目录

ngx_http_limit_req_module	1.7.1.28
ngx_http_log_module	1.7.1.29
ngx_http_map_module	1.7.1.30
ngx_http_memcached_module	1.7.1.31
ngx_http_mirror_module	1.7.1.32
ngx_http_mp4_module	1.7.1.33
ngx_http_perl_module	1.7.1.34
ngx_http_proxy_module	1.7.1.35
ngx_http_random_index_module	1.7.1.36
ngx_http_realip_module	1.7.1.37
ngx_http_referer_module	1.7.1.38
ngx_http_rewrite_module	1.7.1.39
ngx_http_scgi_module	1.7.1.40
ngx_http_secure_link_module	1.7.1.41
ngx_http_session_log_module	1.7.1.42
ngx_http_slice_module	1.7.1.43
ngx_http_spdy_module	1.7.1.44
ngx_http_split_clients_module	1.7.1.45
ngx_http_ssi_module	1.7.1.46
ngx_http_ssl_module	1.7.1.47
ngx_http_status_module	1.7.1.48
ngx_http_stub_status_module	1.7.1.49
ngx_http_sub_module	1.7.1.50
ngx_http_upstream_module	1.7.1.51
ngx_http_upstream_conf_module	1.7.1.52
ngx_http_upstream_hc_module	1.7.1.53
ngx_http_userid_module	1.7.1.54
ngx_http_uwsgi_module	1.7.1.55
ngx_http_v2_module	1.7.1.56
ngx_http_xslt_module	1.7.1.57
Mail	1.7.2
ngx_mail_core_module	1.7.2.1
ngx_mail_auth_http_module	1.7.2.2

贡献导引

请严格按照以下步骤操作，如有任何问题，请提出 issue

- 在 GitHub 上点击 `fork` 将本仓库 fork 到自己的仓库，如 `yourname/nginx-docs`，然后 `clone` 到本地。

```
$ git clone git@github.com:yourname/nginx-docs.git
$ cd nginx-docs
# 将项目与上游关联
$ git remote add source git@github.com:DocsHome/nginx-docs.git
```

- 增加内容或者修复错误后提交，并推送到自己的仓库。

```
$ git add .
$ git commit -a "Fix issue #1: change helo to hello"
$ git push origin/master
```

- 在 GitHub 上提交 `pull request`。
- 请定期更新自己仓库内容。

```
$ git fetch source
$ git rebase source/master
$ git push -f origin master
```

排版规范

本项目排版遵循 中文排版指南 规范。

Nginx 中文文档

Nginx 官方文档中文翻译版，由本人在学习 nginx 时顺带翻译。因部分章节涉及到 Nginx Plus 或者其他内容，我将忽略该部分章节的内容。

如果您发现内容存在错误或者不当之处，欢迎提出 Issue 或 PR，期待您的加入：如何贡献。

在线阅读：Github | GitBook

项目状态

翻译中……

排版规范

本项目排版遵循 中文排版指南 规范。

LICENSE

安装 nginx

nginx 可以以不同的方式安装，安装方式取决于当前的操作系统。

在 Linux 上安装

对于 Linux，可以使用 nginx.org 的 nginx 软件包。

在 FreeBSD 上安装

在 FreeBSD 上，可以从 软件包 或通过 ports 系统安装 nginx。ports 系统提供更大的灵活性，提供了各种选项可供选择。port 将使用指定的选项编译 nginx 并进行安装。

从源码安装

如果需要某些特殊功能软件包和 ports 无法提供，那么可以从源码中编译 nginx。虽然此方式更加灵活，但对于初学者来说可能会很复杂。更多信息请参阅 从源码构建 nginx。

原文档

http://nginx.org/en/docs/install.html

初学者指南

本指南旨在介绍 Nginx 基本内容和一些在 Nginx 上可以完成的简单任务。这里假设您已经安装了 nginx，否则请参阅 安装 nginx 页面。 本指南介绍如何启动、停止 nginx 和重新加载配置，解释配置文件的结构，并介绍如何设置 nginx 以提供静态内容服务，如何配置 nginx 作为代理服务器，以及如何将其连接到一个 FastCGI 应用程序。

nginx 有一个主进程（Master）和几个工作进程（Worker）。主进程的主要目的是读取和评估配置，并维护工作进程。工作进程对请求进行处理。nginx 采用了基于事件模型和依赖于操作系统的机制来有效地在工作进程之间分配请求。工作进程的数量可在配置文件中定义，并且可以针对给定的配置进行修改，或者自动调整到可用 CPU 内核的数量（请参阅 worker_processes）。

配置文件决定了 nginx 及其模块的工作方式。默认情况下，配置文件名为 `nginx.conf`，并放在目录 `/usr/local/nginx/conf`，`/etc/nginx` 或 `/usr/local/etc/nginx` 中。

启动、停止和重新加载配置

要启动 nginx，需要运行可执行文件。nginx 启动之后，可以通过调用可执行文件附带 `-s 参数` 来控制它。 使用以下语法：

```
nginx -s 信号
```

信号可能是以下之一：

- stop - 立即关闭
- quit - 正常关闭
- reload - 重新加载配置文件
- reopen - 重新打开日志文件

例如，要等待工作进程处理完当前的请求才停止 nginx 进程，可以执行以下命令：

```
nginx -s quit
```

> 这个命令的执行用户应该是与启动nginx用户是一致的

在将重新加载配置的命令发送到 nginx 或重新启动之前，配置文件所做的内容更改将不会生效。要重新加载配置，请执行：

```
nginx -s reload
```

一旦主进程（Master）收到要重新加载配置（reload）的信号，它将检查新配置文件的语法有效性，并尝试应用其中提供的配置。如果成功，主进程将启动新的工作进程（Worker），并向旧工作进程发送消息，请求它们关闭。否则，主进程回滚更改，并继续使用旧配置。旧工作进程接收到关闭命令后，停止接受新的请求连接，并继续维护当前请求，直到这些请求都被处理完成之后，旧工作进程将退出。

可以借助 Unix 工具（如 kill 工具）将信号发送到 nginx 进程，信号直接发送到指定进程 ID 的进程。默认情况下，nginx 主进程的进程 ID 是写入在 `/usr/local/nginx/logs` 或 `/var/run` 中的 nginx.pid 文件中。例如，如果主进程 ID 为 1628，则发送 QUIT 信号让 nginx 正常平滑关闭，可执行：

```
kill -s QUIT 1628
```

获取所有正在运行的 nginx 进程列表，可以使用 `ps` 命令，如下：

```
ps -ax | grep nginx
```

有关向 nginx 发送信号的更多信息，请参阅 控制 nginx。

配置文件结构

nginx 是由配置文件中指定的指令控制模块组成。指令可分为简单指令和块指令。一个简单的指令是由空格分隔的名称和参数组成，并以分号 `;` 结尾。块指令具有与简单指令相同的结构，但不是以分号结尾，而是以大括号 `{}` 包围的一组附加指令结尾。如果块指令的大括号内部可以有其它指令，则称这个块指令为上下文（例如：`events`，`http`，`server` 和 `location`）。

配置文件中被放置在任何上下文之外的指令都被认为是主上下文。`events` 和 `http` 指令在主上下文中，`server` 在 `http` 中，`location` 又在 `server` 中。

井号 `#` 之后的行的内容被视为注释。

提供静态内容服务

Web 服务器的一个重要任务是提供文件（比如图片或者静态 HTML 页面）服务。您将实现一个示例，根据请求，将提供来自不同的本地目录的文件：`/data/www`（可能包含 HTML 文件）和 `/data/images`（包含图片）。这需要编辑配置文件，在 `http` 中配置一个包含两个

`location` 块的 `server` 块指令。

```
http {
    server {
    }
}
```

通常，配置文件可以包含几个由监听端口和服务器域名区分的 `server` 块指令。一旦 nginx 决定由哪个 `server` 来处理请求，它会根据 `server` 块中定义的 `location` 指令的参数来检验请求头中指定的URI。

添加如下 `location` 块指令到 `server` 块指令中：

```
location / {
    root /data/www;
}
```

该 `location` 块指令指定 `/` 前缀与请求中的 URI 相比较。对于匹配的请求，URI 将被添加到根指令中指定的路径，即 `/data/ www`，以形成本地文件系统上所请求文件的路径。如果有几个匹配上的 `location` 块指令，nginx 将选择具有最长前缀的 `location` 块。上面的位置块提供最短的前缀，长度为 1，因此只有当所有其它 `location` 块不能匹配时，才会使用该块。

接下来，添加第二个 `location` 指令快：

```
location /images/ {
    root /data;
}
```

以 `/images/` 为开头的请求将会被匹配上（虽然 `location /` 也能匹配上此请求，但是它的前缀更短）

最后，`server` 块指令应如下所示：

```
server {
    location / {
        root /data/www;
    }

    location /images/ {
        root /data;
    }
}
```

这已经是一个监听标准 80 端口并且可以在本地机器上通过 `http://localhost/` 地址来访问的有效配置。响应以 `/images/` 开头的URI请求，服务器将从 `/data/images` 目录发送文件。例如，响应 `http://localhost/images/example.png` 请求，nginx 将发送 `/data/images/example.png` 文件。如果此文件不存在，nginx 将发送一个404错误响应。不以 `/ images/` 开头的 URI 的请求将映射到 `/data/www` 目录。例如，响应 `http://localhost/some/example.html` 请求，nginx 将发送 `/data/www/some/example.html` 文件。

要让新配置立刻生效，如果 nginx 尚未启动可以启动它，否则通过执行以下命令将重新加载配置信号发送到 nginx 的主进程：

```
nginx -s reload
```

> 如果运行的效果没有在预期之中，您可以尝试从 `/usr/local/nginx/logs` 或 `/var/log/nginx` 中的access.log和error.log日志文件中查找原因。

设置一个简单的代理服务器

nginx 的一个常见用途是作为一个代理服务器，作用是接收请求并转发给被代理的服务器，从中取得响应，并将其发送回客户端。

我们将配置一个基本的代理服务器，它为图片请求提供的文件来自本地目录，并将所有其它请求发送给代理的服务器。在此示例中，两个服务器在单个 nginx 实例上定义。

首先，通过向 nginx 的配置文件添加一个 `server` 块来定义代理服务器，其中包含以下内容：

```
server {
    listen 8080;
    root /data/up1;

    location / {
    }
}
```

这是一个监听 8080 端口的简单服务器（以前，由于使用了标准 80 端口，所以没有指定 listen 指令），并将所有请求映射到本地文件系统上的 `/data/up1` 目录。创建此目录并将 `index.html` 文件放入其中。请注意，`root` 指令位于 `server` 上下文中。当选择用于处理请求的 `location` 块不包含 `root` 指令时，将使用此 `root` 指令。

接下来，在上一节中的服务器配置基础上进行修改，使其成为代理服务器配置。在第一个 `location` 块中，使用参数指定的代理服务器的协议，域名和端口（在本例中为 `http://localhost:8080`）放置在 proxy_pass 指令处：

```
server {
    location / {
        proxy_pass http://localhost:8080;
    }

    location /images/ {
        root /data;
    }
}
```

我们将修改使用了 `/images/` 前缀将请求映射到 `/data/images` 目录下的文件的第二个 `location` 块，使其与附带常见的图片文件扩展名的请求相匹配。修改后的 `location` 块如下所示：

```
location ~ \.(gif|jpg|png)$ {
    root /data/images;
}
```

