1. SSL站点

server {

listen 443 ssl;

server\_name www.yinchengku.com

ssl on;

ssl\_certificate /root/www.yinchengku.com\_bundle.crt;

ssl\_certificate\_key /root/www.yinchengku.com.key;

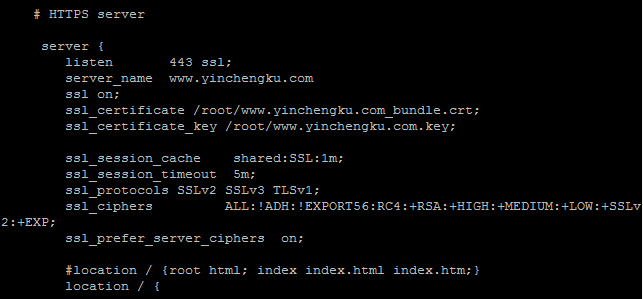
ssl\_session\_cache shared:SSL:1m;

ssl\_session\_timeout 5m;

ssl\_protocols SSLv2 SSLv3 TLSv1;

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

ssl\_prefer\_server\_ciphers on;



SSL证书，安装NameCheap免费SSL 参考http://www.linuxde.net/2013/10/15338.html

1. proxy\_cache静态缓存

首先明确一点，proxy\_cache是对URL路径hash，不能缓存jsp动态页面的session！

在http{ }模块中添加：

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



在location{ }模块中添加

proxy\_redirect off;

proxy\_set\_header Host $host;

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

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

proxy\_connect\_timeout 180;

proxy\_send\_timeout 180;

proxy\_read\_timeout 180;

proxy\_buffer\_size 128k;

proxy\_buffers 4 128k;

proxy\_busy\_buffers\_size 128k;

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

proxy\_cache imgcache;

proxy\_cache\_valid 200 302 1d;

proxy\_cache\_valid 404 10m;

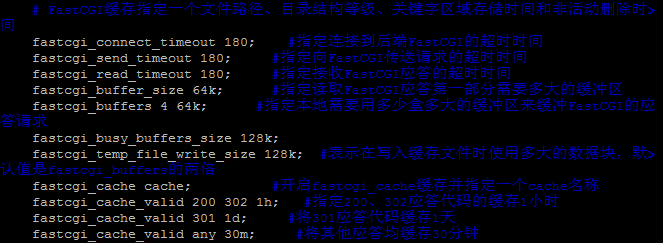
proxy\_cache\_valid any 1h;

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

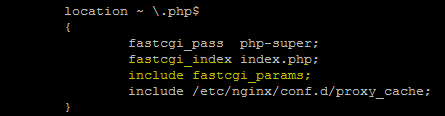
add\_header Nginx-Cache "$upstream\_cache\_status";

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

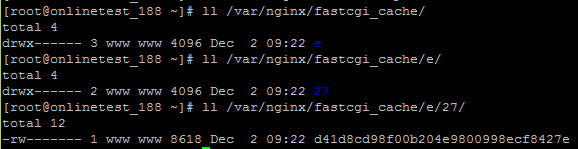
将proxy\_cache写入单独的/etc/nginx/conf.d/proxy\_cache文件中



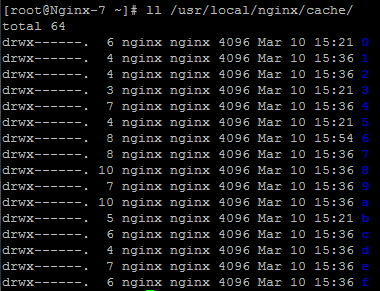
在server模块中引用proxy\_cache这个文件



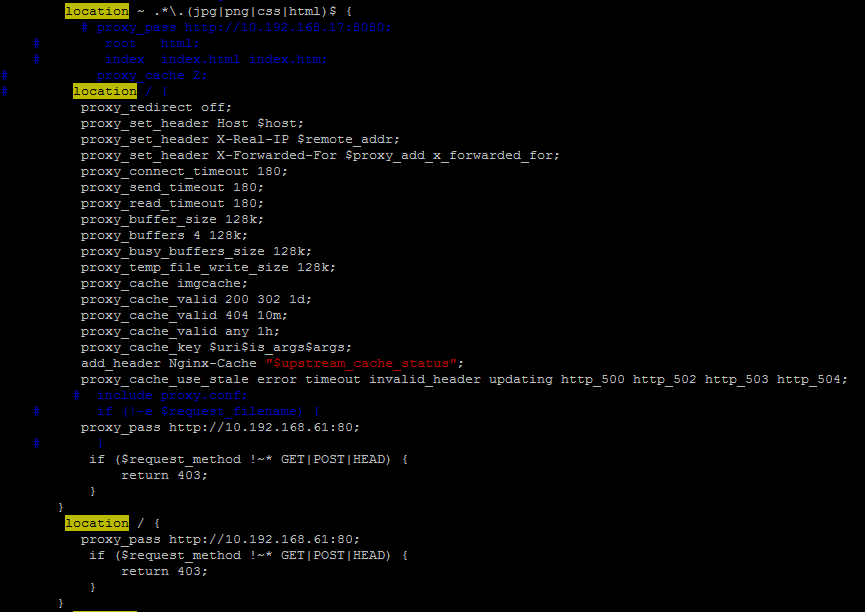
配置生效



缓存路径为/usr/local/nginx/cache



在生产环境中，只能使用proxy\_cache缓存jpg、png、css、html静态文件，那次我用来缓存jsp动态文件导致已登录用户点击其他页面丢失session



1. proxy\_pass负载均衡
2. 转发后台tomcat：

在location{ }模块中添加

proxy\_pass http://10.192.168.17:8080;



1. 多后台负载均衡：

在location{ }模块中添加

proxy\_pass http://tomcat;



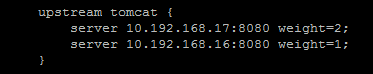
在http{ }模块中添加：

upstream tomcat {

server 10.192.168.17:8080 weight=2;

server 10.192.168.16:8080 weight=1;

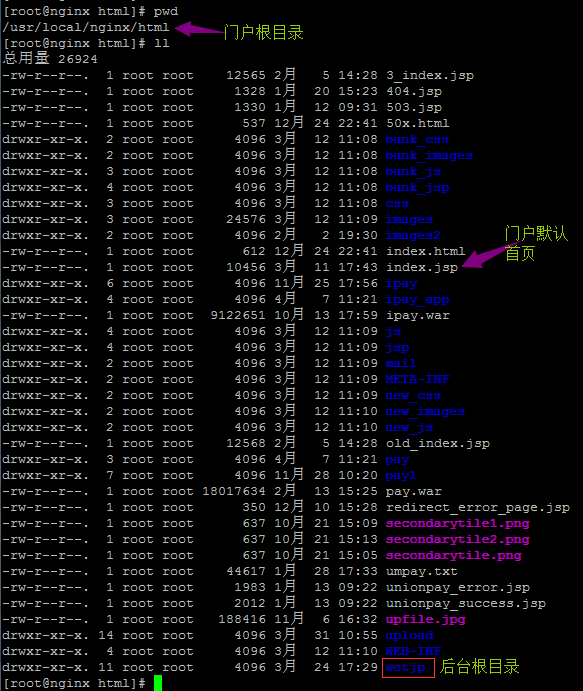
}

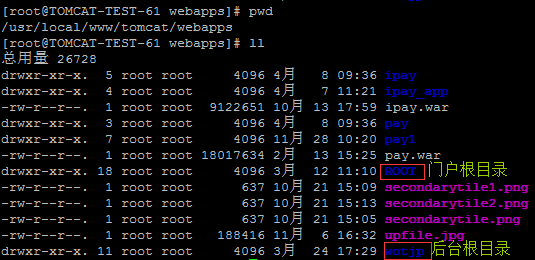


1. jpg、jsp动静分离

环境：nginx项目根目录：/usr/local/nginx/html；tomcat项目根目录：/usr/local/tomcat/webapps/ipay。

注意：由于tomcat上，项目的根目录ipay和后台根目录wotjp在同一级目录/usr/local/tomcat/webapps下，而nginx的项目根和后台根目录必须不属于父子文件夹！





