**原创；微信公众号：千里行走；**

**受限图片大小限制，有些图片不是很清晰，可以到微信公众号查看；**

**目录**

**(1).前述**

**(2).nginx生产级别高性能配置**

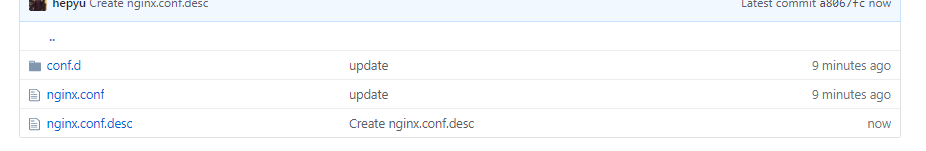
**正文**

**(1).前述**

笔者在github提供了完整的配置文件，同时包含代理基础组件的conf，这些基础组件都是部署在容器里，通过ingress-nginx暴露到外部，然后在通过nginx(openrestry)暴露到公网。

Github地址：

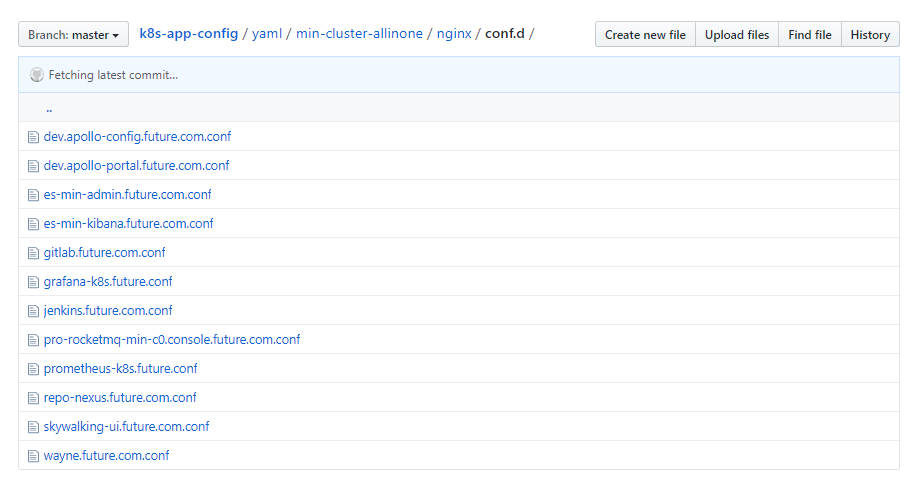
<https://github.com/hepyu/k8s-app-config/tree/master/yaml/min-cluster-allinone/nginx>



如上图：

nginx.conf是主配置文件(生产级别可用的高性能配置)，conf.d下是各个基础组件的nginx反响代理配置文件，nginx.conf.desc中是对nginx.conf每个配置的详细说明和应用场景。

下图是代理的K8s容器中的所有基础组件(conf.d目录)：



**(2).nginx生产级别高性能配置**

位于笔者github：

<https://github.com/hepyu/k8s-app-config/blob/master/yaml/min-cluster-allinone/nginx/nginx.conf>

详细说明位于：

<https://github.com/hepyu/k8s-app-config/blob/master/yaml/min-cluster-allinone/nginx/nginx.conf.desc>

#nginx有很多参数可以配置在http/server/location中，这三者的区别如下，以client\_max\_body\_size举例：

#三者到区别是：

# http{} 中控制着所有nginx收到的请求。

# 而报文大小限制设置在server｛｝中，则控制该server收到的请求报文大小。

# 同理，如果配置在location中，则报文大小限制，只对匹配了location 路由规则的请求生效。

#可以选择在http{ }中设置：client\_max\_body\_size   20m;

#也可以选择在server{ }中设置：client\_max\_body\_size   20m;

#还可以选择在location{ }中设置：client\_max\_body\_size   20m;

#指定启动nginx使用的用户

user hpy hpy;

#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\_processes auto;

#Nginx默认没有开启利用多核CPU,我们可以通过增加worker\_cpu\_affinity配置参数来充分利用多核CPU。

#CPU是任务处理，计算最关键的资源，CPU核越多，性能就越好。

worker\_cpu\_affinity auto;

#worker\_processes auto和worker\_cpu\_affinity auto表示启动的worker process进程数量是cpu个数，且均匀绑定到不同的cpu核上。

#这两个参数配置成功的标志是：每个cpu的使用率基本都是一致的。

#生产情况下，我们一般分配专门机器跑nginx，且只跑nginx，从而最大程度上利用nginx自身高效性和现代cpu的多核高效性。

#指定一个nginx进程可以打开的最多文件描述符数目

#文件描述符是一个简单的整数，用以标明每一个被进程所打开的文件和socket。第一个打开的文件是0，第二个是1，依此类推。Unix 操作系统通常给每个进程能打开的文件数量强加一个限制。更甚的是，unix 通常有一个系统级的限制。所以必须把这个参数调大，否则nginx无法跑满效率。

worker\_rlimit\_nofile 65535;