该参数是一个正则表达式，匹配所有以 `.gif` ，`.jpg` 或 `.png` 结尾的 URI。正则表达式之前应该是 `~` 。相应的请求将映射到 `/data/images` 目录。

当 nginx 选择一个 `location` 块来提供请求时，它首先检查指定前缀的 `location` 指令，记住具有最长前缀的 `location` ，然后检查正则表达式。如果与正则表达式匹配，nginx 会选择此 `location` ，否则选择更早之前记住的那一个。

代理服务器的最终配置如下：

```
server {
    location / {
        proxy_pass http://localhost:8080/;
    }

    location ~ \.(gif|jpg|png)$ {
        root /data/images;
    }
}
```

此 `server` 将过滤以 `.gif` ，`.jpg` 或 `.png` 结尾的请求，并将它们映射到 `/data/images` 目录（通过向 root 指令的参数添加 URI），并将所有其它请求传递到上面配置的代理服务器。

要使新配置立即生效，请将重新加载配置文件信号（reload）发送到 nginx，如前几节所述。

还有更多的指令可用于进一步配置代理连接。

设置 FastCGI 代理

nginx 可被用于将请求路由到运行了使用各种框架和 PHP 等编程语言构建的应用程序的 FastCGI 服务器。

与 FastCGI 服务器协同工作的最基本的 nginx 配置是使用 `fastcgi_pass` 指令而不是 `proxy_pass` 指令，以及 `fastcgi_param` 指令来设置传递给 FastCGI 服务器的参数。假设 FastCGI 服务器可以在 localhost:9000 上访问。以上一节的代理配置为基础，用 `fastcgi_pass` 指令替换 `proxy_pass` 指令，并将参数更改为 `localhost:9000`。在 PHP 中，`SCRIPT_FILENAME` 参数用于确定脚本名称，`QUERY_STRING` 参数用于传递请求参数。最终的配置将是：

```
server {
    location / {
        fastcgi_pass  localhost:9000;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param QUERY_STRING    $query_string;
    }

    location ~ \.(gif|jpg|png)$ {
        root /data/images;
    }
}
```

这里设置一个 `server`，将除了静态图片请求之外的所有请求路由到通过 FastCGI 协议在 `localhost:9000` 上运行的代理服务器。

原文档

- http://nginx.org/en/docs/beginners_guide.html

控制 nginx

可以用信号控制 nginx。默认情况下，主进程（Master）的 pid 写在 `/use/local/nginx/logs/nginx.pid` 文件中。这个文件的位置可以在配置时更改或者在 nginx.conf 文件中使用 `pid` 指令更改。Master 进程支持以下信号：

信号	作用
TERM, INT	快速关闭
QUIT	正常退出
HUP	当改变配置文件时，将有一段过渡时间段（仅 FreeBSD 和 Linux），新启动的 Worker 进程将应用新的配置，旧的 Worker 进程将被平滑退出
USR1	重新打开日志文件
USR2	升级可执行文件
WINCH	正常关闭 Worker 进程

Worker 进程也是可以用信号控制的，尽管这不是必须的。支持以下信号：

信号	作用
TERM, INT	快速关闭
QUIT	正常关闭
USR1	重新打开日志文件
WINCH	调试异常终止（需要开启 `debug_points` ）

配置变更

为了让 nginx 重新读取配置文件，应将 `HUP` 信号发送给 Master 进程。Master 进程首先会检查配置文件的语法有效性，之后尝试应用新的配置，即打开日志文件和新的 socket。如果失败了，它会回滚更改并继续使用旧的配置。如果成功，它将启动新的 Worker 进程并向旧的 Worker 进程发送消息请求它们正常关闭。旧的 Worker 进程关闭监听 socket 并继续为旧的客户端服务，当所有就的客户端被处理完成，旧的 Worker 进程将被关闭。

我们来举例说明一下。 想象一下，nginx 是在 FreeBSD 4.x 命令行上运行的

```
ps axw -o pid,ppid,user,%cpu,vsz,wchan,command | egrep '(nginx|PID)'
```

得到以下输出结果：

```
  PID  PPID USER    %CPU   VSZ WCHAN  COMMAND
33126     1 root     0.0  1148 pause  nginx: master process /usr/local/nginx/sbin/ngin
x
33127 33126 nobody   0.0  1380 kqread nginx: worker process (nginx)
33128 33126 nobody   0.0  1364 kqread nginx: worker process (nginx)
33129 33126 nobody   0.0  1364 kqread nginx: worker process (nginx)
```

如果把 `HUP` 信号发送到 master 进程，输出的结果将会是：

```
 PID  PPID USER    %CPU   VSZ WCHAN  COMMAND
33126     1 root     0.0  1164 pause  nginx: master process /usr/local/nginx/sbin/ngin
x
33129 33126 nobody   0.0  1380 kqread nginx: worker process is shutting down (nginx)
33134 33126 nobody   0.0  1368 kqread nginx: worker process (nginx)
33135 33126 nobody   0.0  1368 kqread nginx: worker process (nginx)
33136 33126 nobody   0.0  1368 kqread nginx: worker process (nginx)
```

其中一个 PID 为 33129 的 worker 进程仍然在继续工作，过一段时间之后它退出了：

```
PID  PPID USER    %CPU   VSZ WCHAN  COMMAND
33126     1 root     0.0  1164 pause  nginx: master process /usr/local/nginx/sbin/ngin
x
33134 33126 nobody   0.0  1368 kqread nginx: worker process (nginx)
33135 33126 nobody   0.0  1368 kqread nginx: worker process (nginx)
33136 33126 nobody   0.0  1368 kqread nginx: worker process (nginx)
```

日志轮转

为了做日志轮转，首先需要重命名。之后应该发送 `USR1` 信号给 master 进程。Master 进程将会重新打开当前所有的日志文件，并将其分配给一个正在运行未经授权的用户为所有者的 worker 进程。成功重新打开之后 Master 进程将会关闭所有打开的文件并且发送消息给 worker 进程要求它们重新打开文件。Worker 进程重新打开新文件和立即关闭旧文件。因此，旧的文件几乎可以立即用于后期处理，例如压缩。

升级可执行文件

为了升级服务器可执行文件，首先应该将新的可执行文件替换旧的可执行文件。之后发送 `USR2` 信号到 master 进程。Master 进程首先将以进程 ID 文件重命名为以 `.oldbin` 为后缀的新文件，例如 `/usr/local/nginx/logs/nginx.pid.oldbin`。之后启动新的二进制文件和依次期待能够新的 worker 进程：

```
  PID  PPID USER    %CPU   VSZ WCHAN  COMMAND
33126     1 root     0.0  1164 pause  nginx: master process /usr/local/nginx/sbin/ngin
x
33134 33126 nobody   0.0  1368 kqread nginx: worker process (nginx)
33135 33126 nobody   0.0  1380 kqread nginx: worker process (nginx)
33136 33126 nobody   0.0  1368 kqread nginx: worker process (nginx)
36264 33126 root     0.0  1148 pause  nginx: master process /usr/local/nginx/sbin/ngin
x
36265 36264 nobody   0.0  1364 kqread nginx: worker process (nginx)
36266 36264 nobody   0.0  1364 kqread nginx: worker process (nginx)
36267 36264 nobody   0.0  1364 kqread nginx: worker process (nginx)
```

之后所有的 worker 进程（旧的和新的）继续接收请求，如果 `WINCH` 信号被发送给了第一个 master 进程，它将向其 worker 进程发送消息要求它们正常关闭，之后它们开始退出：

```
  PID  PPID USER    %CPU   VSZ WCHAN  COMMAND
33126     1 root     0.0  1164 pause  nginx: master process /usr/local/nginx/sbin/ngin
x
33135 33126 nobody   0.0  1380 kqread nginx: worker process is shutting down (nginx)
36264 33126 root     0.0  1148 pause  nginx: master process /usr/local/nginx/sbin/ngin
x
36265 36264 nobody   0.0  1364 kqread nginx: worker process (nginx)
36266 36264 nobody   0.0  1364 kqread nginx: worker process (nginx)
36267 36264 nobody   0.0  1364 kqread nginx: worker process (nginx)
```

过一段时间，仅有新的 worker 进程处理请求：

```
  PID  PPID USER    %CPU   VSZ WCHAN  COMMAND
33126     1 root     0.0  1164 pause  nginx: master process /usr/local/nginx/sbin/ngin
x
36264 33126 root     0.0  1148 pause  nginx: master process /usr/local/nginx/sbin/ngin
x
36265 36264 nobody   0.0  1364 kqread nginx: worker process (nginx)
36266 36264 nobody   0.0  1364 kqread nginx: worker process (nginx)
36267 36264 nobody   0.0  1364 kqread nginx: worker process (nginx)
```

需要注意的是旧的 master 进程不会关闭它的监听 socket，并且如果需要的话，可以管理它来启动 worker 进程。如果出于某些原因不能接受新的可执行文件工作方式，可以执行以下操作之一：

- 发送 `HUP` 信号给旧的 master 进程，旧的 master 进程将会启动不会重新读取配置文件的 worker 进程。之后，通过将 `QUIT` 信号发送到新的主进程就可以正常关闭所有的新进程。
- 发送 `TERM` 信号到新的 master 进程，它将会发送一个消息给 worker 进程要求它们立即关闭，并且它们立即退出（如果由于某些原因新的进程没有退出，应该发送 `KILL` 信号让它们强制退出）。当新的 master 进程退出时，旧 master 将会自动启动新的 worker 进程。

新的 master 进程退出之后，旧的 master 进程会从以进程 ID 命名的文件中忽略掉 `.oldbin` 后缀的文件。

如果升级成功，应该发送 `QUIT` 信号给旧的 master 进程，仅仅新的进程驻留：

```
  PID  PPID USER    %CPU   VSZ WCHAN  COMMAND
36264     1 root     0.0  1148 pause  nginx: master process /usr/local/nginx/sbin/ngin
x
36265 36264 nobody   0.0  1364 kqread nginx: worker process (nginx)
36266 36264 nobody   0.0  1364 kqread nginx: worker process (nginx)
36267 36264 nobody   0.0  1364 kqread nginx: worker process (nginx)
```

原文档

- http://nginx.org/en/docs/control.html

连接处理方式

nginx 支持多种连接处理方式，每一种方式是否可用取决于所用的平台。在支持几种方式的平台上，nginx 会自动选择最有效的方式，然而，如果您需要明确指定使用哪一种方式，可以使用 `use` 指令指定。

支持以下集中处理方式：

- **select**，标准方式。当平台上缺乏其他有效的方式时自动构建。`--with-select-module` 和 `-without-select_module` 配置参数开启或者禁用此模块构建。
- **poll**，标准方式，当平台上缺乏其它有效的处理方式时自动构建此模块。`-with-poll_moudle` 和 `-without-poll_module` 配置项开启或者禁用此模块构建。
- **kqueue**，在 FreeBSD 4.1+, OpenBSD 2.9+, NetBSD 2.0, 和 macOS 使用有效。
- **epoll**，在 Linux 2.6+ 上使用有效。

> 从1.11.3起支持EPOLLRDHUP（Linux 2.6.17，glibc 2.8）和EPOLLEXCLUSIVE（Linux 4.5，glibc 2.24）标志。一些类似于SuSE 8.2这样的老版本提供了对2.4内核支持epll的补丁。