Nginx.conf配置：

代码示例：

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

root html;

if ($request\_method !~\* GET|POST|HEAD) {

return 403;

}

}

location / {

index index.jsp;

proxy\_pass http://10.192.168.61:80;

if ($request\_method !~\* GET|POST|HEAD) {

return 403;

}

}

location ~ .\*\.(jsp|do)?$ {

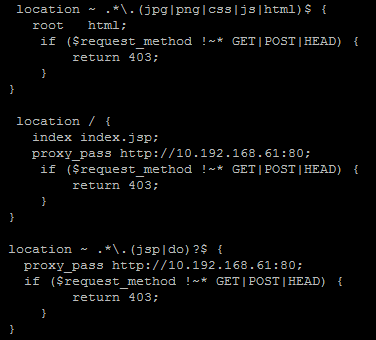
proxy\_pass http://10.192.168.61:80;

if ($request\_method !~\* GET|POST|HEAD) {

return 403;

}

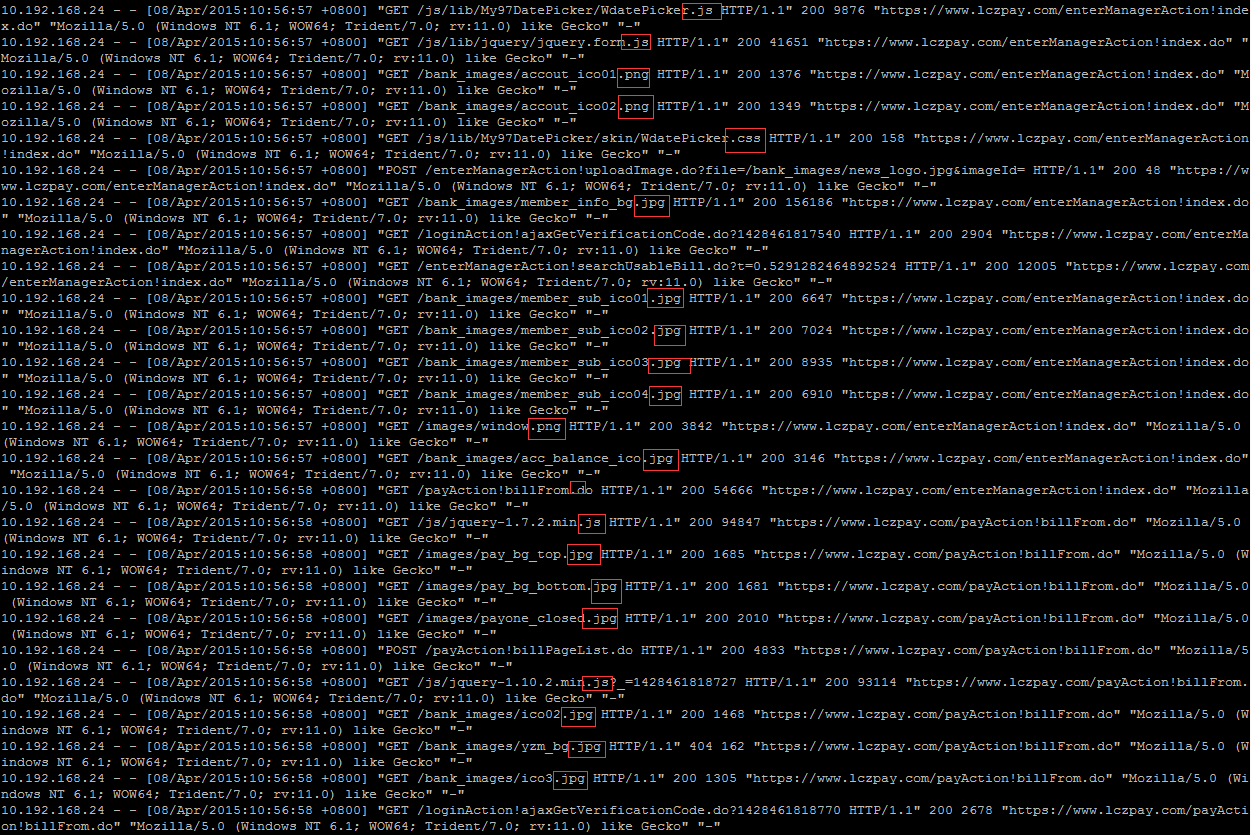
}



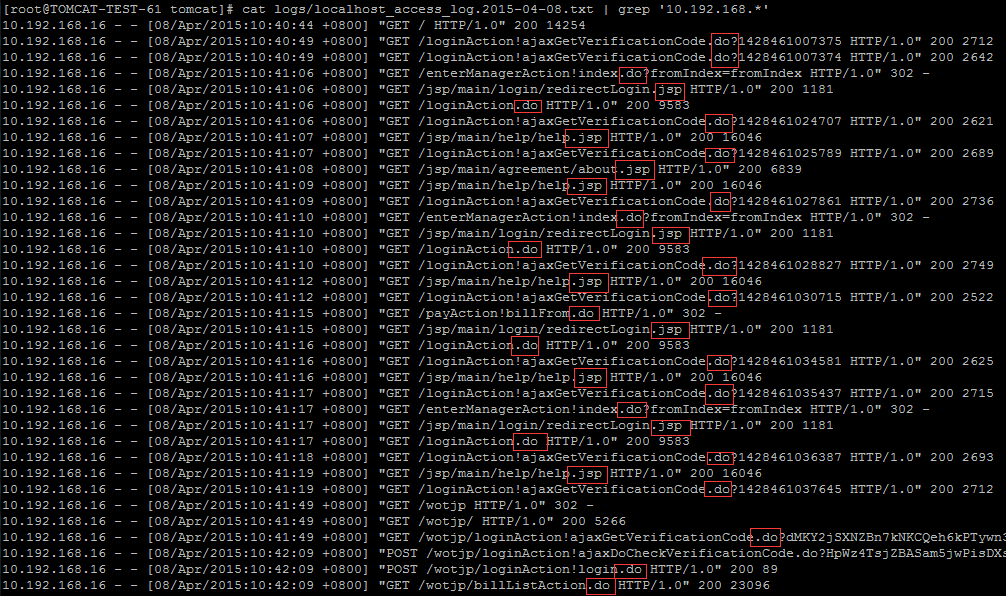
注意：“localtion / ”的index、proxy\_pass都不能少！

效果示例：

Nginx处理了jpg、png、html、css、js文件：



Tomcat只处理了.jsp、.do文件：



注意啦：nginx动静分离后proxy\_cache就失效啦！

5. 使用ab做压力测试

yum安装httpd-tools，使用ab –c -n命令测试：





下载这个index.jsp文件的速率是148kb/s。

1. nginx代理多域名、多端口

架构说明：



代码如下：

server {

listen 80;

server\_name demo.yinchengku.com;

location ~ ^.\*/upload/.\*\.(php|php5|php3|php4|jsp)$

{

deny all;

}

location / {

# proxy\_pass http://10.192.168.61;

rewrite ^/(.\*) http://demo.lczpay.com$1 permanent;

}

if ($request\_method !~\* GET|POST|HEAD) {

return 403;

}

}

server {

listen 80;

server\_name demo.lczpay.com;

location ~ ^.\*/upload/.\*\.(php|php5|php3|php4|jsp)$

{

deny all;

}

location / {

proxy\_pass http://10.192.168.61;

}

if ($request\_method !~\* GET|POST|HEAD) {

return 403;

}

}

server {

listen 80;

server\_name mail.yinchengku.com;

location / {

#proxy\_pass https://10.192.168.4:443;

rewrite ^/(.\*) https://mail.yinchengku.com$1 permanent;

}

error\_page 500 502 503 504 /50x.html;

location = /50x.html {

root html;

}

}

server {

listen 80;

server\_name ldap.yinchengku.com;

location / {

rewrite ^/(.\*) https://ldap.yinchengku.com$1 permanent;

}

}

server {

listen 80;

server\_name php.tengda.com;

#charset koi8-r;

#access\_log logs/host.access.log main;

location / {

root html;

index index.php index.html index.htm;

}

error\_page 500 502 503 504 /50x.html;

location = /50x.html {

root html;