#nginx的error\_log类型如下（从左到右：debug最详细 crit最少）：

#[ debug | info | notice | warn | error | crit ]

#生产环境我们只打error日志；access日志一般是关闭的，否则中断太多，严重影响nginx性能。

error\_log /data/hpy/logs/nginx/error.log notice;

#pid位置

pid /app/3rd/nginx/openresty/nginx/conf/nginx.pid;

#events模块中包含nginx中所有处理连接的设置

events{

#使用epoll的I/O 模型，必开项，极其有利于性能。

use epoll;

#worker\_connections配置表示每个工作进程的并发连接数，默认设置为1024。

#工作进程的最大连接数量 理论上每台nginx服务器的最大连接数为worker\_processes\*worker\_connections worker\_processes为我们在main中开启的进程数。

worker\_connections 65535;

}

http {

#HTTP request里面有一个头叫 Accept，列出浏览器可以接受的mime type，HTTP response 的Content-Type 的值 在Accept 里面。

#这个mime.types指定了nginx可以接受的Content-Type；mime.types文件默认位于nginx.conf的同级目录。

include mime.types;

#默认文件类型

#意思是如果一个文件的mime类型不存在就会使用默认的类型。 通常是这个导致了文件的下载。

#将default\_type application/octet-stream; 修改为default\_type text/html; 这样就默认表示一个文件是 html文件， 就可以在浏览器中查看。

#上面的方面可以解决文件不在浏览器中预览的情况，但是如果有文件的扩展名对应mime信息找不到，也会进行预览， 不管文件是不是文本文件，都是当成是文本文件。

default\_type application/octet-stream;

#log\_format指令用来设置日志的记录格式，它的语法如下：log\_format name format {format ...},其中name表示定义的格式名称，format表示定义的格式样式。

#"version": "2":

# 自定义，表示当前nginx.conf是第几个版本。方便CMDB管理。

#"time": "$time\_iso8601":

# 使用到$time\_iso8601 内嵌变量来获取时间。$time\_iso8601格式如下：2015-08-07T18:12:02+02:00。

#"remote\_addr": "$remote\_addr":

# 1.remote\_addr代表客户端的IP，但它的值不是由客户端提供的，而是服务端根据客户端的ip指定的;

# 当你的浏览器访问某个网站时，假设中间没有任何代理，那么网站的web服务器（Nginx，Apache等）就会把remote\_addr设为你的机器IP;

# 如果你用了某个代理，那么你的浏览器会先访问这个代理，然后再由这个代理转发到网站，这样web服务器就会把remote\_addr设为这台代理机器的IP;

# 2.但是实际场景中，我们即使有代理，也需要将$remote\_addr设置为真实的用户IP，以便记录在日志当中，当然nginx是有这个功能，但是需要编译的时候添加--with-http\_realip\_module 这个模块，默认是没有安装的。

#"status": "$status":

# 记录请求状态。

#"bytes\_sent": "$bytes\_sent":

# 发送给客户端的总字节数。

#"host": "$host":

# $host 是nginx的官方变量，可以从官方查询。

# 举例：

# curl --silent -H "Host: www.xxxx.com" "hostname/xxxx/xxxx.htm"

# 这种方式与windows中更改hosts文件，指定域名对应用的具体host的ip是一样的，

# curl http://ww.xxxx.com/xxxx/xxxx.htm -xhostname:80

#"request\_method": "$request\_method"：

# 请求方式：POST, GET, PUT等。

#"request\_uri": "$request\_uri"：

# 这个变量等于从客户端发送来的原生请求URI，包括参数。它不可以进行修改。

# Example: "/foo/bar.php?arg=baz"

#"uri": "$uri"：

# 不带请求参数的当前URI，$uri不包含主机名，如”/foo/bar.html”。

#"request\_time": "$request\_time":

# 指的就是从接受用户请求的第一个字节到发送完响应数据的时间，即$request\_time包括接收客户端请求数据的时间、后端程序响应的时间、发送响应数据给客户端的时间。

#"response\_time": "$upstream\_response\_time":

# 是指从Nginx向后端建立连接开始到接受完数据然后关闭连接为止的时间。

# 从上面的描述可以看出，$request\_time肯定比$upstream\_response\_time值大；尤其是在客户端采用POST方式提交较大的数据，响应体比较大的时候。在客户端网络条件差的时候，$request\_time还会被放大。

#"http\_referer": "$http\_referer":

# 用来记录从那个页面链接访问过来的；可以使用这个参数做防盗链。

#"body\_bytes\_sent": "$body\_bytes\_sent":

# 发送给客户端的字节数，不包括响应头的大小。

#"bytes\_sent": "$bytes\_sent"：

# 发送给客户端的总字节数。

#"http\_user\_agent": "$http\_user\_agent":