- **/dev/poll**，在 Solaris 7 11/99+，HP / UX 11.22+（eventport），IRIX 6.5.15+ 和 Tru64 UNIX 5.1A+ 有效。
- **eventport**，事件端口，在 Solaris 10+有效（由于已知问题，推荐使用 `/dev/poll` 方式代替）。

原文档

- http://nginx.org/en/docs/events.html

设置哈希

为了快速处理静态数据集，如服务器名称、map 指令值、MIME 类型、请求头字符串的名称，nginx 使用了哈希表。在开始和每次重新配置时，nginx 尽可能选择最小的哈希表，以便存储具有相同散列值的存储桶大小不会超过配置参数（哈希桶大小）。表的大小以桶为单位。调整是持续的，直到表的大小超过哈希的最大大小参数。大多数哈希具有修改这些参数对应的指令，例如，对于服务器名称哈希，它们是 server_names_hash_max 和 server_names_hash_bucket_size。

哈希桶大小参数与处理器的高速缓存线大小的倍数对齐。可以通过减少内存访问的次数来加快现代处理器的哈希键搜索速度。如果哈希桶大小等于处理器的高速缓存线大小，则哈希键搜索期间内存访问次数最坏的情况下将有两次 —— 一是计算桶地址，二是在桶内搜索哈希键期间。因此，如果 nginx 发出消息请求增加到哈希的最大大小或者哈希桶的大小，那么首先应该增加第一个参数。

原文档

- http://nginx.org/en/docs/hash.html

调试日志

要开启调试日志，需要在编译 Nignx 时增加如下配置：

```
./configure --with-debug ...
```

之后应该使用 `error_log` 指令设置调试级别：

```
error_log /path/to/log debug;
```

要验证 nginx 是否已经配置为支持调试功能，请运行 `nginx -V` 命令：

```
configure arguments: --with-debug ...
```

预构建 Linux 包为 nginx-debug 二进制文件的调试日志提供了开箱即用的支持，可以使用命令运行。

```
service nginx stop
service nginx-debug start
```

之后设置 `debug` 级别。Windows 的 nginx 在编译时就已经支持调试日志，因此只需设置 `debug` 级别即可。

请注意，重新定义日志而不指定 `debug` 级别将禁止调试日志。在下面的示例中，重新定义 `server` 上的日志级别，nginx 将不会在此服务器上做日志调试。

```
error_log /path/to/log debug;

http {
    server {
        error_log /path/to/log;
        ...
```

为了避免这种情况，重新定义日志级别应该被注释掉，或者明确指定日志为 `debug` 级别。

```
error_log /path/to/log debug;

http {
    server {
        error_log /path/to/log debug;
        ...
```

为指定客户端做调试日志

也可以仅为选定的客户端地址启用调试日志：

```
error_log /path/to/log;

events {
    debug_connection 192.168.1.1;
    debug_connection 192.168.10.0/24;
}
```

记录日志到循环内存缓冲区

调试日志可以被写入到循环内存缓冲区中：

```
error_log memory:32m debug;
```

在 `debug` 级别将日志写入到内存缓冲区中，即使在高负载情况下也不会对性能产生重大的影响。在这种情况下，可以使用如下 `gdb` 脚本来提取日志：

```
set $log = ngx_cycle->log

while $log->writer != ngx_log_memory_writer
    set $log = $log->next
end

set $buf = (ngx_log_memory_buf_t *) $log->wdata
dump binary memory debug_log.txt $buf->start $buf->end
```

原文档

- http://nginx.org/en/docs/debugging_log.html

记录日志到 syslog

`error_log` 和 `access_log` 指令支持把日志记录到 syslog。以下配置参数将使 nginx 日志记录到 syslog：

```
server=address
```

> 定义 syslog 服务器的地址，可以将该地址指定为附带可选端口的域名或者 IP，或者指定为 “unix:” 前缀之后跟着一个特定的 UNIX 域套接字路径。如果没有指定端口，则使用 UDP 的 514 端口。如果域名解析为多个 IP 地址，则使用第一个地址。

```
facility=string
```

> 设置 syslog 的消息 facility（设备），RFC3164 中定义，facility可以是 `kern`，`user`，`mail`，`daemon`，`auth`，`intern`，`lpr`，`news`，`uucp`，`clock`，`authpriv`，`ftp`，`ntp`，`audit`，`alert`，`cron`，`local0`，`local7` 中的一个，默认是 `local7`。

```
severity=string
```

> 设置 `access_log` 的消息严重程度，在 RFC3164 中定义。可能值与 `error_log` 指令的第二个参数（`level`，级别）相同，默认是 `info`。错误消息的严重程度由 nginx 确定，因此在 `error_log` 指令中将忽略该参数。

```
tag=string
```

> 设置 syslog 消息标签。默认是 `nginx`。

```
nohostname
```

> 禁止将 `hostname` 域添加到 syslog 的消息（1.9.7）头中。

syslog配置示例：

```
error_log syslog:server=192.168.1.1 debug;

access_log syslog:server=unix:/var/log/nginx.sock,nohostname;
access_log syslog:server=[2001:db8::1]:12345,facility=local7,tag=nginx,severity=info c
ombined;
```

记录日志到 syslog 的功能自从 1.7.2 版本开始可用。作为我们 商业订阅 的一部分，记录日志到 syslog 的功能从 1.5.3 开始可用。

原文档

- http://nginx.org/en/docs/syslog.html

配置文件度量单位

度量单位可以是字节、千字节（单位是 k 或者 K ）或者兆字节（单位是 m 或者 M ），例如，`1024` ，`8k` ，`1m` 。

也可以使用 g 或者 G 单位的千兆字节。

可以指定毫秒、秒、分、小时和天等时间来设定时间间隔，使用以下单位：

单位	含义
ms	毫秒
s	秒
m	分钟
h	小时
d	天
w	周
M	月，30天
y	年，365天

可以从大到小的顺序指定多个单位，用可选的空格符间隔组合成单个值。例如，`1h 30m` 与 `90m` 或者 `5400s` 时间相同。没有单位的值表示秒，建议始终加上单位。

某些时间设置只能区分秒级单位。

原文档

http://nginx.org/en/docs/syntax.html

命令行参数

nginx支持以下命令行参数：

- `-?` 或者 `-h` —— 打印命令行参数帮助信息
- `-c file` —— 使用指定的配置文件来替代默认的配置文件
- `-g directive` —— 设置 全局配置指令，例如：

```
nginx -g "pid /var/run/nginx.pid; worker_processes `sysctl -n hw.ncpu`;"
```

- `-p prefix` —— 设置 nginx 路径前缀，比如一个存放着服务器文件的目录（默认是 `/usr/local/nginx` ）。
- `-q` —— 在配置测试期间禁止非错误信息。
- `-s signal` —— 向 Master 进程发送信号，信号参数可以是以下：
  - `stop` —— 快速关闭。
  - `quit` —— 正常关闭。
  - `reload` —— 重新加载配置，使用新配置后启动新的 Worker 进程并正常退出旧的工作进程。
  - `reopen` —— 重新打开日志文件。
- `-t` —— 测试配置文件：nginx 检查配置文件的语法正确性，之后尝试打开配置中引用到的文件。
- `-T` —— 与 `-t` 一样，但另外将配置文件转储到标准输出（1.9.2）。
- `-v` —— 打印 nginx 版本。
- `-V` —— 打印 nginx 版本，编译器版本和配置参数。

原文档

http://nginx.org/en/docs/switches.html

Windows 下的 nginx

Nginx 的 Windows 版本使用了本地的 Win32 API（而不是 Cygwin 模拟层）。目前仅使用 `select()` 连接处理方式。由于此版本和其他存在已知的问题的 Nginx Windows 版本都被认为是 beta 版本，因此您不应该期望它具有高性能和可扩展性。现在，它提供了与 Unix 版本的 nginx 几乎相同的功能，除了 XSLT 过滤器、图像过滤器、GeoIP 模块和嵌入式 Perl 语言。

要安装 nginx 的 Windows 版本，请 下载 最新的主线发行版（1.13.4），因为 nginx 的主线分支包含了所有已知的补丁。之后解压文件到 `nginx-1.13.4` 目录下，然后运行 `nginx`。以下是 `C盘` 的根目录：

```
cd c:\
unzip nginx-1.13.4.zip
cd nginx-1.13.4
start nginx
```

运行 `tasklist` 命令行工具查看 nginx 进程：

```
C:\nginx-1.13.4>tasklist /fi "imagename eq nginx.exe"

Image Name                     PID Session Name        Session#    Mem Usage
========== ======== ================ =========== ============
nginx.exe                      652 Console                    0      2 780 K
nginx.exe                     1332 Console                    0      3 112 K
```

其中有一个是主进程（master），另一个是工作进程（worker）。如果 nginx 未能启动，请在错误日志 `logs\error.log` 中查找原因。如果日志文件尚未创建，可以在 Windows 事件日志中查找原因。如果显示的页面为错误页面，而不是预期结果，也可以在 `logs\error.log` 中查找原因。

Nginx 的 Windows 版本使用运行目录作为配置文件中的相对路径前缀。在上面的例子中，前缀是 `C:\nginx-1.13.4\`。在配置文件中的路径必须使类 Unix 风格的正斜杠：

```
access_log   logs/site.log;
root         C:/web/html;
```

Nginx 的 Windows 版本作为标准的控制台应用程序（而不是服务）运行，可以使用以下命令进行管理：

- `nginx -s stop` 快速退出
- `nginx -s quit` 正常退出

- `nginx -s reload` 重新加载配置文件，使用新的配置启动工作进程，正常关闭旧的工作进程
- `nginx -s reopen` 重新打开日志文件

已知问题

- 虽然可以启动多个工作进程，但实际上只有一个工作进程做完全部的工作
- 一个工作进程可以处理不超过 1024 个并发连接
- 不支持 UDP 代理功能

以后可能的发展

- 作为服务运行
- 使用 I/O 完成端口作为连接处理方式
- 在单个工作进程中使用多个工作线程

原文档

http://nginx.org/en/docs/windows.html

nginx 如何处理请求

基于名称的虚拟服务器

nginx 首先决定哪个 `server` 应该处理请求，让我们从一个简单的配置开始，三个虚拟服务器都监听了 `*:80` 端口：

```
server {
    listen      80;
    server_name example.org www.example.org;
    ...
}

server {
    listen      80;
    server_name example.net www.example.net;
    ...
}

server {
    listen      80;
    server_name example.com www.example.com;
    ...
}
```

在此配置中，nginx 仅检验请求的 header 域中的 `Host` ，以确定请求应该被路由到哪一个 `server` 。如果其值与任何的 `server` 名称不匹配，或者该请求根本不包含此 header 域，nginx 会将请求路由到该端口的默认 `server` 中。在上面的配置中，默认 `server` 是第一个（这是 nginx 的标准默认行为）。你也可以在 `listen` 指令中使用 `default_server` 参数，明确地设置默认的 `server` 。

```
server {
    listen      80 default_server;
    server_name example.net www.example.net;
    ...
}
```

> `default_server` 参数自 0.8.21 版本起可用。在早期版本中，应该使用 `default` 参数。

请注意， `default_server` 是 `listen port` 的属性，而不是 `server_name` 的。之后会有更多关于这方面的内容。

如何使用未定义的 server 名称来阻止处理请求

如果不允许没有 “Host” header 字段的请求，可以定义一个丢弃请求的 server：

```
server {
    listen      80;
    server_name "";
    return      444;
}
```