}

#

location ~ \.php$ {

root html;

fastcgi\_pass 127.0.0.1:9000;

fastcgi\_index index.php;

fastcgi\_param SCRIPT\_FILENAME $document\_root$fastcgi\_script\_name;

fastcgi\_param SCRIPT\_NAME $fastcgi\_script\_name;

# fastcgi\_param SCRIPT\_FILENAME /scripts$fastcgi\_script\_name;

include fastcgi\_params;

}

}

# HTTPS server

#

server {

listen 443 ssl;

server\_name mail.yinchengku.com;

ssl on;

ssl\_certificate /root/iRedMail.crt;

ssl\_certificate\_key /root/iRedMail.key;

ssl\_protocols TLSv1 TLSv1.1 TLSv1.2;

location / {

proxy\_set\_header Host mail.yinchengku.com;

proxy\_pass https://10.192.168.4:443;

proxy\_redirect off;

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

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

}

}

server {

listen 443 ssl;

server\_name ldap.yinchengku.com;

ssl on;

ssl\_certificate /root/iRedMail.crt;

ssl\_certificate\_key /root/iRedMail.key;

ssl\_protocols TLSv1 TLSv1.1 TLSv1.2;

location / {

proxy\_set\_header Host ldap.yinchengku.com;

proxy\_pass https://10.192.168.4:443;

proxy\_redirect off;

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

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

}

}

}

目的：demo.lczpay.com和demo.yincehngku.com均通过http方式访问，mail.yinchengku.com和ldap.yinchengku.com只能通过https方式访问。

7.同域名多项目

情景：同一域名下，前台、后台项目放在同一目录下，默认打开网站https://www.lczpay.com/，访问的是前台，其根目录是/usr/local/tomcat7/webapps/ROOT/；打开<https://www.lczpay.com/wotjp/>，访问的是后台，其根目录是/usr/local/tomcat7/webapps/wotjp/。

nginx.conf配置如下：

location / {

root /usr/local/tomcat7/webapps/ROOT;

index index.jsp;

}

location /wotjp {

root /usr/local/tomcat7/webapps;

index jsp/main/login/login.jsp;

}

location ~ .\*\.(jsp|do)$ {

proxy\_pass http://127.0.0.1:8080;

if ($request\_method !~\* GET|POST|HEAD) {

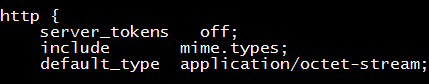
return 403;

}

}

8.隐藏版本号

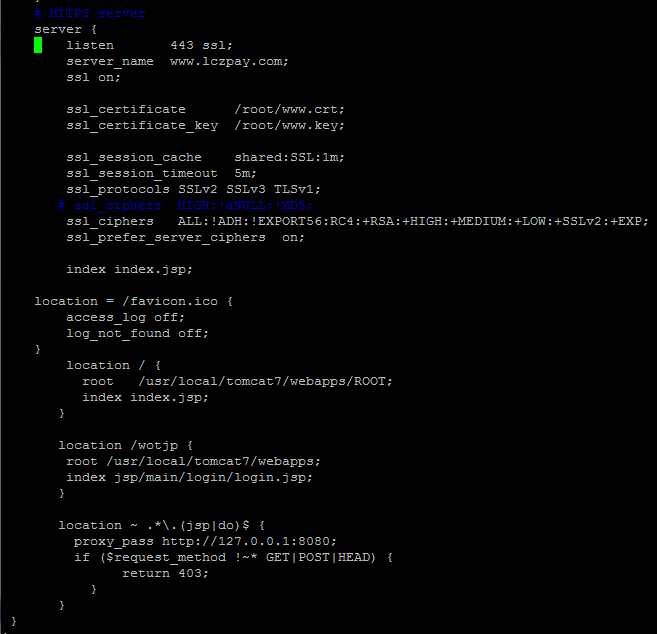
在http模块中加入server\_tokens off;



效果：



注：后台首页不是/usr/local/tomcat7/webapps/wotjp/index.php，而是/usr/local/tomcat7/webapps/wotjp/jsp/main/login/login.jsp。



访问首页和后台都很成功，而且也做到了动静分离。

9. gzip

配置如下：

gzip on;

gzip\_static on;

gzip\_min\_length 1k;

gzip\_buffers 8 16k;

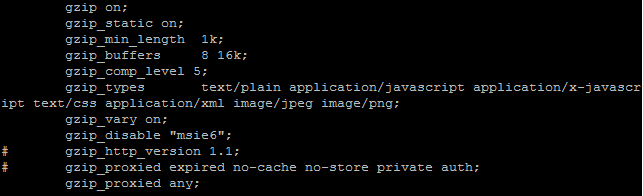
gzip\_comp\_level 5;

gzip\_types text/plain application/javascript application/x-javascript text/css application/xml image/jpeg image/png;

gzip\_vary on;

gzip\_disable "msie6";

gzip\_proxied any;



其中，gzip\_min\_length指定原文件的体积大于多少才压缩；gzip\_comp\_level指定压缩等级，为1-9，等级越高越消耗CPU；gzip\_types指定压缩的文件类型。

gzip\_vary 指令的意义：Enables or disables inserting the “Vary: Accept-Encoding” response header field if the directives gzip, gzip\_static, or gunzip are active.

假设CDN/缓存 向Nginx 请求资源，Nginx 在响应头里插入“Vary: Accept-Encoding” ， CDN/缓存会把 Accept-Encoding 的值 作为 传入参数 来做hash 计算，这样 Accept-Encoding 不一样，就代表 「资源」是不一样的(即使文件是一样的)。它的好处是可以根据客户端请求的 Accept-Encoding 来返回不一样的资源，包括是不是压缩 和 各种压缩格式。如果没有“Vary: Accept-Encoding”，CDN/缓存 可能给客户端错误的资源，比如客户端不支持压缩，却收到gzip压缩的资源，客户端就解析不了了。

和http头有关系，加个vary头，给代理服务器用的，有的浏览器支持压缩，有的不支持，所以避免浪费不支持的也压缩，所以根据客户端的HTTP头来判断，是否需要压缩

gzip\_disable 指令接受一个正则表达式，当请求头中的 UserAgent 字段满足这个正则时，响应不会启用 GZip，这是为了解决在某些浏览器启用 GZip 带来的问题。特别地，指令值 msie6 等价于 MSIE [4-6]\.

10. HTTP/1.1 的缓存机制

HTTP/1.1 的缓存机制稍微有点复杂，这里简单介绍下：

首先，服务端可以通过响应头里的 Last-Modified（最后修改时间） 或者 ETag（内容特征） 标记实体。浏览器会存下这些标记，并在下次请求时带上 If-Modified-Since: 上次 Last-Modified 的内容 或 If-None-Match: 上次 ETag 的内容，询问服务端资源是否过期。如果服务端发现并没有过期，直接返回一个状态码为 304、正文为空的响应，告知浏览器使用本地缓存；如果资源有更新，服务端返回状态码 200、新的 Last-Modified、Etag 和正文。这个过程被称之为 HTTP 的协商缓存，通常也叫做弱缓存。

可以看到协商缓存并不会节省连接数，但是在缓存生效时，会大幅减小传输内容（304 响应没有正文，一般只有几百字节）。另外为什么有两个响应头都可以用来实现协商缓存呢？这是因为一开始用的 Last-Modified 有两个问题：1）只能精确到秒，1 秒内的多次变化反映不出来；2）时间采用绝对值，如果服务端 / 客户端时间不对都可能导致缓存失效 在轮询的负载均衡算法中，如果各机器读到的文件修改时间不一致，有缓存无故失效和缓存不更新的风险。HTTP/1.1 并没有规定 ETag 的生成规则，而一般实现者都是对资源内容做摘要，能解决前面两个问题。

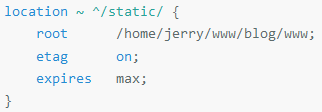
另外一种缓存机制是服务端通过响应头告诉浏览器，在什么时间之前（Expires）或在多长时间之内（Cache-Control: Max-age=xxx），不要再请求服务器了。这个机制我们通常称之为 HTTP 的强缓存。