# 记录客户端浏览器相关信息；可以使用这个参数做防盗链。

#"http\_x\_forwarded\_for": "$http\_x\_forwarded\_for":

# REMOTE\_ADDR代表着客户端的IP，但是这个客户端是相对服务器而言的，也就是实际上与服务器相连的机器的IP（建立tcp连接的那个），这个值是不可伪造的，如果没有代理的话，这个值就是用户实际的IP值，有代理的话，用户的请求会经过代理再到服务器，这个时候REMOTE\_ADDR会被设置为代理机器的IP值。

# 正如前面所说，有了代理就获取不了用户的真实IP，由此X-Forwarded-For应运而生，它是一个非正式协议，在请求转发到代理的时候代理会添加一个X-Forwarded-For头，将连接它的客户端IP（也就是你的上网机器IP）加到这个头信息里，这样末端的服务器就能获取真正上网的人的IP了。

# 假设用户的请求顺序如下：

# 网民电脑ip->代理服务器1–>代理服务器2–>目标服务器

# REMOTE\_ADDR:代理服务器2的IP值

# X-Forwarded-For就是：网民电脑IP，代理1的IP，代理2的IP

# 在这里只有REMOTE\_ADDR是可信的，其他从客户端获取的数据都是不可信的，都是可伪造的。下面简单示例下一个篡改X-Forwarded-For的情况：

#"coohua\_id": "$http\_coohua\_id":

#"ukey": "$http\_u\_key":

#"vkey": "$http\_v\_key":

#"cookie": "$http\_cookie":

# 获取全部cookie信息。

log\_format access

'{"version": "2", '

'"time": "$time\_iso8601", '

'"remote\_addr": "$remote\_addr", '

'"status": "$status", '

'"bytes\_sent": "$bytes\_sent", '

'"host": "$host", '

'"request\_method": "$request\_method", '

'"request\_uri": "$request\_uri", '

'"request\_time": "$request\_time", '

'"response\_time": "$upstream\_response\_time",'

'"http\_referer": "$http\_referer", '

'"body\_bytes\_sent": "$body\_bytes\_sent", '

'"http\_user\_agent": "$http\_user\_agent", '

'"http\_x\_forwarded\_for": "$http\_x\_forwarded\_for", '

'"coohua\_id": "$http\_coohua\_id", '

'"ukey": "$http\_u\_key", '

'"vkey": "$http\_v\_key", '

'"cookie": "$http\_cookie"}';

#用来指定日志文件的存放路径。

access\_log /data/coohua/logs/nginx/access.log access;

#这个参数指定了是否记录客户端的请求出现404错误的日志。

log\_not\_found off;

#隐藏版本号

#off：表示赢藏nginx版本号。

server\_tokens off;

#开启0拷贝

#零拷贝主要的任务就是避免CPU将数据从一块存储拷贝到另外一块存储，主要就是利用各种零拷贝技术，避免让CPU做大量的数据拷贝任务，减少不必要的拷贝；

#或者让别的组件来做这一类简单的数据传输任务，让CPU解脱出来专注于别的任务。这样就可以让系统资源的利用更加有效。

#当需要对一个文件进行传输的时候，其具体流程细节如下：

#(1).sendfile off(关闭0拷贝)下的流程：

# 1、调用read函数，文件数据被copy到内核缓冲区

# 2、read函数返回，文件数据从内核缓冲区copy到用户缓冲区

# 3、write函数调用，将文件数据从用户缓冲区copy到内核与socket相关的缓冲区。

# 4、数据从socket缓冲区copy到相关协议引擎。

# 以上细节是传统read/write方式进行网络文件传输的方式，我们可以看到，在这个过程当中，文件数据实际上是经过了四次copy操作：

# 硬盘—>内核buf—>用户buf—>socket相关缓冲区—>协议引擎

#(2).sendfile on(开启0拷贝)下的流程：

# 1、sendfile系统调用，文件数据被copy至内核缓冲区

# 2、再从内核缓冲区copy至内核中socket相关的缓冲区

# 3、最后再socket相关的缓冲区copy到协议引擎

# 这个过程数据经历的拷贝操作如下：

# 硬盘—>内核缓冲区—>协议引擎

sendfile on;

#tcp\_nopush是一个 socket 选项，并且只有在启用了 sendfile 之后才生效(可以理解，只有sendfile开启才能确保tcp\_nopush的效率)；

#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\_nopush on;

#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。

tcp\_nodelay on;

#关于tcp\_nopush与tcp\_nodelay的混合使用：

#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。

#关于这部分内容的更多介绍可以看这篇文章：

# NGINX OPTIMIZATION: UNDERSTANDING SENDFILE, TCP\_NODELAY AND TCP\_NOPUSH。

# url link：https://thoughts.t37.net/nginx-optimization-understanding-sendfile-tcp-nodelay-and-tcp-nopush-c55cdd276765

#指定客户端与服务端建立连接后发送 request body 的超时时间；

#如果客户端在指定时间内没有发送任何内容，Nginx 返回 HTTP 408（Request Timed Out）；

#对于弱网用户，尤其是移动用户来说，这个配置是能大幅提供nginx的性能的，防止nginx资源浪费。

# 配置段: http, server, location

client\_body\_timeout 10;