这里的 `server` 名称设置为一个空字符串，会匹配不带 `Host` 的 header 域请求，nginx 会返回一个表示关闭连接的非标准代码 444。

> 自 0.8.48 版本开始，这是 `server` 名称的默认设置，因此可以省略 `server name ""`。在早期版本中，机器的主机名被作为 `server` 的默认名称。

基于名称和 IP 混合的虚拟服务器

让我们看看更加复杂的配置，其中一些虚拟服务器监听在不同的 IP 地址上监听：

```
server {
    listen      192.168.1.1:80;
    server_name example.org www.example.org;
    ...
}

server {
    listen      192.168.1.1:80;
    server_name example.net www.example.net;
    ...
}

server {
    listen      192.168.1.2:80;
    server_name example.com www.example.com;
    ...
}
```

此配置中，nginx 首先根据 `server` 块的 `listen` 指令检验请求的 IP 和端口。之后，根据与 IP 和端口相匹配的 `server` 块的 `server_name` 项对请求的“Host” header 域进行检验。如果找不到服务器的名称（server_name），请求将由 `default_server` 处理。例如，在 `192.168.1.1:80` 上收到的对 `www.example.com` 的请求将由 `192.168.1.1:80` 端口的 `default_server`（即第一个 server）处理，因为没有 `www.example.com` 在此端口上定义。

如上所述，`default_server` 是 `listen port` 的属性，可以为不同的端口定义不同的 `default_server`：

```
server {
    listen      192.168.1.1:80;
    server_name example.org www.example.org;
    ...
}

server {
    listen      192.168.1.1:80 default_server;
    server_name example.net www.example.net;
    ...
}

server {
    listen      192.168.1.2:80 default_server;
    server_name example.com www.example.com;
    ...
}
```

一个简单的 PHP 站点配置

现在让我们来看看 nginx 是如何选择一个 `location` 来处理典型的简单 PHP 站点的请求：

```
server {
    listen      80;
    server_name example.org www.example.org;
    root        /data/www;

    location / {
        index   index.html index.php;
    }

    location ~* \.(gif|jpg|png)$ {
        expires 30d;
    }

    location ~ \.php$ {
        fastcgi_pass  localhost:9000;
        fastcgi_param SCRIPT_FILENAME
                      $document_root$fastcgi_script_name;
        include       fastcgi_params;
    }
}
```

nginx 首先忽略排序搜索具有最明确字符串的前缀 `location`。在上面的配置中，唯一有符合的是前缀 `location` 为 `/`，因为它匹配任何请求，它将被用作最后的手段。之后，nginx 按照配置文件中列出的顺序检查由 `location` 的正则表达式。第一个匹配表达式停止搜索，

nginx 将使用此 `location`。如果没有正则表达式匹配请求，那么 nginx 将使用前面找到的最明确的前缀 `location`。

请注意，所有类型的 `location` 仅仅是检验请求的 URI 部分，不带参数。这样做是因为查询字符串中的参数可以有多种形式，例如：

```
/index.php?user=john&page=1
/index.php?page=1&user=john
```

此外，任何人都可以在查询字符串中请求任何内容：

```
/index.php?page=1&something+else&user=john
```

现在来看看在上面的配置中是如何请求的：

- 请求 `/logo.gif` 首先与 前缀 `location` 为 `/` 相匹配，然后由正则表达式 `\.(gif|jpg|png)$` 匹配，因此由后一个 `location` 处理。使用指令 `root /data/www` 将请求映射到 `/data/www/logo.gif` 文件，并将文件发送给客户端。
- 一个 `/index.php` 的请求也是首先与前缀 `location` 为 `/` 相匹配，然后是正则表达式 `\.(php)$`。因此，它由后一个 `location` 处理，请求将被传递给在 `localhost:9000` 上监听的 FastCGI 服务器。`fastcgi_param` 指令将 FastCGI 参数 `SCRPT_FILENAME` 设置为 `/data/www/index.php`，FastCGI 服务器执行该文件。变量 `$document_root` 与 `root` 指令的值是一样的，变量 `$fastcgi_script_name` 的值为请求URI，即 `/index.php`。
- `/about.html` 请求仅与前缀 `location` 为 `/` 相匹配，因此由此 `location` 处理。使用指令 `root /data/www` 将请求映射到 `/data/www/about.html` 文件，并将文件发送给客户端。
- 处理请求 `/` 更复杂。它与前缀 `location` 为 `/` 相匹配。因此由该 `location` 处理。然后，`index` 指令根据其参数和 `root /data/www` 指令检验索引文件是否存在。如果文件 `/data/www/index.html` 不存在，并且文件 `/data/www/index.php` 存在，则该指令执行内部重定向到 `/index.php`，如果请求是由客户端发起的，nginx 将再次搜索 `location`。如之前所述，重定向请求最终由 FastCGI 服务器处理。

由 Igor Sysoev 撰写 由 Brian Mercer 编辑

原文

- http://nginx.org/en/docs/http/request_processing.html

服务器名称

服务器名称是使用 server_name 指令定义的，它确定了哪一个 server 块被给定的请求所使用。另请参见 nginx 如何处理请求。可以使用精确的名称、通配符或者正则表达式来定义他们：

```
server {
    listen       80;
    server_name  example.org  www.example.org;
    ...
}

server {
    listen       80;
    server_name  *.example.org;
    ...
}

server {
    listen       80;
    server_name  mail.*;
    ...
}

server {
    listen       80;
    server_name  ~^(?<user>.+)\.example\.net$;
    ...
}
```

当通过名称搜索虚拟服务器时，如果名称与多个指定的变体匹配，例如通配符和正则表达式，则将按照优先顺序选择第一个匹配的变体：

1. 精确的名称
2. 以星号开头的最长的通配符名称，例如 `*.example.org`
3. 以星号结尾的最长的通配符名称，例如 `mail.*`
4. 第一个匹配的正则表达式（按照在配置文件中出现的顺序）

通配符名称

通配符名称只能在名称的开头或者结尾包含一个星号，且只能在点的边界上包含星号。这些名称为 `www.*.example.org` 和 `w*.example.org` 都是无效的。然而，可以使用正则表达式指定这些名称，例如，`~^www\..+\.example\.org$”` 和 `~^w.*\.example\.org$` 。星号可以匹配多个

名称部分。名称 `*.example.org` 不仅匹配了 `www.example.org`，还匹配了 `www.sub.example.org`。

可以使用 `.example.org` 形式的特殊通配符名称来匹配确切的名称 `example.org` 和通配符 `*.example.org`。

正则表达式名称

nginx 使用的正则表达式与 Perl 编程语言（PCRE）使用的正则表达式兼容。要使用正则表达式，服务器名称必须以波浪符号开头：

```
server_name  ~^www\d+\.example\.net$;
```

否则将被视为确切的名称，或者如果表达式中包含星号，将被视为通配符名称（并且可能是无效的）。别忘了设置 `^` 和 `$` 锚点。它们在语法上不是必须的，但在逻辑上是需要的。还要注意的是，域名的点应该使用反斜杠转义。正则表达式中的 `{` 和 `}` 应该被引号括起：

```
server_name  "~^(?<name>\w\d{1,3}+)\.example\.net$";
```

否则 nginx 将无法启动并且显示错误信息：

```
directive "server_name" is not terminated by ";" in ...
```

正则表达式名称捕获到的内容可以作为变量为后面所用：

```
server {
    server_name   ~^(www\.)?(?<domain>.+)$;

    location / {
        root   /sites/$domain;
    }
}
```

PCRE 库使用以下语法捕获名称：

- `?<name>` Perl 5.10 兼容语法，自 PCRE-7.0 起支持
- `?'name'` Perl 5.10 兼容语法，自 PCRE-7.0 起支持
- `?P<name>` Python 兼容语法，自 PCRE-4.0 起支持

如果 nginx 无法启动并且显示错误信息：

```
pcre_compile() failed: unrecognized character after (?< in ...
```

说明 PCRE 库比较老，应该尝试使用语法 `?P<name>`。捕获也可以使用数字形式：

```
server {
    server_name   ~^(www\.)?(.+)$;

    location / {
        root   /sites/$2;
    }
}
```

然而，这种方式仅限于简单情况（如上所述），因为数字引用容易被覆盖。

其他名称

有一些服务器名称需要被特别处理。

如果需要处理一个在不是默认的 server 块中的没有“Host” header 域的请求，应该指定一个空名称：

```
server {
    listen       80;
    server_name  example.org  www.example.org  "";
    ...
}
```

如果 server 块中没有定义 server_name，那么 nginx 将使用空名称作为服务器名。

> 此情况下，nginx 的版本到 0.8.48 使用机器的主机名作为服务器名称。

如果服务器名称被定义为 `$hostname`（0.9.4），则使用机器的主机名。

如果有人使用 IP 地址而不是服务器名称来发出请求，则 “Host” 请求 header 域将包含 IP 地址，可以使用 IP 地址作为服务器名称来处理请求。

```
server {
    listen       80;
    server_name  example.org
                 www.example.org
                 ""
                 192.168.1.1
                 ;
    ...
}
```

在所有的服务器示例中，可以发现一个奇怪的名称 `_`：

```
server {
    listen       80  default_server;
    server_name  _;
    return       444;
}
```

这个名称没有什么特别之处，它只是众多无效域名中的一个，它永远不会与任何真实的名称相交。此外还有其他无效的名称，如 `--` 和 `!@#` 也是如此。

nginx 到 0.6.25 版本支持特殊的名称 `*`，这常被错误地理解为所有名称。它并不是所有或者通配符服务器名称。相反，它的功能现在是由 server_name_in_redirect 指令提供。现在不推荐使用特殊的名称 `*`，应该使用 server_name_in_redirect 指令。请注意，无法使用 server_name 指令来指定所有名称或者默认服务器。这是 listen 指令的属性，而不是 server_name 的。请参阅 nginx 如何处理请求。可以定义监听 `*:80` 和 `*:8080` 端口的服务器，并指定一个为 `*:8080` 端口的默认服务器，另一个为 `*:80` 端口的默认服务器。

```
server {
    listen       80;
    listen       8080  default_server;
    server_name  example.net;
    ...
}

server {
    listen       80  default_server;
    listen       8080;
    server_name  example.org;
    ...
}
```

优化

确切的名称、以星号开头的通配符名称和以星号结尾的通配符名称被存储在绑定到监听端口的三种哈希表中。哈希表的大小可以在配置阶段优化，因此可以在 CPU 缓存未命中的最低情况下找到名称。设置哈希表的具体细节在单独的文档中提供。

首先搜索确切的名称哈希表。如果为找到名称，则会搜索以星号开头的通配符名称的哈希表。如果还是没有找到名称，则搜索以星号结尾的通配符名称哈希。

搜索通配符哈希表比搜索确切名称的哈希表要慢，因为名称是通过域部分搜索的。请注意，特殊通配符格式 `.example.org` 存储在通配符哈希表中，而不是确切名称哈希表中。

由于正则表达式是按顺序验证的，因此是最慢的方法，并且是不可扩展的。

由于这些原因，最好是尽可能使用确切的名称。例如，如果最常见被请求的服务器名称是 `example.org` 和 `www.example.org`，则明确定义它们是最有效的：

```
server {
    listen       80;
    server_name  example.org  www.example.org  *.example.org;
    ...
}
```

简化形式：

```
server {
    listen       80;
    server_name  .example.org;
    ...
}
```