一旦资源命中强缓存规则后，再次访问完全没有 HTTP 请求（Chrome 开发者工具的 Network 面板依然会显示请求，但是会注明 from cache；Firefox 的 firebug 也类似，会注明 BFCache），这会大幅提升性能。所以我们一般会对 CSS、JS、图片等资源使用强缓存，而入口文件（HTML）一般使用协商缓存或不缓存，这样可以通过修改入口文件中对强缓存资源的引入 URL 来达到即时更新的目的。

这里也解释下为什么有了 Expire，还要有 Cache-Control。也有两个原因：1）Cache-Control 功能更强大，对缓存的控制能力更强；2）Cache-Control 采用的 max-age 是相对时间，不受服务端 / 客户端时间不对的影响。

另外关于浏览器的刷新（F5 / cmd + r）和强刷（Ctrl + F5 / shift + cmd +r）：普通刷新会使用协商缓存，忽略强缓存；强刷会忽略浏览器所有缓存（并且请求头会携带 Cache-Control:no-cache 和 Pragma:no-cache，用来通知所有中间节点忽略缓存）。只有从地址栏或收藏夹输入网址、点击链接等情况下，浏览器才会使用强缓存。

默认情况下，Nginx 对于静态资源都会输出 Last-Modified，而 ETag、Expire 和 Cache-Control 则需要自己配置：



11. proxy\_set\_header

一般来说，gzip压缩是否启用，除了服务器支持外，客户端也要支持。当客户端发送Accept-Encoding:gzip这个request header，服务器即认为其能接受gzip压缩，就响应一个Content-Encoding:gzip，并发送压缩内容；假如客户端没有发送 Accept-Encoding，那么服务器就把源代码老老实实地打印出去。

但能不能让客户端无论有没有发送Accept-Encoding，服务器都会发送压缩内容呢？

gzip压缩不在前端nginx进行，前端主要是用来强制修改request header，即写上：

proxy\_set\_header Accept-Encoding 'gzip';

这样，后台的nginx无论如何都将接收到Accept- Encoding:gzip，而不管客户端有没有发。

12. Last-Modified和Etags

基础知识

　　1) 什么是”Last-Modified”?

　　在浏览器第一次请求某一个URL时，服务器端的返回状态会是200，内容是你请求的资源，同时有一个Last-Modified的属性标记此文件在服务期端最后被修改的时间，格式类似这样：

　　Last-Modified: Fri, 12 May 2006 18:53:33 GMT

　　客户端第二次请求此URL时，根据 HTTP 协议的规定，浏览器会向服务器传送 If-Modified-Since 报头，询问该时间之后文件是否有被修改过：

　　If-Modified-Since: Fri, 12 May 2006 18:53:33 GMT

　　如果服务器端的资源没有变化，则自动返回 HTTP 304 （Not Changed.）状态码，内容为空，这样就节省了传输数据量。当服务器端代码发生改变或者重启服务器时，则重新发出资源，返回和第一次请求时类似。从而保证不向客户端重复发出资源，也保证当服务器有变化时，客户端能够得到最新的资源。

　　2) 什么是”Etag”?

　　HTTP 协议规格说明定义ETag为“被请求变量的实体值” （参见 —— 章节 14.19）。 另一种说法是，ETag是一个可以与Web资源关联的记号（token）。典型的Web资源可以一个Web页，但也可能是JSON或XML文档。服务器单独负责判断记号是什么及其含义，并在HTTP响应头中将其传送到客户端，以下是服务器端返回的格式：

　　ETag: "50b1c1d4f775c61:df3"

　　客户端的查询更新格式是这样的：

　　If-None-Match: W/"50b1c1d4f775c61:df3"

　　如果ETag没改变，则返回状态304然后不返回，这也和Last-Modified一样。本人测试Etag主要在断点下载时比较有用。

　　Last-Modified和Etags如何帮助提高性能?

　　聪明的开发者会把Last-Modified 和ETags请求的http报头一起使用，这样可利用客户端（例如浏览器）的缓存。因为服务器首先产生 Last-Modified/Etag标记，服务器可在稍后使用它来判断页面是否已经被修改。本质上，客户端通过将该记号传回服务器要求服务器验证其（客户端）缓存。

过程如下:

1,客户端请求一个页面（A）。

2,服务器返回页面A，并在给A加上一个Last-Modified/ETag。

3,客户端展现该页面，并将页面连同Last-Modified/ETag一起缓存。

4,客户再次请求页面A，并将上次请求时服务器返回的Last-Modified/ETag一起传递给服务器。

5,服务器检查该Last-Modified或ETag，并判断出该页面自上次客户端请求之后还未被修改，直接返回响应304和一个空的响应体。

正确使用Etag和Expires标识处理，可以使得页面更加有效被Cache。

在客户端通过浏览器发出第一次请求某一个URL时，根据 HTTP 协议的规定，浏览器会向服务器传送报头(Http Request Header)，服务器端响应同时记录相关属性标记(Http Reponse Header)，服务器端的返回状态会是200，格式类似如下：

HTTP/1.1 200 OK

Date: Tue, 03 Mar 2009 04:58:40 GMT

Content-Type: image/jpeg

Content-Length: 83185

Last-Modified: Mon, 22 Nov 2010 16:29:24 GMT

Cache-Control: max-age=2592000

Expires: Thu, 02 Apr 2009 05:14:08 GMT

Etag: "xok.la-961AA72-4CEA99B4415628″

客户端第二次请求此URL时，根据 HTTP 协议的规定，浏览器会向服务器传送报头(Http Request Header)，服务器端响应并记录相关记录属性标记文件没有发生改动,服务器端返回304，直接从缓存中读取：

HTTP/1.x 304 Not Modified

Date: Tue, 03 Mar 2009 05:03:56 GMT

Content-Type: image/jpeg

Content-Length: 83185

Last-Modified: Mon, 22 Nov 2010 16:29:24 GMT

Cache-Control: max-age=2592000

Expires: Thu, 02 Apr 2009 05:14:08 GMT

Etag: "xok.la-961AA72-4CEA99B4415628″

其中Last-Modified、Expires和Etag是标记页面缓存标识

一、Last-Modified、Expires和Etag相关工作原理

1、Last-Modified

在浏览器第一次请求某一个URL时，服务器端的返回状态会是200，内容是你请求的资源，同时有一个Last-Modified的属性标记 (Http Reponse Header)此文件在服务期端最后被修改的时间，格式类似这样：

Last-Modified: Mon, 22 Nov 2010 16:29:24 GMT客户端第二次请求此URL时，根据 HTTP 协议的规定，浏览器会向服务器传送 If-Modified-Since 报头(Http Request Header)，询问该时间之后文件是否有被修改过：

If-Modified-Since: Mon, 22 Nov 2010 16:29:24 GMT如果服务器端的资源没有变化，则自动返回 HTTP 304 （NotChanged.）状态码，内容为空，这样就节省了传输数据量。当服务器端代码发生改变或者重启服务器时，则重新发出资源，返回和第一次请求时类 似。从而保证不向客户端重复发出资源，也保证当服务器有变化时，客户端能够得到最新的资源。

注：如果If-Modified-Since的时间比服务器当前时间(当前的请求时间request\_time)还晚，会认为是个非法请求

2、Etag工作原理

HTTP 协议规格说明定义ETag为”被请求变量的实体标记” （参见14.19）。简单点即服务器响应时给请求URL标记，并在HTTP响应头中将其传送到客户端，类似服务器端返回的格式：

Etag: "xok.la-961AA72-4CEA99B4415628″客户端的查询更新格式是这样的：

If-None-Match: "xok.la-961AA72-4CEA99B4415628″如果ETag没改变，则返回状态304。

即:在客户端发出请求 后，Http Reponse Header中包含 Etag: “xok.la-961AA72-4CEA99B4415628″

标识，等于告诉Client端，你拿到的这个的资源有表示 ID：xok.la-961AA72-4CEA99B4415628。当下次需要发Request索要同一个 URI的时候，浏览器同时发出一个If-None-Match报头( Http RequestHeader)此时包头中信息包含上次访问得到的Etag: “xok.la-961AA72-4CEA99B4415628″标识。