#HTTP 是一种无状态协议，客户端向服务器发送一个 TCP 请求，服务端响应完毕后断开连接。

#如果客户端向服务器发送多个请求，每个请求都要建立各自独立的连接以传输数据。

#HTTP 有一个 KeepAlive 模式，它告诉 webserver 在处理完一个请求后保持这个 TCP 连接的打开状态。

#若接收到来自客户端的其它请求，服务端会利用这个未被关闭的连接，而不需要再建立一个连接。

#KeepAlive 在一段时间内保持打开状态，它们会在这段时间内占用资源。占用过多就会影响性能。

#Nginx 使用 keepalive\_timeout 来指定 KeepAlive 的超时时间（timeout）。指定每个 TCP 连接最多可以保持多长时间。

#Nginx 的默认值是 75 秒，有些浏览器最多只保持 60 秒，所以可以设定为 60 秒。若将它设置为 0，就禁止了 keepalive 连接。

#通常keepalive\_timeout应该比client\_body\_timeout大。

# 配置段: http, server, location

keepalive\_timeout 60;

#keepalive\_requests指令用于设置一个keep-alive连接上可以服务的请求的最大数量，当最大请求数量达到时，连接被关闭。

#默认是100。

#这个参数的真实含义：

# 是指一个keep alive建立之后，nginx就会为这个连接设置一个计数器，记录这个keep alive的长连接上已经接收并处理的客户端请求的数量。

# 如果达到这个参数设置的最大值时，则nginx会强行关闭这个长连接，逼迫客户端不得不重新建立新的长连接。

#大多数情况下当QPS(每秒请求数)不是很高时，默认值100凑合够用。但是，对于一些QPS比较高（比如超过10000QPS，甚至达到30000,50000甚至更高) 的场景，默认的100就显得太低。

#简单计算一下，QPS=10000时，客户端每秒发送10000个请求(通常建立有多个长连接)，每个连接只能最多跑100次请求。

# 意味着平均每秒钟就会有100个长连接因此被nginx关闭。同样意味着为了保持QPS，客户端不得不每秒中重新新建100个连接。

# 因此，就会发现有大量的TIME\_WAIT的socket连接(即使此时keep alive已经在client和nginx之间生效)。

# 因此对于QPS较高的场景，非常有必要加大这个参数，以避免出现大量连接被生成再抛弃的情况，减少TIME\_WAIT。

#对于黏度/交互度很高的用户场景，这个值是能够大幅提供nginx的服务能力的。

keepalive\_requests 1000;

#默认值是4K，同pagesize大小，request请求头超过大小nginx会返回400。

#客户端请求头部的缓冲区大小，这个可以根据你的系统分页大小来设置，一般一个请求头的大小不会超过1k，不过由于一般系统分页都要大于1k，所以这里设置为系统分页大小。

#查看系统分页可以使用 getconf PAGESIZE命令

#现在我们很多时候都会往Header里放一堆东西，所以这个值需要根据实际情况调整。

client\_header\_buffer\_size 32k;

#关于client\_header\_buffer\_size和large\_client\_header\_buffers

# 先根据client\_header\_buffer\_size配置的值分配一个buffer，

# 如果分配的buffer无法容纳 request\_line/request\_header，那么就会再次根据large\_client\_header\_buffers配置的参数分配large\_buffer，

# 如果large\_buffer还是无法容纳，那么就会返回414（处理request\_line）/400（处理request\_header）错误。

#large\_client\_header\_buffers设置客户端请求的Header头缓冲区大小，默认为4K。

# 客户端请求行不能超过设置的第一个数，请求的Header头信息不能大于设置的第二个数，否则会报"Request URI too large"(414)或“Bad request”(400)错误。

# 如果客户端的Cookie信息较大，则需增加缓冲区大小

large\_client\_header\_buffers 4 64k;

#这个directive让Nginx将所有的request body存储在一个缓冲当中，它的默认值是off。

#启用它可以优化读取$request\_body变量时的I/O性能。可以在http, server 和 location模块中定义。

client\_body\_in\_single\_buffer on;

#这个directive设定了request body的缓冲大小。

# 如果body超过了缓冲的大小，那么整个body或者部分body将被写入一个临时文件。

# 如果Nginx被设置成使用文件缓冲而不使用内存缓冲，那么这个dirctive就无效。

# client\_body\_buffer\_size在32位系统上默认是8k，在64位系统上默认是16k。

#可以在http, server 和 location模块中指定。

client\_body\_buffer\_size 128k;