如果定义了大量的服务器名称，或者定义了非常长的服务器名称，则可能需要调整 `http` 级别的 server_names_hash_max_size 和 server_names_hash_bucket_size 指令。server_names_hash_bucket_size 指令的默认值可能等于 32 或者 64 或者其他值。具体取决于 CPU 超高速缓存储器线的大小。如果默认值为 32，并且服务器名称定义为 `too.long.server.name.example.org`，则 nginx 将无法启动并显示错误信息：

```
could not build the server_names_hash,
you should increase server_names_hash_bucket_size: 32
```

在这种情况下，指令值应该增加到 2 倍：

```
http {
    server_names_hash_bucket_size  64;
    ...
```

如果定义了大量的服务器名称，则会显示另一个错误信息：

```
could not build the server_names_hash,
you should increase either server_names_hash_max_size: 512
or server_names_hash_bucket_size: 32
```

在这种情况下，首先尝试将 server_names_hash_max_size 设置为接近服务器名称数量的数值。如果这样没有用，或者如果 nginx 的启动时间长的无法忍受，那么请尝试增加 server_names_hash_bucket_size 的值。

如果服务器是监听端口的唯一服务器，那么 nginx 将不会验证服务器名称（并且不会为监听端口建立哈希表）。但是，有一个例外。如果服务器名称具有捕获（captures）的正则表达式，则 nginx 必须执行表达式才能获取捕获。

兼容性

- 自 0.9.4 起，支持特殊的服务器名称 `$hostname` 。
- 自 0.8.48 起，默认服务器名称的值为空名称 `""` 。
- 自 0.8.25 其，已经支持命名的正则表达式服务器名称捕获。
- 自 0.7.40 起，支持正则表达式名称捕获。
- 自 0.7.12 起，支持空的服务器名称 `""` 。
- 自 0.6.25 起，支持使用通配符服务器名称和正则表达式作为第一服务器名称。
- 自 0.6.7 起，支持正则表达式服务名称。
- 自 0.6.0 起，支持类似 `example.*` 的通配符。
- 自 0.3.18 起，支持类似 `.example.org` 。
- 自 0.1.13 起，支持类似 `*.example.org` 的通配符。

由 **Igor Sysoev** 书写，由 **Brian Mercer** 编辑

原文

- http://nginx.org/en/docs/http/server_names.html

使用 nginx 作为 HTTP 负载均衡器

介绍

负载均衡跨越多个应用程序实例，是一种常用的技术，其用于优化资源利用率、最大化吞吐量、减少延迟和确保容错配置。

可以使用 nginx 作为高效的 HTTP 负载均衡器，将流量分布到多个应用服务器，并通过 nginx 提高 web 应用程序的性能、可扩展性和可靠性。

负载均衡方法

nginx 支持以下负载均衡机制（或方法）：

- 轮询（**round-robin**）- 发送给应用服务器的请求以轮询的方式分发
- 最少连接（**least-connected**）- 下一个请求被分配给具有最少数量活动连接的服务器
- **ip** 哈希（**ip-hash**）- 使用哈希函数确定下一个请求应该选择哪一个服务器（基于客户端的 IP 地址）

默认负载均衡配置

使用 nginx 进行负载均衡的最简单配置如下所示：

```
http {
    upstream myapp1 {
        server srv1.example.com;
        server srv2.example.com;
        server srv3.example.com;
    }

    server {
        listen 80;

        location / {
            proxy_pass http://myapp1;
        }
    }
}
```

在上述示例中，在 srv1-srv3 上运行相同的应用的三个实例。当负载均衡方法没有被特别配置时，默认采用轮询（round-robin）。所有请求都被 代理 到服务器组 myapp1，nginx 应用 HTTP 负载均衡来分发请求。

nginx 中的反向代理实现包括 HTTP、HTTPS、FastCGI、uwsgi、SCGI 和 memcached。

要配置 HTTPS 而不是 HTTP 负载均衡，只需要使用 HTTPS 协议。

在为 FastCGI、uwsgi、SCGI 或 memcached 设置负载均衡时，分别使用 fastcgi_pass、uwsgi_pass、scgi_pass 和 memcached_pass 指令。

最少连接负载均衡

另一个负载均衡的规则是最少连接。在一些请求需要更长的时间才能完成的情况下，最少连接可以更公正地控制应用程序实例的负载。

使用最少连接的负载均衡，nginx 将尽量不给过于繁忙的应用服务器负载过多的请求，而是将新的请求分发到不太忙的服务器。

当使用 least_conn 指令作为服务组配置的一部分时，将激活 nginx 中的最少连接负载均衡：

```
upstream myapp1 {
        least_conn;
        server srv1.example.com;
        server srv2.example.com;
        server srv3.example.com;
    }
```

会话持久化

请注意，使用轮询或者最少连接的负载均衡，每个后续客户端的请求都可能被分配到不同的服务器。不能保证同一个客户端始终指向同一个服务器。

如果需要将客户端绑定到特定的应用服务器，换而言之，使客户端会话「粘滞」或者「永久」，始终尝试选择特定的服务器，IP 哈希负载均衡机制可以做到这点。

使用 IP 哈希，客户端的 IP 地址用作为哈希键，以确定应用为客户端请求选择服务器组中的哪个服务器。此方法确保了来自同一个客户端的请求始终被定向到同一台服务器，除非该服务器不可用。

要配置 IP 哈希负载均衡，只需要将 ip_hash 指令添加到服务器 upstream 组配置中即可：

```
upstream myapp1 {
    ip_hash;
    server srv1.example.com;
    server srv2.example.com;
    server srv3.example.com;
}
```

加权负载均衡

还可以通过使用服务器权重进一步加强 nginx 的负载均衡算法。

在上面的示例中，服务器权重没有被配置，这意味对于特定的负载均衡方法来说所有指定的服务器都具有同等资格。

特别是使用轮询方式，这也意味着服务器上的请求分配或多或少都是相等的 —— 只要有足够的请求，并且以统一的方式足够快速地完成请求处理。

当服务器指定 weight 参数时，权重将作为负载均衡决策的一部分进行核算。

```
upstream myapp1 {
       server srv1.example.com weight=3;
       server srv2.example.com;
       server srv3.example.com;
   }
```

通过这样配置，每 5 个新的请求将分布在应用程序实例之中，如下所示：三个请求被定向到 srv1，一个请求将转到 srv2，另一个请求将转到 srv3。

在 nginx 的最近版本中，可以在最少连接和 IP 哈希负载均衡中使用权重。

健康检查

nginx 中的反向代理实现包括了带内（或者被动）服务器健康检查。如果特定服务器的响应失败并出现错误，则 nginx 会将此服务器标记为失败，并尝试避免为此后续请求选择此服务器而浪费一段时间。

max_files 用于设置在 fail_timeout 期间应该与服务器通信失败尝试的次数。默认情况下，max_files 设置为 1。当设置为 0 时，该服务器的健康检查将被禁用。fail_timeout 参数还定义了服务器被标记为失败的时间。在服务器发生故障后的 fail_timeout 间隔后，nginx 开始以实时客户端的请求优雅地探测服务器。如果探测成功，则将服务器标记为活动。

进一步阅读

此外，还有更多的指令和参数可以控制 nginx 中的服务器负载均衡，例如，proxy_next_upstream、backup、down 和 keepalive。有关更多的信息，请查看我们的参考文档。

最后但同样重要，应用程序负载均衡、应用程序健康检查、活动监控和服务组动态重新配置作为我们 NGINX Plus 付费订阅的一部分。

以下文章详细介绍了 NGINX Plus 负载均衡：

- NGINX 负载均衡与 NGINX Plus
- NGINX 负载均衡与 Nginx Plus 第2部分

原文档

- http://nginx.org/en/docs/http/load_balancing.html

配置 HTTPS 服务器

要配置 HTTPS 服务器，必须在 server 块中的 监听套接字 上启用 `ssl` 参数，并且指定服务器证书 和 私钥文件 的位置：

```
server {
    listen              443 ssl;
    server_name         www.example.com;
    ssl_certificate     www.example.com.crt;
    ssl_certificate_key www.example.com.key;
    ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers         HIGH:!aNULL:!MD5;
    ...
}
```

服务器证书是一个公共实体。它被发送到每个连接到服务器的客户端。私钥是一个安全实体，存储在一个访问受限的文件中，但是它对 nginx 的主进程必须是可读的。私钥也可以存储在与证书相同的文件中：

```
ssl_certificate     www.example.com.cert;
ssl_certificate_key www.example.com.cert;
```

这种情况下，文件的访问也应该被限制。虽然证书和密钥存储在一个文件中，但只有证书能被发送给客户端。

可以使用 ssl_protocols 和 ssl_ciphers 指令来限制连接，使其仅包括 SSL/TLS 的版本和密码。默认情况下，nginx 使用版本为 `ssl_protocols TLSv1 TLSv1.1 TLSv1.2` ，密码为 `ssl_ciphers HIGH:!aNULL:!MD5` ，因此通常不需要配置它们。请注意，这些指令的默认值已经被 更改 多次。

HTTPS 服务器优化

SSL 操作会消耗额外的 CPU 资源。在多处理器系统上，应该运行多个 工作进程（worker process），不得少于可用 CPU 核心的数量。大多数 CPU 密集型操作是发生在 SSL 握手时。有两种方法可以最大程度地减少每个客户端执行这些操作的次数。首先，启用 keepalive 连接，通过一个连接来发送多个请求，第二个是复用 SSL 会话参数，避免相同的和后续的连接发生 SSL 握手。会话存储在工作进程间共享的 SSL 会话缓存中，由 ssl_session_cache 指令配置。1MB 缓存包含约 4000 个会话。默认缓存超时时间为 5 分钟，可以用 ssl_session_timeout 指令来增加。以下是一个优化具有 10MB 共享会话缓存的多核系统的配置示例：

```
worker_processes auto;

http {
    ssl_session_cache   shared:SSL:10m;
    ssl_session_timeout 10m;

    server {
        listen              443 ssl;
        server_name         www.example.com;
        keepalive_timeout   70;

        ssl_certificate     www.example.com.crt;
        ssl_certificate_key www.example.com.key;
        ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
        ssl_ciphers         HIGH:!aNULL:!MD5;
        ...
```

SSL 证书链

某些浏览器可能会不承认由知名证书颁发机构签发的证书，而其他浏览器可能会接受该证书。之所以发生这种情况，是因为颁发机构已经在特定浏览器分发了一个中间证书，该证书不存在于知名可信证书颁发机构的证书库中。在这种情况下，权威机构提供了一系列链式证书，这些证书应该与已签名的服务器证书相连。服务器证书必须出现在组合文件中的链式证书之前：

```
$ cat www.example.com.crt bundle.crt > www.example.com.chained.crt
```

在 ssl_certificate 指令中使用生成的文件：

```
server {
    listen              443 ssl;
    server_name         www.example.com;
    ssl_certificate     www.example.com.chained.crt;
    ssl_certificate_key www.example.com.key;
    ...
}
```

如果服务器证书与捆绑的链式证书的相连顺序错误，nginx 将无法启动并显示错误消息：

```
SSL_CTX_use_PrivateKey_file(" ... /www.example.com.key") failed
   (SSL: error:0B080074:x509 certificate routines:
    X509_check_private_key:key values mismatch)
```