If-None-Match: "xok.la-961AA72-4CEA99B4415628",这样，Client端等于Cache了两份，服务器端就会比对2者的etag。如果 If- None-Match为False，不返回200，返回304 (Not Modified) Response。

3、Expires

给出的 日期/时间后，被响应认为是过时。如Expires: Thu, 02 Apr 2009 05:14:08 GMT

需和Last-Modified结合使用。用于控制请求文件的有效时间，当请求数据在有效期内时客 户端浏览器从缓存请求数据而不是服务器端. 当缓存中数据失效或过期，才决定从服务器更新数据。

4、Last-Modified和Expires

Last- Modified标识能够节省一点带宽，但是还是逃不掉发一个HTTP请求出去，而且要和Expires一起用。而Expires标识却使得浏览器干脆连 HTTP请求都不用发，比如当用户F5或者点击Refresh按钮的时候就算对于有Expires的URI，一样也会发一个HTTP请求出去，所 以，Last-Modified还是要用的，而 且要和Expires一起用。

5、 Etag和Expires

如果服务器端同时设置了 Etag和Expires 时，Etag原理同样，即与Last-Modified/Etag对应的HttpRequest Header:If-M+dified-Since和If-None-Match。我们可以看到这两个Header的值和WebServer发出的 Last-Modified,Etag值完全一样；在完全匹配If-Modified-Since和If-None-Match即检查完修改时间和 Etag之后，服务器才能返回304.

6、Last-Modified和Etag

Last-Modified 和ETags请求的http报头一起使用，服务器首先产生 Last-Modified/Etag标记，服务器可在稍后使用它来判断页面是否已经被修改，来决定文件是否继续缓存

过程如下:

1. 客户端请求一个页面（A）。

2. 服务器返回页面A，并在给A加上一个Last-Modified/ETag。

3. 客户端展现该页面，并将页面连同Last-Modified/ETag一起缓存。

4. 客户再次请求页面A，并将上次请求时服务器返回的Last-Modified/ETag一起传递给服务器。

5. 服务器检查该Last-Modified或ETag，并判断出该页面自上次客户端请求之后还未被修改，直接返回响应304和一个空的响应体。

注：

1、Last- Modified和Etag头都是由Web Server发出的Http Reponse Header，Web Server应该同时支持这两种头。

2、Web Server发送完Last-Modified/Etag头给客户端后，客户端会缓存这些头；

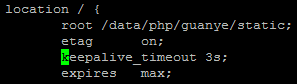
3、客户端再次发起相同页面的请求时，将分别发送与Last-Modified/Etag对应的Http RequestHeader:If-Modified-Since和If-None-Match。我们可以看到这两个Header的值和 WebServer发出的Last-Modified,Etag值完全一样；

4、 通过上述值到服务器端检查，判断文件是否继续缓存；

13.keep-alive

目前大部分浏览器都是用http1.1协议，也就是说默认都会发起Keep-Alive的连接请求了，所以是否能完成一个完整的Keep- Alive连接就看服务器设置情况。

Nginx只对静态文件的传输启用keepalive，配置很简单：



keepalive\_timeout 3s;表示keepalive时间是3秒。

对于upstream模块也可以使用keepalive

KeepAlive有TCP层、HTTP层两种。TCP是一种通信的方式，HTTP协议是一种事务协议。

TCP为什么要有KeepAlive？TCP协议通过一种巧妙的方式去解决这个问题，当超过一段时间之后，TCP自动发送一个数据为空的报文给对方，如果对方回应了这个报文，说明对方还在线，链接可以继续保持，如果对方没有报文返回，并且重试了多次之后则认为链接丢失，没有必要保持链接

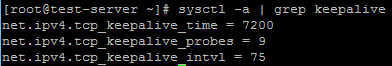
Linux Kernel有三个选项影响到KeepAlive的行为：

net.ipv4.tcpkeepaliveintvl = 75

net.ipv4.tcpkeepaliveprobes = 9

net.ipv4.tcpkeepalivetime = 7200

tcpkeepalivetime的单位是秒，表示TCP链接在多少秒之后没有数据报文传输启动探测报文; tcpkeepaliveintvl单位是也秒,表示前一个探测报文和后一个探测报文之间的时间间隔，tcpkeepaliveprobes表示探测的次数。



TCP socket也有三个选项和内核对应，通过setsockopt系统调用针对单独的socket进行设置：

TCPKEEPCNT: 覆盖 tcpkeepaliveprobes

TCPKEEPIDLE: 覆盖 tcpkeepalivetime

TCPKEEPINTVL: 覆盖 tcpkeepalive\_intvl

举个例子，以我的系统默认设置为例，kernel默认设置的tcpkeepalivetime是7200s, 如果我在应用程序中针对socket开启了KeepAlive,然后设置的TCP\_KEEPIDLE为60，那么TCP协议栈在发现TCP链接空闲了60s没有数据传输的时候就会发送第一个探测报文。

一个完整的HTTP事务，有链接的建立，请求的发送，响应接收，断开链接这四个过程,早期通过HTTP协议传输的数据以文本为主，一个请求可能就把所有要返回的数据取到，但是，现在要展现一张完整的页面需要很多个请求才能完成，如图片,JS,CSS等，如果每一个HTTP请求都需要新建并断开一个TCP，这个开销是完全没有必要的，开启HTTP Keep-Alive之后，能复用已有的TCP链接，当前一个请求已经响应完毕，服务器端没有立即关闭TCP链接，而是等待一段时间接收浏览器端可能发送过来的第二个请求，通常浏览器在第一个请求返回之后会立即发送第二个请求，如果某一时刻只能有一个链接，同一个TCP链接处理的请求越多，开启KeepAlive能节省的TCP建立和关闭的消耗就越多。当然通常会启用多个链接去从服务器器上请求资源，但是开启了Keep-Alive之后，仍然能加快资源的加载速度。HTTP/1.1之后默认开启Keep-Alive, 在HTTP的头域中增加Connection选项。当设置为Connection:keep-alive表示开启，设置为Connection:close表示关闭。实际上HTTP的KeepAlive写法是Keep-Alive，跟TCP的KeepAlive写法上也有不同。所以TCP KeepAlive和HTTP的Keep-Alive不是同一回事情。

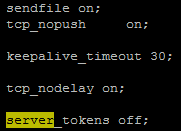
14.TCP 优化

配置如下：

sendfile on;

tcp\_nopush on;

tcp\_nodelay on;



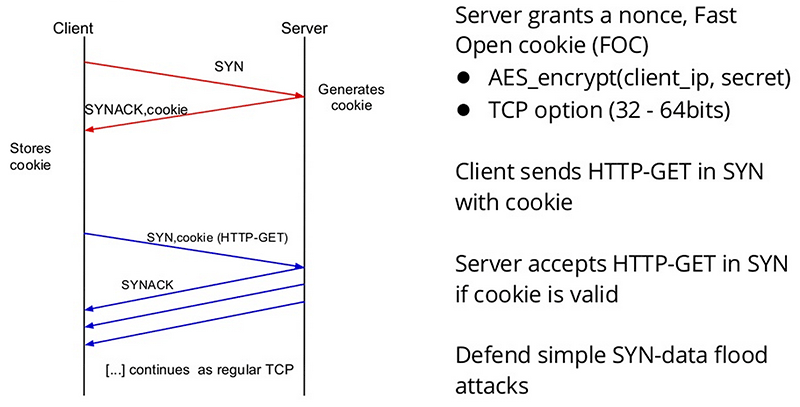
sendfile 是一个系统调用，直接在内核空间完成文件发送，不需要先 read 再 write，没有上下文切换开销。sendfile 配置可以提高 Nginx 静态资源托管效率。

TCP\_NOPUSH 是 FreeBSD 的一个 socket 选项，对应 Linux 的 TCP\_CORK，Nginx 里统一用 tcp\_nopush 来控制它，并且只有在启用了 sendfile 之后才生效。启用它之后，数据包会累计到一定大小之后才会发送，减小了额外开销，提高网络效率。