#相关场景&问题：

# 服务器有时上出现很多499的错误，出现499错误的原因是客户端关闭了连接；

# 服务端在执行时中途关闭浏览器退出之后php/java等后端是否还会继续执行；

#proxy\_ignore\_client\_abort是否开启proxy忽略客户端中断。

# 即如果此项设置为on开启，则服务器会忽略客户端中断，一直等着代理服务执行返回。并且如果执行没有发生错误，记录的日志是200日志。如果超时则会记录504。

# 如果设置为off，则客户端中断后服务器端nginx立即记录499日志，但要注意，此时代理端的PHP/Java程序会依然继续执行。

#nginx的proxy\_ignore\_client\_abort默认是关闭的，即请求过程中如果客户端端主动关闭请求或者客户端网络断掉，那么Nginx会记录499。所以如果不想看到499报错，可以修改配置：proxy\_ignore\_client\_abort on ;

#也就是说，499错误并不是一个问题，如果出现了大量的499的话，需要考虑为什么发生了这么多的客户端中断的问题。

proxy\_ignore\_client\_abort on;

#HTTP头是可以包含英文字母([A-Za-z])、数字([0-9])、连接号(-)hyphens, 也可义是下划线(\_)。在使用nginx的时候应该避免使用包含下划线的HTTP头。主要的原因有以下2点。

# 1.默认的情况下nginx引用header变量时不能使用带下划线的变量。要解决这样的问题只能单独配置underscores\_in\_headers on。

# 2.默认的情况下会忽略掉带下划线的变量。要解决这个需要配置ignore\_invalid\_headers off。

underscores\_in\_headers on;

ignore\_invalid\_headers off;

#客户端向服务端发送一个完整的 request header 的超时时间；

#如果客户端在指定时间内没有发送一个完整的 request header，Nginx 返回 HTTP 408（Request Timed Out）；

#对于弱网用户，尤其是移动用户来说，这个配置是能大幅提供nginx的性能的，防止nginx资源浪费。

# 配置段: http, server, location

client\_header\_timeout 9;

#这条directive指定了向客户端传输数据的超时时间。默认值为60秒，可以在http, server 和 location模块中定义。

send\_timeout 60;

#三者到区别是：http{} 中控制着所有nginx收到的请求。

# 而报文大小限制设置在server｛｝中，则控制该server收到的请求报文大小，同理，如果配置在location中，则报文大小限制，只对匹配了location 路由规则的请求生效。

#可以选择在http{ }中设置：client\_max\_body\_size   20m;

#也可以选择在server{ }中设置：client\_max\_body\_size   20m;

#还可以选择在location{ }中设置：client\_max\_body\_size   20m;

#我们使用web上传文件时用nginx做代理必须配置此项，否则文件太大会发生中断。

client\_max\_body\_size 50m;

#网站加载的速度取决于浏览器必须下载的所有文件的大小。减少要传输的文件的大小可以使网站不仅加载更快，而且对于那些宽带是按量计费的人来说也更友好。

#gzip是一种流行的数据压缩程序。您可以使用gzip压缩Nginx实时文件。这些文件在检索时由支持它的浏览器解压缩，好处是web服务器和浏览器之间传输的数据量更小，速度更快。

#gzip不一定适用于所有文件的压缩。

# 例如，文本文件压缩得非常好，通常会缩小两倍以上。

# 另一方面，诸如JPEG或PNG文件之类的图像已经按其性质进行压缩，使用gzip压缩很难有好的压缩效果或者甚至没有效果。

# 压缩文件会占用服务器资源，因此最好只压缩那些压缩效果好的文件。

# mod\_gzip configurations

#开启gzip，默认值gzip off(关闭)

gzip on;

#它的默认值是1.1，就是说对HTTP/1.1协议的请求才会进行gzip压缩。

gzip\_http\_version 1.0;

#gzip的压缩比，1-9个数量级，数据越大，压缩的程度越高，压缩后占用空间越小，但是效率最低，更加消耗CPU，一般为6即可。

gzip\_comp\_level 6;

#设置一个将要被压缩的响应的最小长度值(即长度小于这个值的响应将不会被压缩)。

#设置这项的主要原因是在数据过小的情况下，压缩效果不明显，不如不进行压缩，建议设置成512，只对大于512的响应数据进行压缩（页面字节数从header头中的Content-Length中进行获取）

gzip\_min\_length 1k;

#该指令在nginx使用反向代理的时候起作用，是否压缩取决于请求头中的“Via”字段，指令中可以同时指定多个不同的参数。根据请求和响应来决定启用或禁用对代理请求的响应的压缩

# （1）off：禁用对所有代理请求的压缩

# （2） expired：当响应头中包含过期时间时，启用压缩

# （3） no-cache：当响应头的Cache-Control字段为no-cache时，启用压缩

# （4） no-store：当响应头的Cache-Control字段为no-store时，启用压缩

# （5） private：当响应头的Cache-Control字段为no-store时，启用压缩

# --cache\_control用于设置缓存机制

# （6）no\_last\_modified:当响应头不包含响应最后修改时间字段时，启用压缩。

# （7）no\_etag:当想用头中不包含被请求变量的实体值时，启用压缩

# （8）auth：当响应头包含用于授权http证书的Authorization字段时，启用压缩

# （9）any：岁所有代理请求开启压缩。

#默认值 gzip\_proxied off;

gzip\_proxied any;