因为 nginx 已尝试使用私钥和捆绑的第一个证书而不是服务器证书。

浏览器通常会存储他们收到的中间证书，这些证书由受信任的机构签名，因此积极使用这些存储证书的浏览器可能已经具有所需的中间证书，不会发生不承认没有链接捆绑发送的证书的情况。可以使用 openssl 命令行工具来确保服务器发送完整的证书链，例如：

```
$ openssl s_client -connect www.godaddy.com:443
...
Certificate chain
 0 s:/C=US/ST=Arizona/L=Scottsdale/1.3.6.1.4.1.311.60.2.1.3=US
     /1.3.6.1.4.1.311.60.2.1.2=AZ/O=GoDaddy.com, Inc
     /OU=MIS Department/CN=www.GoDaddy.com
     /serialNumber=0796928-7/2.5.4.15=V1.0, Clause 5.(b)
   i:/C=US/ST=Arizona/L=Scottsdale/O=GoDaddy.com, Inc.
     /OU=http://certificates.godaddy.com/repository
     /CN=Go Daddy Secure Certification Authority
     /serialNumber=07969287
 1 s:/C=US/ST=Arizona/L=Scottsdale/O=GoDaddy.com, Inc.
     /OU=http://certificates.godaddy.com/repository
     /CN=Go Daddy Secure Certification Authority
     /serialNumber=07969287
   i:/C=US/O=The Go Daddy Group, Inc.
     /OU=Go Daddy Class 2 Certification Authority
 2 s:/C=US/O=The Go Daddy Group, Inc.
     /OU=Go Daddy Class 2 Certification Authority
   i:/L=ValiCert Validation Network/O=ValiCert, Inc.
     /OU=ValiCert Class 2 Policy Validation Authority
     /CN=http://www.valicert.com//emailAddress=info@valicert.com
...
```

在本示例中，www.GoDaddy.com 服务器证书 #0 的主体（“s”）由发行机构（“i”）签名，发行机构本身是证书 #1 的主体，是由知名发行机构 ValiCert, Inc. 签署的证书 #2 的主体，其证书存储在浏览器的内置证书库中。

如果没有添加证书包（certificate bundle），将仅显示服务器证书 #0。

HTTP/HTTPS 服务器

可以配置单个服务器来处理 HTTP 和 HTTPS 请求：

```
server {
    listen              80;
    listen              443 ssl;
    server_name         www.example.com;
    ssl_certificate     www.example.com.crt;
    ssl_certificate_key www.example.com.key;
    ...
}
```

> 在 0.7.14 之前，无法为各个 socket 选择性地启用 SSL，如上所示。只能使用 ssl 指令为整个服务器启用 SSL，从而无法设置单个 HTTP/HTTPS 服务器。可以通过添加 listen 指令的 ssl 参数来解决这个问题。因此，不建议在现在的版本中使用 ssl 指令。

基于名称的 HTTPS 服务器

当配置两个或多个 HTTPS 服务器监听单个 IP 地址时，会出现一个常见问题：

```
server {
    listen          443 ssl;
    server_name     www.example.com;
    ssl_certificate www.example.com.crt;
    ...
}

server {
    listen          443 ssl;
    server_name     www.example.org;
    ssl_certificate www.example.org.crt;
    ...
}
```

使用了此配置，浏览器会接收默认服务器的证书，即 www.example.com，而无视所请求的服务器名称。这是由 SSL 协议行为引起的。SSL连接在浏览器发送 HTTP 请求之前建立，nginx 并不知道请求的服务器名称。因此，它只能提供默认服务器的证书。

最古老、最强大的解决方法是为每个 HTTPS 服务器分配一个单独的 IP 地址：

```
server {
    listen          192.168.1.1:443 ssl;
    server_name     www.example.com;
    ssl_certificate www.example.com.crt;
    ...
}

server {
    listen          192.168.1.2:443 ssl;
    server_name     www.example.org;
    ssl_certificate www.example.org.crt;
    ...
}
```

具有多个名称的 SSL 证书

虽然还有其他方法允许在多个 HTTPS 服务器之间共享一个 IP 地址。但他们都有自己的缺点。一种方法是在 SubjectAltName 证书字段中使用多个名称的证书，例如 `www.example.com` 和 `www.example.org`。但是，SubjectAltName 字段长度有限。

另一种方法是使用附带通配符名称的证书，例如 `*.example.org`。通配符证书保护指定域的所有子域，但只能在同一级别上。此证书能匹配 `www.example.org`，但与 `example.org` 和 `www.sub.example.org` 不匹配。当然，这两种方法也可以组合使用的。SubjectAltName 字段中的证书可能包含确切名称和通配符名称，例如 `example.org` 和 `*.example.org`。

最好是将证书文件与名称、私钥文件放置在 http 级配置，以便在所有服务器中继承其单个内存副本：

```
ssl_certificate     common.crt;
ssl_certificate_key common.key;

server {
    listen          443 ssl;
    server_name     www.example.com;
    ...
}

server {
    listen          443 ssl;
    server_name     www.example.org;
    ...
```

服务器名称指示

在单个 IP 地址上运行多个 HTTPS 服务器的更通用的解决方案是 TLS 服务器名称指示扩展（SNI，RFC 6066），其允许浏览器在 SSL 握手期间传递所请求的服务器名称，因此，服务器将知道应该为此连接使用哪个证书。然而，SNI 对浏览器的支持是有限的。目前，它仅支持以下版本开始的浏览器：

- Opera 8.0
- MSIE 7.0（但仅在 Windows Vista 或更高版本）
- Firefox 2.0 和使用 Mozilla Platform rv:1.8.1 的其他浏览器
- Safari 3.2.1（支持 SNI 的 Windows 版本需要 Vista 或更高版本）
- Chrome（支持 SNI 的 Windows 版本需要 Vista 或更高版本）

> 只有域名可以在 SNI 中传递，然而，如果请求包含 IP 地址，某些浏览器可能会错误地将服务器的 IP 地址作为名称传递。不应该依靠这个。

要在 nginx 中使用 SNI，其必须支持构建后的 nginx 二进制的 OpenSSL 库以及在可在运行时动态链接的库。自 0.9.8f 版本起（OpenSSL），如果 OpenSSL 使用了配置选项 `--enable-tlsext` 构建，是支持 SNI 的。自 OpenSSL 0.9.8j 起，此选项是默认启用。如果 nginx 是用

SNI 支持构建的，那么当使用 `nginx -V` 命令时 ，nginx 会显示：

```
$ nginx -V
...
TLS SNI support enabled
...
```

但是，如果启用了 SNI 的 nginx 在没有 SNI 支持的情况下动态链接到 OpenSSL 库，那么 nginx 将会显示警告：

```
nginx was built with SNI support, however, now it is linked
dynamically to an OpenSSL library which has no tlsext support,
therefore SNI is not available
```

兼容性

- 从 0.8.21 和 0.7.62 起，SNI 支持状态通过 `-V` 开关显示。
- 从 0.7.14 开始，支持 listen 指令的 ssl 参数。在 0.8.21 之前，只能与 `default` 参数一起指定。
- 从 0.5.23 起，支持 SNI。
- 从 0.5.6 起，支持共享 SSL 会话缓存。
- 1.9.1 及更高版本：默认 SSL 协议为 TLSv1、TLSv1.1 和 TLSv1.2（如果 OpenSSL 库支持）。
- 0.7.65、0.8.19 及更高版本：默认 SSL 协议为 SSLv3、TLSv1、TLSv1.1 和 TLSv1.2（如果 OpenSSL 库支持）。
- 0.7.64、0.8.18 及更早版本：默认 SSL 协议为 SSLv2、SSLv3 和 TLSv1。
- 1.0.5及更高版本：默认 SSL 密码为 `HIGH:!aNULL:!MD5` 。
- 0.7.65、0.8.20 及更高版本：默认 SSL 密码为 `HIGH:!ADH:!MD5` 。
- 0.8.19 版本：默认 SSL 密码为 `ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM` 。
- 0.7.64、0.8.18 及更早版本：默认SSL密码为 `ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP` 。

由 **Igor Sysoev** 撰写，**Brian Mercer** 编辑

原文档

- Configuring HTTPS servers

nginx 如何处理 TCP/UDP 会话

来自客户端的 TCP/UDP 会话以阶段的形式被逐步处理：

阶段	描述
Post-accept	接收客户端请求后的第一个阶段。ngx_stream_realip_module 模块在此阶段被调用。
Pre-access	初步检查访问，ngx_stream_limit_conn_module 模块在此阶段被调用。
Access	实际处理之前的客户端访问限制，ngx_stream_access_module 模块在此阶段被调用。
SSL	TLS/SSL 终止，ngx_stream_ssl_module 模块在此阶段被调用。
Preread	将数据的初始字节读入 预读缓冲区 中，以允许如 ngx_stream_ssl_preread_module 之类的模块在处理前分析数据。
Content	实际处理数据的强制阶段，通常 代理 到 upstream 服务器，或者返回一个特定的值给客户端
Log	此为最后阶段，客户端会话处理结果将被记录， ngx_stream_log_module module 模块在此阶段被调用。

原文档

http://nginx.org/en/docs/stream/stream_processing.html

关于 nginScript

nginScript 是 JavaScript 语言的一个子集，其可在 http 和 stream 中实现位置（location）和变量处理器。nginScript 符合 ECMAScript 5.1 规范和部分 ECMAScript 6 扩展。合规性仍在不断发展。

目前支持什么

- 布尔值、数字、字符串、对象、数组、函数和正则表达式
- ES5.1 运算符，ES7 幂运算符
- ES5.1语句：`var` 、`if` 、`else` 、`switch` 、`for` 、`for in` 、`while` 、`do while` 、`break` 、`continue` 、`return` 、`try` 、`catch` 、`throw` 、`finally`
- ES6 `Number` 和 `Math` 的属性和方法
- `String` 方法：
  - ES5.1：`fromCharCode` 、`concat` 、`slice` 、`substring` 、`substr` 、`charAt` 、`charCodeAt` 、`indexOf` 、`lastIndexOf` 、`toLowerCase` 、`toUpperCase` 、`trim` 、`search` 、`match` 、`split` 、`replace`
  - ES6：`fromCodePoint` 、`codePointAt` 、`includes` 、`startsWith` 、`endsWith` 、`repeat`
  - 非标准：`fromUTF8` 、`toUTF8` 、`fromBytes` 、`toBytes`
- `Object` 方法：
  - ES5.1：`create` （不支持属性列表），`keys` 、`defineProperty` 、`defineProperties` 、`getOwnPropertyDescriptor` 、`getPrototypeOf` 、`hasOwnProperty` 、`isPrototypeOf` 、`preventExtensions` 、`isExtensible` 、`freeze` 、`isFrozen` 、`seal` 、`isSealed`
- `Array` 方法：
  - ES5.1：`isArray` 、`slice` 、`splice` 、`push` 、`pop` 、`unshift` 、`shift` 、`reverse` 、`sort` 、`join` 、`concat` 、`indexOf` 、`lastIndexOf` 、`forEach` 、`some` 、`every` 、`filter` 、`map` 、`reduce` 、`reduceRight`
  - ES6：`of` 、`fill` 、`find` 、`findIndex`
  - ES7：`includes`
- ES5.1 `Function` 方法：`call` 、`apply` 、`bind`
- ES5.1 `RegExp` 方法：`test` 、`exec`
- ES5.1 `Date` 方法
- ES5.1 全局函数：`isFinite` 、`isNaN` 、`parseFloat` 、`parseInt` 、`decodeURI` 、`decodeURIComponent` 、`encodeURI` 、`encodeURIComponent`