TCP\_NODELAY 也是一个 socket 选项，启用后会禁用 Nagle 算法，尽快发送数据，某些情况下可以节约 200ms（Nagle 算法原理是：在发出去的数据还未被确认之前，新生成的小数据先存起来，凑满一个 MSS 或者等到收到确认后再发送）。Nginx 只会针对处于 keep-alive 状态的 TCP 连接才会启用 tcp\_nodelay。

可以看到 TCP\_NOPUSH 是要等数据包累积到一定大小才发送，TCP\_NODELAY 是要尽快发送，二者相互矛盾。实际上，它们确实可以一起用，最终的效果是先填满包，再尽快发送。

还有一个 TCP 优化策略叫 TCP Fast Open（TFO），这里先介绍下，配置在后面贴出。TFO 的作用是用来优化 TCP 握手过程。客户端第一次建立连接还是要走三次握手，所不同的是客户端在第一个 SYN 会设置一个 Fast Open 标识，服务端会生成 Fast Open Cookie 并放在 SYN-ACK 里，然后客户端就可以把这个 Cookie 存起来供之后的 SYN 用。下面这个图形象地描述了这个过程：



现阶段只有 Linux、ChromeOS 和 Android 5.0 的 Chrome / Chromium 才支持 TFO，所以实际用途并不大

15. SPDY SSL

现阶段 Nginx 只支持 SPDY/3.1，需要http\_spdy\_module 和 http\_ssl\_module两个模块，配置如下：

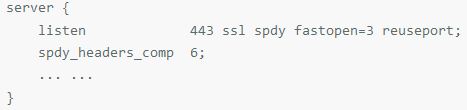
server {

listen 443 ssl spdy fastopen=3 reuseport;

spdy\_headers\_comp 6;

... ...

}



fastopen=3 用来开启前面介绍过的 TCP Fast Open 功能。3 代表最多只能有 3 个未经三次握手的 TCP 链接在排队。超过这个限制，服务端会退化到采用普通的 TCP 握手流程。这是为了减少资源耗尽攻击：TFO 可以在第一次 SYN 的时候发送 HTTP 请求，而服务端会校验 Fast Open Cookie（FOC），如果通过就开始处理请求。如果不加限制，恶意客户端可以利用合法的 FOC 发送大量请求耗光服务端资源。

reuseport 就是用来启用前面介绍过的 TCP SO\_REUSEPORT 选项的配置。

HTTPS 优化

server {

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

ssl\_session\_timeout 10m;

ssl\_session\_tickets on;

ssl\_stapling on;

ssl\_stapling\_verify on;

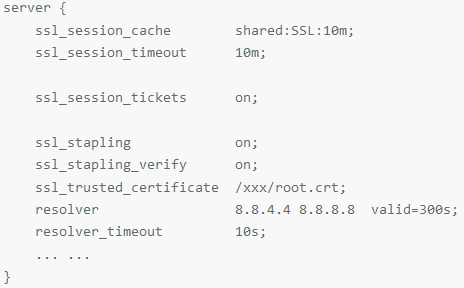
ssl\_trusted\_certificate /xxx/root.crt;

resolver 8.8.4.4 8.8.8.8 valid=300s;

resolver\_timeout 10s;

... ...

}

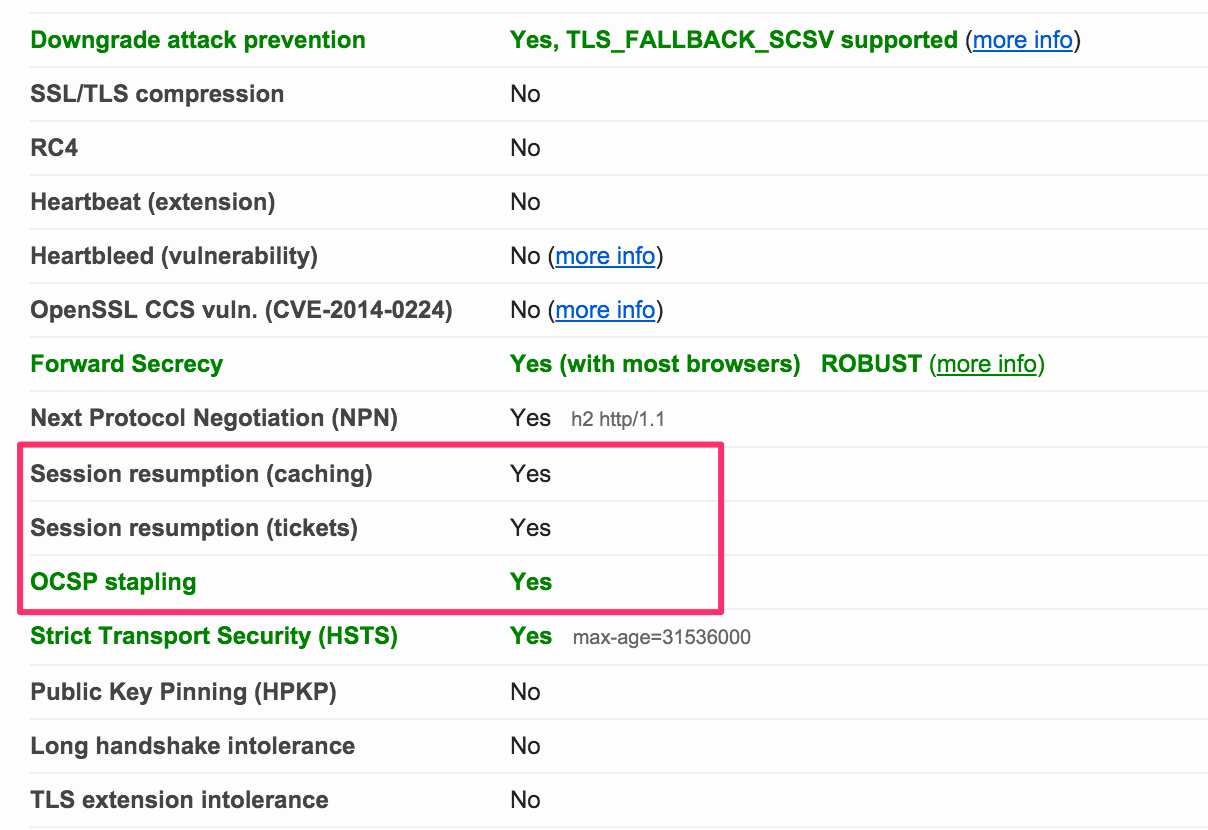


这部分配置就两部分内容：TLS 会话恢复和 OCSP stapling。

TLS 会话恢复的目的是为了简化 TLS 握手，有两种方案：Session Cache 和 Session Ticket。他们都是将之前握手的 Session 存起来供后续连接使用，所不同是 Cache 存在服务端，占用服务端资源；Ticket 存在客户端，不占用服务端资源。另外目前主流浏览器都支持 Session Cache，而 Session Ticket 的支持度一般。

ssl\_stapling 开始的几行用来配置 OCSP stapling 策略。浏览器可能会在建立 TLS 连接时在线验证证书有效性，从而阻塞 TLS 握手，拖慢整体速度。OCSP stapling 是一种优化措施，服务端通过它可以在证书链中封装证书颁发机构的 OCSP（Online Certificate Status Protocol）响应，从而让浏览器跳过在线查询。服务端获取 OCSP 一方面更快（因为服务端一般有更好的网络环境），另一方面可以更好地缓存。有关 OCSP stapling 的详细介绍，可以看这里。

这些策略设置好之后，可以通过 [Qualys SSL Server Test](https://www.ssllabs.com/ssltest/index.html) 这个工具来验证是否生效，



在给 Nginx 指定证书时，需要选择合适的证书链。因为浏览器在验证证书信任链时，会从站点证书开始，递归验证父证书，直至信任的根证书。这里涉及到两个问题：1）服务器证书是在握手期间发送的，由于 TCP 初始拥塞窗口的存在，如果证书太长很可能会产生额外的往返开销；2）如果服务端证书没包含中间证书，大部分浏览器可以正常工作，但会暂停验证并根据子证书指定的父证书 URL 自己获取中间证书。这个过程会产生额外的 DNS 解析、建立 TCP 连接等开销。配置服务端证书链的最佳实践是包含站点证书和中间证书两部分。有的证书提供商签出来的证书级别比较多，这会导致证书链变长，选择的时候需要特别注意。

ssl\_certificate /home/jerry/ssl/server.crt;

ssl\_certificate\_key /home/jerry/ssl/server.key;

ssl\_dhparam /home/jerry/ssl/dhparams.pem;

ssl\_ciphers ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA384:ECDHE-ECDSA-AES256-SHA:ECDHE-RSA-AES256-SHA:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA256:AES128-GCM-SHA256:AES256-GCM-SHA384:DES-CBC3-SHA;