#该指令用于只用gzip功能时，是否发送Vary: Accept-Encoding响应头字段，通知接收方响应使用了gzip压缩

gzip\_vary on;

#不启用压缩的条件，IE6对Gzip不友好，所以不压缩

gzip\_disable msie6;

#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\_buffers 4 64k;

#需要压缩的文件mime类型

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

#即允许重新定义或添加字段传递给代理服务器的请求头。该值可以包含文本、变量和它们的组合。在没有定义proxy\_set\_header时会继承之前定义的值。默认情况下，只有两个字段被重定义：

# proxy\_set\_header Host $proxy\_host;

# proxy\_set\_header Connection close;

# mod\_http\_proxy

#举例说明$host, $proxy\_host的使用和区别：

#location /front {

# proxy\_pass http://web;

# proxy\_set\_header Host $proxy\_host;

#}

#当匹配到/front时，使用web处理，到upstream就匹配到abc.com，这里直接转换成IP进行转发。

#1.假如abc.com是在另一台nginx下配置的，ip为10.10.10.10，则$proxy\_host则对应为10.10.10.10。

# 此时相当于设置了Host为10.10.10.10。如果想让Host是abc.com，则进行如下设置：

# proxy\_set\_header Host abc.com;

#2.如果不想改变请求头“Host”的值，可以这样来设置：

# proxy\_set\_header Host $http\_host;

#3.但是，如果客户端请求头中没有携带这个头部，那么传递到后端服务器的请求也不含这个头部。

# 这种情况下，更好的方式是使用$host变量——它的值在请求包含“Host”请求头时为“Host”字段的值，在请求未携带“Host”请求头时为虚拟主机的主域名：

# proxy\_set\_header Host $host;

#4.此外，服务器名可以和后端服务器的端口一起传送：

# proxy\_set\_header Host $host:$proxy\_port;

proxy\_set\_header Host $host;

#设置真实客户端IP

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

#1、X-Forwarded-For的定义：

# X-Forwarded-For:简称XFF头，它代表客户端，也就是HTTP的请求端真实的IP，只有在通过了HTTP 代理或者负载均衡服务器时才会添加该项。它不是RFC中定义的标准请求头信息，在squid缓存代理服务器开发文档中可以找到该项的详细介绍。

# 标准格式如下：

# X-Forwarded-For: client1, proxy1, proxy2

# 从标准格式可以看出，X-Forwarded-For头信息可以有多个，中间用逗号分隔，第一项为真实的客户端ip，剩下的就是曾经经过的代理或负载均衡的ip地址，经过几个就会出现几个。

#2、依照WEB架构图进行分析

# a.在有CDN的情况下，当用户请求经过CDN后到达Nginx负载均衡服务器时，其X-Forwarded-For头信息应该为 客户端IP,CDN的IP 但实际情况并非如此，一般情况下CDN服务商为了自身安全考虑会将这个信息做些改动，只保留客户端IP。

# 我们可以通过php程序获得X-Forwarded-For信息或者通过Nginx的add header方法来设置返回头来查看。

# b.下面来分析请求头到达Nginx负载均衡服务器的情况；在默认情况下，Nginx并不会对X-Forwarded-For头做任何的处理，除非用户使用proxy\_set\_header 参数设置：

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

# $proxy\_add\_x\_forwarded\_for变量包含客户端请求头中的"X-Forwarded-For"，与$remote\_addr用逗号分开，

# 如果没有"X-Forwarded-For" 请求头，则$proxy\_add\_x\_forwarded\_for等于$remote\_addr。$remote\_addr变量的值是客户端的IP。

# c.当Nginx设置X-Forwarded-For于$proxy\_add\_x\_forwarded\_for后会有两种情况发生：

# c1.如果从CDN过来的请求没有设置X-Forwarded-For头（通常这种事情不会发生），而到了我们这里Nginx设置将其设置为$proxy\_add\_x\_forwarded\_for的话，X-Forwarded-For的信息应该为CDN的IP，因为相对于Nginx负载均衡来说客户端即为CDN，这样的话，后端的web程序时死活也获得不了真实用户的IP的。

# c2.CDN设置了X-Forwarded-For，我们这里又设置了一次，且值为$proxy\_add\_x\_forwarded\_for的话，那么X-Forwarded-For的内容变成 ”客户端IP,Nginx负载均衡服务器IP“如果是这种情况的话，那后端的程序通过X-Forwarded-For获得客户端IP，则取逗号分隔的第一项即可。