还不支持什么

- ES6 `let` 和 `const` 声明
- 标签
- `arguments` 数组
- `eval` 函数
- `JSON` 对象
- `Error` 对象
- `setTimeout` 、 `setInterval` 、 `setImmediate` 函数
- 非整数分数（.235），二进制（0b0101），八进制（0o77）字面量

下载与安装

nginScript 可用于以下两个模块：

- ngx_http_js_module
- ngx_stream_js_module

这两个模块都不是默认构建的，它们应该从源文件中编译或者作为一个 Linux 软件包来安装

Linux 包安装方式

在 Linux 环境中，可以使用 nginScript 模块包：

- `nginx-module-njs` - nginScript 动态模块
- `nginx-module-njs-dbg` - `nginx-module-njs` 包的调试符号

源码构建方式

可以使用以下命令克隆 nginScript 的源码仓库：（需要 Mercurial 客户端）：

```
hg clone http://hg.nginx.org/njs
```

然后使用 `--add-module` 配置参数进行编译模块：

```
./configure --add-module=path-to-njs/nginx
```

该模块也可以构建为动态的：

```
./configure --add-dynamic-module=path-to-njs/nginx
```

原文档

http://nginx.org/en/docs/njs_about.html

nginx：Linux 软件包

目前，nginx 软件支持以下 Linux 发行版：

RHEL/CentOS：

版本	支持平台
6.x	x86_64，i386
7.x	x86_64，ppc64le

Debian：

版本	代号	支持平台
7.x	wheezy	x86_64，i386
8.x	jessie	x86_64，i386
9.x	stretch	x86_64，i386

Ubuntu：

版本	代号	支持平台
12.04	precise	x86_64，i386
14.04	trusty	x86_64，i386, aarch64/arm64
16.04	xenial	x86_64，i386, ppc64el，aarch64/arm64
16.10	yakkety	x86_64，i386

SLES：

版本	支持平台
12	x86_64

要启用 Linux 软件包的自动更新，可设置 RHEL/CentOS 发行版的 yum 仓库（repository），Debian/Ubuntu 发行版的 apt 仓库或 SLES 的 zypper 仓库。

稳定版本的预构建软件包

要设置 RHEL/CentOS 的 yum 仓库，请创建名为 `/etc/yum.repos.d/nginx.repo` 的文件，其中包含以下内容：

```
[nginx]
name=nginx repo
baseurl=http://nginx.org/packages/OS/OSRELEASE/$basearch/
gpgcheck=0
enabled=1
```

根据所使用的发行版，使用 `rhel` 或 `centos` 替换掉 `OS`，对于 6.x 或 7.x 版本，将 `OSRELEASE` 替换为 `6` 或 `7`。

对于 Debian/Ubuntu，为了验证 nginx 仓库签名，并且在安装 nginx 软件包时消除关于缺少 PGP 密钥的警告，需要将用于将 nginx 软件包和仓库签署的密钥添加到 `apt` 程序密钥环中。请从我们的网站下载此密钥，并使用以下命令将其添加到 `apt` 程序密钥环：

```
sudo apt-key add nginx_signing.key
```

对于 Debian，使用 Debian 发行版代号替换掉 `codename`，并将以下内容追加到 `/etc/apt/sources.list` 文件末尾：

```
deb http://nginx.org/packages/debian/ codename nginx
deb-src http://nginx.org/packages/debian/ codename nginx
```

对于 Ubuntu，使用 Ubuntu 发行版代号替换掉 `codename`，并将以下内容追加到 `/etc/apt/sources.list` 文件末尾：

```
deb http://nginx.org/packages/ubuntu/ codename nginx
deb-src http://nginx.org/packages/ubuntu/ codename nginx
```

对于 Debian/Ubuntu，请运行以下命令：

```
apt-get update
apt-get install nginx
```

对于 SLES，运行以下命令：

```
zypper addrepo -G -t yum -c 'http://nginx.org/packages/sles/12' nginx
```

主线版本的预构建软件包

要设置 RHEL/CentOS 的 yum 仓库，请创建名为 `/etc/yum.repos.d/nginx.repo` 的文件，其中包含以下内容：

```
[nginx]
name=nginx repo
baseurl=http://nginx.org/packages/mainline/OS/OSRELEASE/$basearch/
gpgcheck=0
enabled=1
```

根据所使用的发行版，使用 `rhel` 或 `centos` 替换掉 `OS`，对于 6.x 或 7.x 版本，将 `OSRELEASE` 替换为 `6` 或 `7`。

对于 Debian/Ubuntu，为了验证 nginx 仓库签名，并且在安装 nginx 软件包时消除关于缺少 PGP 密钥的警告，必须将用于将 nginx 软件包和仓库签署的密钥添加到 `apt` 程序密钥环中。请从我们的网站下载此 密钥，并使用以下命令将其添加到 `apt` 程序密钥环：

```
sudo apt-key add nginx_signing.key
```

对于 Debian，使用 Debian 发行版代号替换 `codename`，并将以下内容追加到 `/etc/apt/sources.list` 文件末尾：

```
deb http://nginx.org/packages/mainline/debian/ codename nginx
deb-src http://nginx.org/packages/mainline/debian/ codename nginx
```

对于 Ubuntu，使用 Ubuntu 发行版代号替换 `codename`，并将以下内容追加到 `/etc/apt/sources.list` 文件末尾：

```
deb http://nginx.org/packages/mainline/ubuntu/ codename nginx
deb-src http://nginx.org/packages/mainline/ubuntu/ codename nginx
```

对于 Debian/Ubuntu，请运行以下命令：

```
apt-get update
apt-get install nginx
```

对于 SLES，请运行以下命令：

```
zypper addrepo -G -t yum -c 'http://nginx.org/packages/mainline/sles/12' nginx
```

源码包

源码包可以在 源码包库 中找到。

`default` 分支保存当前主线版本的源码包，而 `stable-*` 分支包含了稳定版本的最新源码。要构建二进制包，请在 Debian/Ubuntu 上的 `debian/` 目录或在 RHEL/CentOS/SLES 上的 `rpm/SPECS/` 中运行 `make`。

源码包在 类 BSD 的两项条款许可证 下发行，与 nginx 相同。

动态模块

主 nginx 包使用了所有模块进行构建，没有使用到附加库，以避免额外的依赖。自 1.9.11 版本开始，nginx 支持 动态模块，并将以下模块构建为动态模块，以独立软件包的形式发布：

```
nginx-module-geoip
nginx-module-image-filter
nginx-module-njs
nginx-module-perl
nginx-module-xslt
```

签名

RPM 软件包和 Debian/Ubuntu 仓库都使用数字签名来验证下载包的完整性和来源。为了检查签名，需要下载 nginx 签名密钥 并将其导入 rpm 或 apt 程序密钥环：

在 Debian/Ubuntu 上：

```
sudo apt-key add nginx_signing.key
```

在 RHEL/CentOS 上；

```
sudo rpm --import nginx_signing.key
```

在 SLES 上：

```
sudo rpm --import nginx_signing.key
```

Debian/Ubuntu/SLES 签名默认情况被检查，但是在 RHEL/CentOS 上，需要在 `/etc/yum.repos.d/nginx.repo` 文件中进行设置：

```
gpgcheck= 1
```

由于我们的 PGP 密钥 和软件包都位于同一台服务器上，因此它们都是可信的。强烈建议另行验证下载的 PGP 密钥的真实性。PGP 具有“Web of Trust”（信任网络）的概念，当一个密钥是由别人的密钥签署的，而另一个密钥则由另一个密钥签名。它通常可以建立一个从任意密钥到您知道和信任的某人的密钥，从而验证链中第一个密钥的真实性。这个概念在 GPG Mini Howto 中有详细描述。我们的钥匙有足够的签名，其真实性比较容易检查。

原文档

http://nginx.org/en/linux_packages.html#distributions

从源码构建 nginx

编译时使用 `configure` 命令进行配置。它定义了系统的各个方面，包括了 nginx 进行连接处理使用的方法。最终它会创建出一个 `Makefile` 。`configure` 命令支持以下参数：

- **--prefix=path**

  定义一个用于保留服务器文件的目录。此目录也将用于所有通过 `configure` 设置的相对路径（除了库源码路径外）和 `nginx.conf` 配置文件。默认设置为 `/usr/local/nginx` 目录。

- **--sbin-path=path**

  设置 nginx 可执行文件的名称。此名称仅在安装过程中使用。默认情况下，文件名为 `prefix/sbin/nginx` 。

- **--conf-path=path**

  设置 `nginx.conf` 配置文件的名称。如果需要，nginx 可以使用不同的配置文件启动，方法是使用命令行参数 `-c` 指定文件。默认情况下，文件名为 `prefix/conf/nginx.conf` 。

- **--pid-path=path**

  设置存储主进程的进程 ID 的 nginx.pid 文件名称。安装后，可以在 `nginx.conf` 配置文件中使用 pid 指令更改文件名。默认文件名为 `prefix/logs/nginx.pid` 。

- **--error-log-path=path**

  设置主要错误、警告和诊断文件的名称。安装后，可以在 `nginx.conf` 配置文件中使用 error_log 指令更改文件名。默认情况下，文件名为 `prefix/logs/error.log` 。

- **--http-log-path=path**

  设置 HTTP 服务器主请求日志文件名称。安装后，可以在 `nginx.conf` 配置文件中使用 access_log 指令更改文件名。默认情况下，文件名为 `prefix/logs/access.log` 。

- **--build=name**

  设置一个可选的 nginx 构建名称

- **--user=name**

  设置一个非特权用户名称，其凭据将由工作进程使用。安装后，可以在 `nginx.conf` 配置文件中使用 user 指令更改名称。默认的用户名为 `nobody` 。

- **--group=name**

设置一个组的名称，其凭据将由工作进程使用。安装后，可以在 `nginx.conf` 配置文件中使用 user 指令更改名称。默认情况下，组名称设置为一个非特权用户的名称。

- **--with-select_module 和 --without-select_module**

  启用或禁用构建允许服务器使用 `select()` 方法的模块。如果平台不支持其他更合适的方法（如 kqueue、epoll 或 /dev/poll），则将自动构建该模块。

- **--with-poll_module 和 --without-poll_module**

  启用或禁用构建允许服务器使用 `poll()` 方法的模块。如果平台不支持其他更合适的方法（如 kqueue、epoll 或 /dev/poll），则将自动构建该模块。

- **--without-http_gzip_module**

  禁用构建 HTTP 服务器响应压缩模块。需要 zlib 库来构建和运行此模块。

- **--without-http_rewrite_module**

  禁用构建允许 HTTP 服务器重定向请求和更改请求 URI 的模块。需要 PCRE 库来构建和运行此模块。

- **--without-http_proxy_module**

  禁用构建 HTTP 服务器代理模块。

- **--with-http_ssl_module**

  启用构建可将 HTTPS 协议支持添加到 HTTP 服务器的模块。默认情况下，此模块参与构建。构建和运行此模块需要 OpenSSL 库支持。

- **--with-pcre=path**

  设置 PCRE 库的源路径。发行版（4.4 至 8.40 版本）需要从 PCRE 站点下载并提取。其余工作由 nginx 的 `./configure` 和 `make` 完成。该库是 location 指令和 ngx_http_rewrite_module 模块中正则表达式支持所必需的。

- **--with-pcre-jit**

  使用“即时编译（just-in-time compilation）”支持（1.1.12版本的 pcre_jit 指令）构建 PCRE 库。

- **--with-zlib=path**

  设置 zlib 库的源路径。发行版（1.1.3 至 1.2.11 版本）需要从 zlib 站点下载并提取。其余工作由 nginx 的 `./configure` 和 `make` 完成。该库是 ngx_http_gzip_module 模块所必需的。