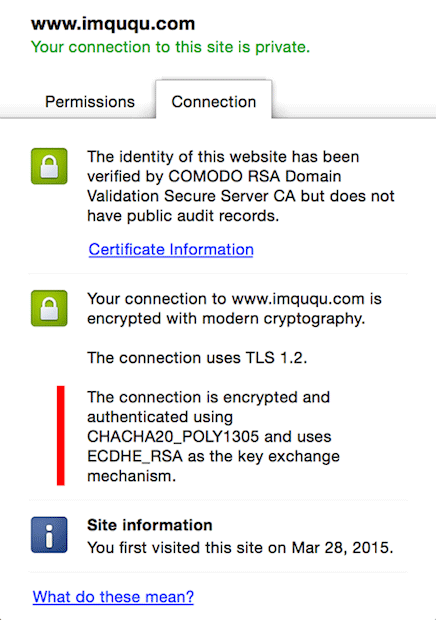
ssl\_prefer\_server\_ciphers on;

ssl\_protocols TLSv1 TLSv1.1 TLSv1.2;

ssl\_dhparam 的配置，可以参考这篇文章：[Guide to Deploying Diffie-Hellman for TLS](https://weakdh.org/sysadmin.html)。

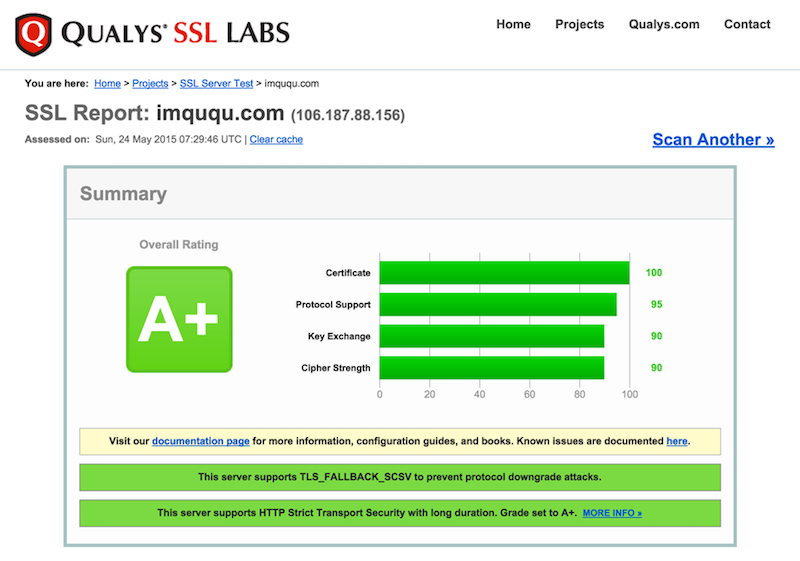
ssl\_ciphers 如何配置可以参考[这个网站](https://cipherli.st/)。我在 ssl\_ciphers 最开始加上了 CHACHA20，这是因为我的 Nginx 支持了 CHACHA20\_POLY1305 加密算法，这是由 Google 开发的新一代加密方式，它有两方面优势：更好的安全性和更好的性能（尤其是在移动和可穿戴设备上）

启用 CHACHA20\_POLY1305 最简单的方法是在编译 Nginx 时，使用 [LibreSSL](http://www.libressl.org/) 代替 OpenSSL。下面是用 Chrome 访问我的博客时，点击地址栏小锁显示的信息，可以看到加密方式使用的就是 CHACHA20\_POLY1305：



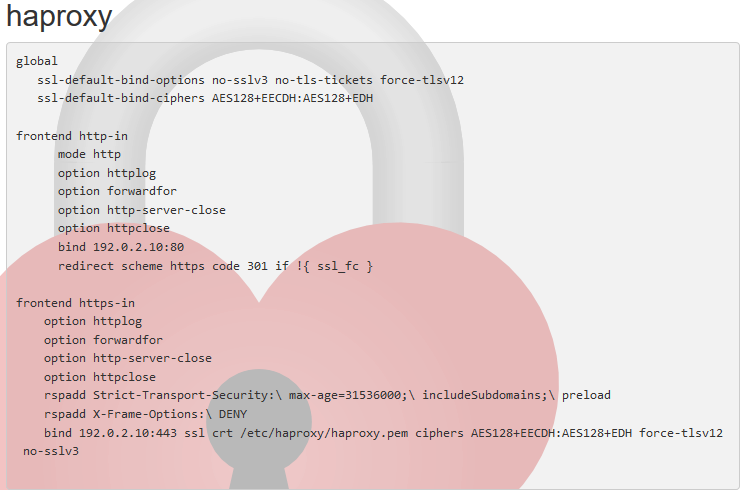
ssl\_prefer\_server\_ciphers 配置为 on，可以确保在 TLSv1 握手时，使用服务端的配置项，以增强安全性。

最终效果是我的博客在 [ssllabs](https://www.ssllabs.com/ssltest/index.html) 的测试中达到了 A+（原文：<https://imququ.com/post/my-nginx-conf-for-security.html>）



16.Haproxy配置SSL

参考：<https://cipherli.st/>



17. reuseport

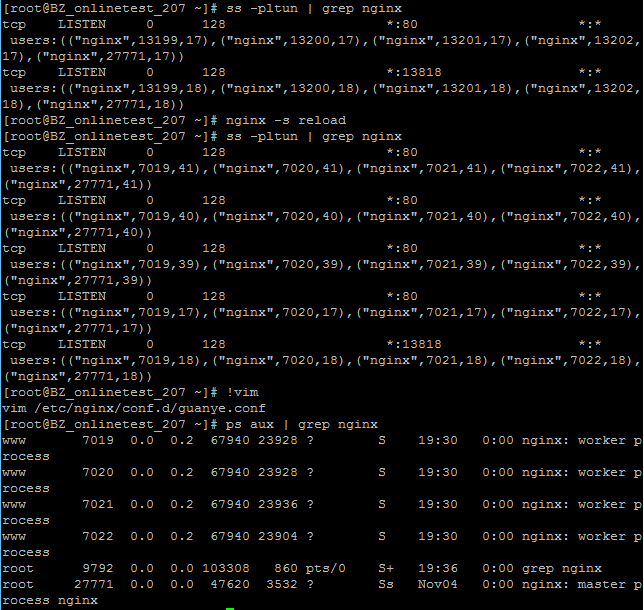
在server模块配置如下：

server {

listen 80 reuseport;