# 如上两点所说，如果我们知道了CDN设置了X-Forwarded-For信息，且只有客户端真实的IP的话，那么我们的Nginx负载均衡服务器可以不必理会该头，让它默认即可。

#其实Nginx中还有一个$http\_x\_forwarded\_for变量，这个变量中保存的内容就是请求中的X-Forwarded-For信息。

#如果后端获得X-Forwarded-For信息的程序兼容性不好的话（没有考虑到X-Forwarded-For含有多个IP的情况），最好就不要将X-Forwarded-For设置为 $proxy\_add\_x\_forwarded\_for。

# 应该设置为$http\_x\_forwarded\_for 或者干脆不设置！

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

#http协议，HTTP协议中对长连接的支持是从1.1版本之后才有的，因此最好通过proxy\_http\_version指令设置为”1.1”；

#必设项，否则很影响nginx性能。

proxy\_http\_version 1.1;

#表示nginx每次访问完后端server后，如何处理与后端server的这次请求连接，默认配置是close，即每次访问完后端服务后都关闭此次连接，显然这回给后端server带来大量的TIME\_WAIT连接，降低后端server性能。

proxy\_set\_header Connection "";

#综上，我们总结一下出现TIME\_WAIT的情况：

#出现大量TIME\_WAIT的情况主要有两种：

# 1）导致 nginx端出现大量TIME\_WAIT的情况有两种：

# keepalive\_requests设置比较小，高并发下超过此值后nginx会强制关闭和客户端保持的keepalive长连接；（主动关闭连接后导致nginx出现TIME\_WAIT）

# keepalive设置的比较小（空闲数太小），导致高并发下nginx会频繁出现连接数震荡（超过该值会关闭连接），不停的关闭、开启和后端server保持的keepalive长连接；

# 2）导致后端server端出现大量TIME\_WAIT的情况：

# nginx没有打开和后端的长连接，即：没有设置proxy\_http\_version 1.1;和proxy\_set\_header Connection “”;从而导致后端server每次关闭连接，高并发下就会出现server端出现大量TIME\_WAIT

#在使用Nginx做反向代理功能时，有时会出现重定向的url不是我们想要的url，这时候就可以使用proxy\_redirect进行url重定向设置了。

#proxy\_redirect功能比较强大,其作用是对发送给客户端的URL进行修改。

#默认：proxy\_redirect default;

#配置块（使用的字段）：http、server、location

#我们是关闭此项的。

proxy\_redirect off;

#语法：proxy\_buffers 数量 大小

#默认值：proxy\_buffers 8 4k/8k

#上下文：http,server,location

#该指令设置缓冲区的大小和数量,从被代理的后端服务器取得的响应内容,会放置到这里. 默认情况下,一个缓冲区的大小等于内存页面大小,可能是4K也可能是8K,这取决于平台。

proxy\_buffers 64 8k;

#proxy\_connect\_timeout 是和后端建立连接的超时时间,单位秒，默认值60秒。

proxy\_connect\_timeout 60;

#缓存临时目录。

#后端的响应并不直接返回客户端，而是先写到一个临时文件中，然后被rename一下当做缓存放在 proxy\_cache\_path 。

#0.8.9版本以后允许temp和cache两个目录在不同文件系统上（分区），然而为了减少性能损失还是建议把它们设成一个文件系统上

proxy\_temp\_path /app/3rd/nginx/default/proxy\_temp;

#设置缓存目录，目录里的文件名是 cache\_key 的MD5值。

# levels=1:2 keys\_zone=cache\_one:512m：

# 表示采用2级目录结构，Web缓存区名称为cache\_one，内存缓存空间大小为512MB，这个缓冲zone可以被多次使用。

# inactive=1d max\_size=2g：

# 表示1天没有被访问的内容自动清除，硬盘最大缓存空间为2GB，超过这个大学将清除最近最少使用的数据。

proxy\_cache\_path /app/3rd/nginx/proxy\_cache levels=1:2 keys\_zone=cache\_one:512m inactive=1d max\_size=2g;

# fight DDoS attack, tune the numbers below according your application!!!

#默认值: none，不开启

#配置段: http

#设置一块共享内存限制域用来保存键值的状态参数。 特别是保存了当前超出请求的数量。 键的值就是指定的变量（空值不会被计算）。如

# limit\_req\_zone $binary\_remote\_addr zone=req:20m rate=1r/s;

# 区域名称为one，大小为20m，平均处理的请求频率不能超过每秒一次。

# 说明：

# 键值是客户端IP。

# 使用$binary\_remote\_addr变量， 可以将每条状态记录的大小减少到64个字节，这样1M的内存可以保存大约1万6千个64字节的记录。

# 如果限制域的存储空间耗尽了，对于后续所有请求，服务器都会返回 503 (Service Temporarily Unavailable)错误。

# 速度可以设置为每秒处理请求数和每分钟处理请求数，其值必须是整数，所以如果你需要指定每秒处理少于1个的请求，2秒处理一个请求，可以使用 “30r/m”。

#limit\_req\_zone $binary\_remote\_addr zone=req:20m rate=3r/s;

