 使用 expires 语句

location ~\* \.(jpg|jpeg|png|gif|ico|css|js)$ { # 匹配图片、js、css 文件

expires 15d; # 设置过期时间为 15 天

}

## 其它时间单位

ms # 毫秒

s # 秒

m # 分钟

h # 小时

d # 天

w # 星期

M # 月(30d)

y # 年(365d)

# 也可以进行组合，如 1h30m 表示 1 小时 30 分钟，1y6M 表示 1 年 6 个月

## upstream缓存

proxy\_cache\_path /usr/share/nginx/html/cache levels=1:2 keys\_zone=cache:10m max\_size=10g inactive=60m

use\_temp\_path=off;

server {

...

location / {

...

proxy\_cache cache;

proxy\_cache\_use\_stale error timeout invalid\_header http\_500 http\_502 http\_503 http\_504;

proxy\_cache\_valid 200 304 12h;

proxy\_cache\_valid any 10m;

proxy\_cache\_key $host$uri$is\_args$args;

add\_header Nginx-Cache "$upstream\_cache\_status";

}

}

# levels=1:2 两级层次结构的目录

# keys\_zone 缓存区命名

# max\_size 缓存上限值

# inactive 缓存过期时长

# use\_temp\_path=off 将在缓存这些文件时将它们写入同一个目录下

# proxy\_cache\_use\_stale 当出现相应的错误时，不返回错误信息，而是返回已缓存的内容

# proxy\_cache\_valid 指定状态的缓存过期时长

# proxy\_cache\_key 根据key映射成一个hash值，然后保存至本地文件

**举例**

# 当后端web服务器上也配置多个虚拟主机时，需要用该header来区分反向代理哪个主机名。

proxy\_set\_header Host $host;

# 通过$remote\_addr变量获取前端客户真实IP地址。

proxy\_set\_header X-Real-IP $remote\_addr;

# 通过$remote\_addr变量获取前端客户真实IP地址。

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

# 设置缓冲区

proxy\_buffering on;

# 设置后端服务器的响应头大小，是响应头的缓冲区。

proxy\_buffer\_size 4k;

# 设置缓冲区的数量和大小，nginx从代理的后端服务器获取的响应信息会放置到缓冲区。

proxy\_buffers 8 32k;

#此设置表示nginx会在没有完全读完后端响应的时候就开始向客户端传送数据，所以它会划出一部分缓冲区来专门向客户端传送数据。建议为proxy\_buffers中单个缓冲区大小的2倍)

proxy\_busy\_buffers\_size 64K；

# 指定当响应内容大于proxy\_buffers指定的缓冲区时, 写入硬盘的临时文件的大小。

proxy\_max\_temp\_file\_size 1024m;

# 一次访问能写入的临时文件的大小，默认是proxy\_buffer\_size和proxy\_buffers中设置的缓冲区大小的2倍。

proxy\_temp\_file\_write\_size 128k;

#

proxy\_request\_buffering on;

#

proxy\_ignore\_headers Set-Cookie;

# 表示与后端服务器连接的超时时间

proxy\_connect\_timeout 60s;

# 表示后端服务器的数据回传时间

proxy\_send\_timeout 300;

# 设置nginx从代理的后端服务器获取信息的时间

proxy\_read\_timeout 300s;