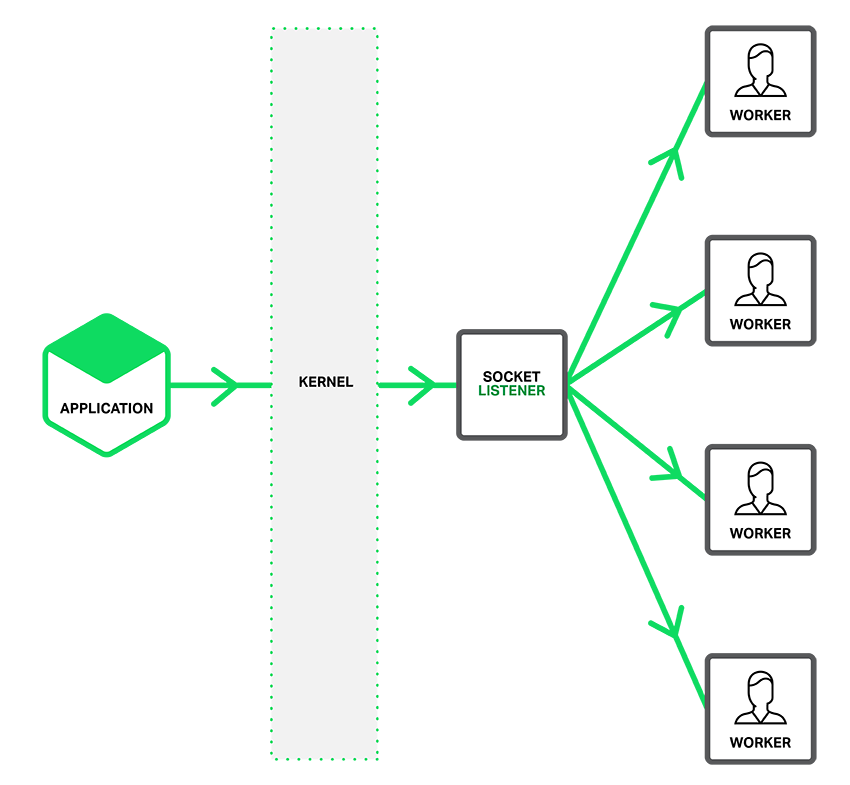


重启后，nginx的80端口有4个，而nginx-master的进程还是只有1个

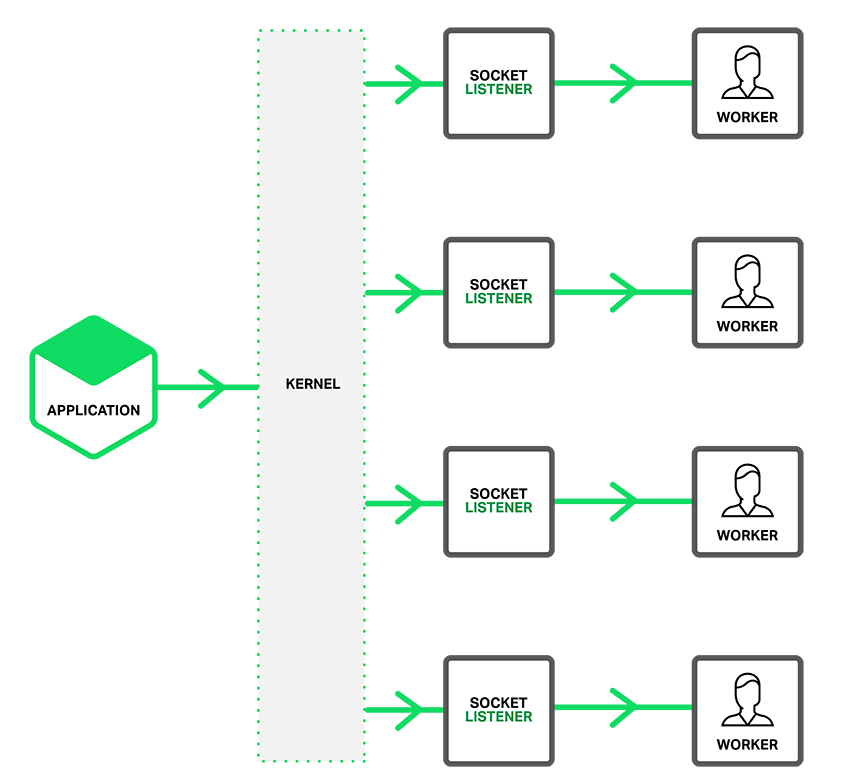


NGINX 1.9版以后就支持socket的SO\_REUSEPORT选项，这个选项允许多个socket监听同个IP地址和端口的组合。内核负载均衡这些进来的sockets连接，将这些socket有效的分片，它通过分布进程上的连接以提升性能。

正如下图所描述的，当SO\_REUSEPORT选项没开启时，连接进来时监听socket默认会通知某个进程。如果您包含了accept\_mutex off这个指令，此时会唤醒所有的工作进程，它们将为了得到它产生竞争。这就是所谓的惊群现象。（解释下：如果使用epoll且不用锁，当监听端口有读操作时，是会产生惊群现象的）



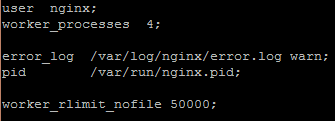
启用SO\_REUSEPORT选项后，每个进程将有个独立的监听socket。内核决定哪个是有效的socket（进程）得到这个连接。这样做降低了延迟并提高了工作进程的性能，它也意味着工作进程在准备处理它们前被赋予了新的连接。



18．高层的配置

worker\_processes auto;

worker\_rlimit\_nofile 100000;



worker\_processes 定义了nginx对外提供web服务时的worder进程数。最优值取决于许多因素，包括（但不限于）CPU核的数量、存储数据的硬盘数量及负载模式。不能确定的时候，将其设置为可用的CPU内核数将是一个好的开始（设置为“auto”将尝试自动检测它）。

worker\_rlimit\_nofile 更改worker进程的最大打开文件数限制。如果没设置的话，这个值为操作系统的限制。设置后你的操作系统和Nginx可以处理比“ulimit -a”更多的文件，所以把这个值设高，这样nginx就不会有“too many open files”问题了。

19. Events模块

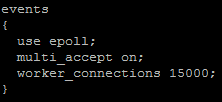
events {

worker\_connections 15000;

multi\_accept on;

use epoll;

}



use 设置用于复用客户端线程的轮询方法。如果你使用Linux 2.6+，你应该使用epoll。如果你使用\*BSD，你应该使用kqueue。想知道更多有关事件轮询？看下维基百科吧（注意，想了解一切的话可能需要neckbeard和操作系统的课程基础）。

worker\_connections设置可由一个worker进程同时打开的最大连接数。如果设置了上面提到的worker\_rlimit\_nofile，我们可以将这个值设得很高。记住，最大客户数也由系统的可用socket连接数限制（~ 64K），所以设置不切实际的高没什么好处。

multi\_accept 告诉nginx收到一个新连接通知后接受尽可能多的连接。

20. HTTP 模块

access\_log off;

error\_log /var/log/nginx/error.log warn;



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(aka,YOLO)

error\_log 告诉nginx只能记录严重的错误

sendfile on;

tcp\_nopush on;

server\_tokens off;

keepalive\_timeout 5;

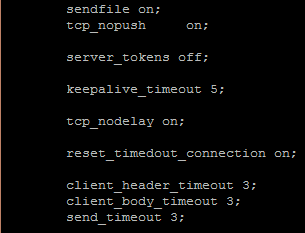
tcp\_nodelay on;

reset\_timedout\_connection on;

client\_header\_timeout 3;

client\_body\_timeout 3;

send\_timeout 3



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。

sendfile可以让sendfile()发挥作用。sendfile()可以在磁盘和TCP socket之间互相拷贝数据(或任意两个文件描述符)。Pre-sendfile是传送数据之前在用户空间申请数据缓冲区。之后用read()将数据从文件拷贝到这个缓冲区，write()将缓冲区数据写入网络。sendfile()是立即将数据从磁盘读到OS缓存。因为这种拷贝是在内核完成的，sendfile()要比组合read()和write()以及打开关闭丢弃缓冲更加有效(更多有关于sendfile)

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

tcp\_nodelay 告诉nginx不要缓存数据，而是一段一段的发送--当需要及时发送数据时，就应该给应用设置这个属性，这样发送一小块数据信息时就不能立即得到返回值。

keepalive\_timeout 给客户端分配keep-alive链接超时时间。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。

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。

gzip on;

gzip\_static on;

gzip\_min\_length 1k;

gzip\_buffers 8 16k;

# gzip\_http\_version 1.1;

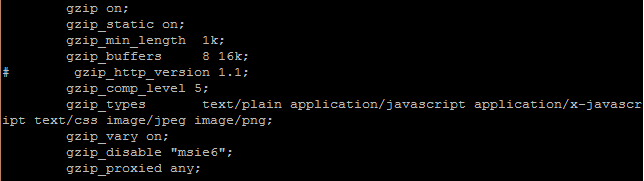
gzip\_comp\_level 5;

gzip\_types text/plain application/javascript application/x-javascript text/css image/jpeg image/png;

gzip\_vary on;

gzip\_disable "msie6";

gzip\_proxied any;



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，你也可以再添加更多的格式。