#语法: limit\_req zone=name [burst=number] [nodelay];

#默认值: —

#配置段: http, server, location

# 设置对应的共享内存限制域和允许被处理的最大请求数阈值。

# 如果请求的频率超过了限制域配置的值，请求处理会被延迟，所以所有的请求都是以定义的频率被处理的。

# 超过频率限制的请求会被延迟，直到被延迟的请求数超过了定义的阈值，这时，这个请求会被终止，并返回503 (Service Temporarily Unavailable) 错误。这个阈值的默认值为0。如：

# limit\_req\_zone $binary\_remote\_addr zone=ttlsa\_com:10m rate=1r/s;

# server {

# location /www.ttlsa.com/ {

# limit\_req zone=ttlsa\_com burst=5;

# }

# 限制平均每秒不超过一个请求，同时允许超过频率限制的请求数不多于5个。

# 如果不希望超过的请求被延迟，可以用nodelay参数,如：

# limit\_req zone=ttlsa\_com burst=5 nodelay;

#注意：

# 这些都是临时处理手段，不能作为常态，因为一旦超出说明现在负载均衡不够用，应该想办法解决，比如查看是不是有人刷/增加机器/负载均衡调优等。

# 只要在下载场景例外。

#limit\_req zone=req burst=60;

#要限制用户的连接数可以通过Limit zone模块来达到目的，即限制同一用户IP地址的并发连接数。

# 该模块提供了两个命令limit\_zone和limit\_conn，其中limit\_zone只能用在http区段，而limit\_conn可以用在http, server, location区段。

#该指令用于定义一个zone，该zome将会被用于存储会话状态。

#能够存储的会话数量是由分被交付的变量和memory\_max\_size的大小决定的。

#例如：

# limit\_zone one $binary\_remote\_addr 10m;

# 客户端的IP地址被用作会话，注意，这里使用的是$binary\_remote\_addr而不是$remote\_addr；

# 这是因为，$remote\_addr的长度为7到15个字节，它的会话信息的长度为 32 或 64 bytes；$binary\_remote\_addr 的长度为 4 字节，会话信息的长度为 32 字节。

# 当设置1M的一个zone时，那么如果是用$binary\_remote\_addr方式，该zone将会存放32000个会话。

#limit\_zone conn $binary\_remote\_addr 20m;

#该指令用于为一个会话设定最大的并发连接数。如果并发请求数超过这个限制，那么将会出现"Service unavailable" (503)。

#例如：

limit\_zone one $binary\_remote\_addr 10m;

server {

location /download/ {

limit\_conn one 1;

}

#这个设置将会使得来自用同一个IP的并发连接不能超过1个连接。

#limit\_conn conn 5;

#设置每一个连接的下载速度。

#limit\_rate 50k;

#从下载到你指定的文件大小之后开始限速。

#limit\_rate\_after 1m;

#如果你定义了大量的服务器名字，或者是定义了少见的长的服务器名字，那么你需要在http级别（或者叫区段）调整指令"server\_names\_hash\_max\_size"和"server\_names\_hash\_bucket\_size"的值。

#指令"server\_names\_hash\_bucket\_size"默认值可以是32、64或者是其他值，这依赖于CPU缓存行（ache line）的大小。

# 如果默认值为32，当你定义"too.long.server.name.nginx.org"作为服务器的名字，那么Nginx将会在启动时失败，并且显示如下错误信息：

# 1. could not build the server\_names\_hash,

# 2. you should increase server\_names\_hash\_bucket\_size: 32

# 在以上情况时，应该设置该指令的值为以前值的两倍：

# 1. http {

# 2. server\_names\_hash\_bucket\_size 64;

# 3. …

#如果你指定了大量的服务器名字，那么将会遇到另一个错误信息：

# 1. could not build the server\_names\_hash,

# 2. you should increase either server\_names\_hash\_max\_size: 512

# 3. or server\_names\_hash\_bucket\_size: 32

#

#应该首先尝试设置"server\_names\_hash\_max\_size"的值接近于服务器名字的数量。

# 如果修改这个指令的值仍然没有起到作用，或者是当Nginx的启动太慢时，才去尝试增加指令"server\_names\_hash\_bucket\_size"的值。

# 如果服务器仅监听在一个端口，那么Nginx根本就不会检测服务器的名字（也不会为该端口建立哈希表）。

# 然而，却有一个例外，如果指令"server\_name"的值是一个捕获的正则表达式，那么Nginx不得不执行一个表达式来获取捕获。

#简单理解：

# server\_names\_hash\_max\_size表示nginx可以捕获的服务器名字的最大数量。

# server\_names\_hash\_bucket\_size表示nginx中每个服务器名字的最大长度。

server\_names\_hash\_max\_size 1024;

server\_names\_hash\_bucket\_size 128;

include conf.d/\*.conf;

}