- **--with-cc-opt=parameters**

设置添加到 CFLAGS 变量的额外参数。当在 FreeBSD 下使用系统的 PCRE 库时，应指定 `--with-cc-opt="-I /usr/local/include"` 。如果需要增加 `select()` 所支持的文件数量，也可以在这里指定，如： `--with-cc-opt="-D FD_SETSIZE=2048"` 。

- **--with-ld-opt=parameters**

  设置链接期间使用的其他参数。在 FreeBSD 下使用系统 PCRE 库时，应指定--with-ld-opt="-L /usr/local/lib"`。

参数使用示例：

```
./configure \
    --sbin-path=/usr/local/nginx/nginx \
    --conf-path=/usr/local/nginx/nginx.conf \
    --pid-path=/usr/local/nginx/nginx.pid \
    --with-http_ssl_module \
    --with-pcre=../pcre-8.40 \
    --with-zlib=../zlib-1.2.11
```

配置完成之后，使用 `make` 和 `make install` 编译和安装 nginx。

原文档

http://nginx.org/en/docs/configure.html

在 Win32 平台上使用 Visual C 构建 nginx

先决条件

要在 Microsoft Win32® 平台上构建 nginx，您需要：

- Microsoft Visual C编译器。已知 Microsoft Visual Studio® 8 和 10 可以正常工作。
- MSYS。
- 如果您要构建 OpenSSL® 和有 SSL 支持的 nginx，则需要 Perl。例如 ActivePerl 或 Strawberry Perl。
- Mercurial 客户端
- PCRE、zlib 和 OpenSSL 库源代码。

构建步骤

在开始构建之前，确保将 Perl、Mercurial 和 MSYS 的 bin 目录路径添加到 PATH 环境变量中。从 Visual C 目录运行 vcvarsall.bat 脚本设置 Visual C 环境。

构建 nginx：

- 启动 MSYS bash。
- 检出 hg.nginx.org 仓库中的 nginx 源代码。例如：

```
hg clone http://hg.nginx.org/nginx
```

- 创建一个 build 和 lib 目录，并将 zlib、PCRE 和 OpenSSL 库源码解压到 lib 目录中：

```
mkdir objs
mkdir objs/lib
cd objs/lib
tar -xzf ../../pcre-8.41.tar.gz
tar -xzf ../../zlib-1.2.11.tar.gz
tar -xzf ../../openssl-1.0.2k.tar.gz
```

- 运行 configure 脚本：

```
auto/configure --with-cc=cl --builddir=objs --prefix= \
--conf-path=conf/nginx.conf --pid-path=logs/nginx.pid \
--http-log-path=logs/access.log --error-log-path=logs/error.log \
--sbin-path=nginx.exe --http-client-body-temp-path=temp/client_body_temp \
--http-proxy-temp-path=temp/proxy_temp \
--http-fastcgi-temp-path=temp/fastcgi_temp \
--with-cc-opt=-DFD_SETSIZE=1024 --with-pcre=objs/lib/pcre-8.41 \
--with-zlib=objs/lib/zlib-1.2.11 --with-openssl=objs/lib/openssl-1.0.2k \
--with-select_module --with-http_ssl_module
```

- 运行 make：

```
nmake -f objs/Makefile
```

相关内容

Windows 下的 nginx

原文档

http://nginx.org/en/docs/howto_build_on_win32.html

使用 DTrace pid 提供程序调试 nginx

本文假设读者对 nginx 内部原理和 DTrace 有了一定的了解。

虽然使用了 --with-debug 选项构建的 nginx 已经提供了大量关于请求处理的信息，但有时候更有必要详细地跟踪代码路径的特定部分，同时省略其余不必要的调试输出。DTrace pid 提供程序（在 Solaris，MacOS 上可用）是一个用于浏览用户程序内部的有用工具，因为它不需要更改任何代码，就可以帮助您完成任务。跟踪和打印 nginx 函数调用的简单 DTrace 脚本示例如下所示：

```
#pragma D option flowindent

pid$target:nginx::entry {
}

pid$target:nginx::return {
}
```

尽管如此，DTrace 的函数调用跟踪功能仅提供有限的有用信息。实时检查的功能参数通常更加有趣，但也更复杂一些。以下示例旨在帮助读者熟悉 DTrace 以及使用 DTrace 分析 nginx 行为的过程。

使用 DTrace 与 nginx 的常见方案之一是：附加到 nginx 的工作进程来记录请求行和请求开始时间。附加的相应函数是 `ngx_http_process_request()`，参数指向的是一个 `ngx_http_request_t` 结构的指针。使用 DTrace 脚本实现这种请求日志记录可以简单到：

```
pid$target::*ngx_http_process_request:entry
{
    this->request = (ngx_http_request_t *)copyin(arg0, sizeof(ngx_http_request_t));
    this->request_line = stringof(copyin((uintptr_t)this->request->request_line.data,
                                         this->request->request_line.len));
    printf("request line = %s\n", this->request_line);
    printf("request start sec = %d\n", this->request->start_sec);
}
```

需要注意的是，在上面的示例中，DTrace 需要引用 `ngx_http_process_request` 结构的一些相关信息。不幸的是，虽然可以在 DTrace 脚本中使用特定的 `#include` 指令，然后将其传递给 C 预处理器（使用 `-C` 标志），但这并不能真正起效。由于大量的交叉依赖，几乎所有的 nginx 头文件都必须包含在内。反过来，基于 `configure` 脚本设置，nginx 头将包括 PCRE、OpenSSL 和各种系统头文件。理论上，在 DTrace 脚本预处理和编译时，与特定的 nginx 构建相关的所有头文件都有可能被包含进来，实际上 DTrace 脚本很有可能由于某些头文件中的未知语法而造成无法编译。

上述问题可以通过在 DTrace 脚本中仅包含相关且必要的结构和类型定义来解决。DTrace 必须知道结构、类型和字段偏移的大小。因此，通过手动优化用于 DTrace 的结构定义，可以进一步降低依赖。

让我们使用上面的 DTrace 脚本示例，看看它需要哪些结构定义才能正常地工作。

首先应该包含由 configure 生成的 `objs/ngx_auto_config.h` 文件，因为它定义了一些影响各个方面的 `#ifdef` 常量。之后，一些基本类型和定义（如 `ngx_str_t`，`ngx_table_elt_t`，`ngx_uint_t` 等）应放在 DTrace 脚本的开头。这些定义经常被使用但不太可能经常改变的。

那里有一个包含许多指向其他结构的指针的 ngx_http_process_request_t 结构。因为这些指针与这个脚本无关，而且因为它们具有相同的大小，所以可以用 void 指针来替换它们。但最好添加合适的 typedef，而不是更改定义：

```
typedef ngx_http_upstream_t     void;
typedef ngx_http_request_body_t void;
```

最后但同样重要的是，需要添加两个成员结构的定义（`ngx_http_headers_in_t`，`ngx_http_headers_out_t`）、回调函数声明和常量定义。

最后，DTrace 脚本可以从 这里 下载。

以下示例是运行此脚本的输出：

```
# dtrace -C -I ./objs -s trace_process_request.d -p 4848
dtrace: script 'trace_process_request.d' matched 1 probe
CPU     ID                    FUNCTION:NAME
  1      4 .XAbmO.ngx_http_process_request:entry request line = GET / HTTP/1.1
request start sec = 1349162898

  0      4 .XAbmO.ngx_http_process_request:entry request line = GET /en/docs/nginx_dtr
ace_pid_provider.html HTTP/1.1
request start sec = 1349162899
```

使用类似的技术，读者应该能够跟踪其他 nginx 函数调用。

相关阅读

- Solaris 动态跟踪指南
- DTrace pid 提供程序介绍

原文档

http://nginx.org/en/docs/nginx_dtrace_pid_provider.html

转换重写规则

重定向到主站点

使用共享主机的用户以前仅使用 Apache 的 `.htaccess` 文件来配置一切，通常翻译下列规则：

```
RewriteCond  %{HTTP_HOST}  example.org
RewriteRule  (.*)          http://www.example.org$1
```

像这样：

```
server {
    listen       80;
    server_name  www.example.org  example.org;
    if ($http_host = example.org) {
        rewrite  (.*)  http://www.example.org$1;
    }
    ...
}
```

这是一个错误、麻烦而无效的做法。正确的方式是为 `example.org` 定义一个单独的服务器：

```
server {
    listen       80;
    server_name  example.org;
    return       301 http://www.example.org$request_uri;
}

server {
    listen       80;
    server_name  www.example.org;
    ...
}
```

在 0.9.1 之前的版本，重定向可以通过以下方式实现：

```
rewrite ^ http://www.example.org$request_uri?;
```

另一个例子是使用了颠倒逻辑，即 所有不是 **`example.com`** 和 **`www.example.com`** 的：

```
RewriteCond  %{HTTP_HOST}  !example.com
RewriteCond  %{HTTP_HOST}  !www.example.com
RewriteRule  (.*)          http://www.example.com$1
```

应该简单地定义 `example.com` 、 `www.example.com` 和 其他一切：

```
server {
    listen       80;
    server_name  example.com www.example.com;
    ...
}

server {
    listen       80 default_server;
    server_name  _;
    return       301 http://example.com$request_uri;
}
```

在 0.9.1 之前的版本，重定向可以通过以下方式实现：

```
rewrite ^ http://example.com$request_uri?;
```

转换 Mongrel 规则

典型的 Mongrel 规则：

```
DocumentRoot /var/www/myapp.com/current/public

RewriteCond %{DOCUMENT_ROOT}/system/maintenance.html -f
RewriteCond %{SCRIPT_FILENAME} !maintenance.html
RewriteRule ^.*$ %{DOCUMENT_ROOT}/system/maintenance.html [L]

RewriteCond %{REQUEST_FILENAME} -f
RewriteRule ^(.*)$ $1 [QSA,L]

RewriteCond %{REQUEST_FILENAME}/index.html -f
RewriteRule ^(.*)$ $1/index.html [QSA,L]

RewriteCond %{REQUEST_FILENAME}.html -f
RewriteRule ^(.*)$ $1.html [QSA,L]

RewriteRule ^/(.*)$ balancer://mongrel_cluster%{REQUEST_URI} [P,QSA,L]
```

应该转换为：

```
location / {
    root       /var/www/myapp.com/current/public;

    try_files  /system/maintenance.html
               $uri  $uri/index.html $uri.html
               @mongrel;
}

location @mongrel {
    proxy_pass  http://mongrel;
}
```

原文档

http://nginx.org/en/docs/http/converting_rewrite_rules.html

WebSocket 代理

要将客户端与服务器之间的连接从 HTTP/1.1 转换为 WebSocket，可是使用 HTTP/1.1 中的协议切换 机制。

然而，有一个微妙的地方：由于 `Upgrade` 是一个逐跳（hop-by-hop）头，它不会从客户端传递到代理服务器。当使用转发代理时，客户端可以使用 `CONNECT` 方法来规避此问题。然而，这不适用于反向代理，因为客户端不知道任何代理服务器，这需要在代理服务器上进行特殊处理。

自 1.3.13 版本以来，nginx 实现了特殊的操作模式，如果代理服务器返回一个 101响应码（交换协议），则客户机和代理服务器之间将建立隧道，客户端 通过请求中的 `Upgrade` 头来请求协议交换。

如上所述，包括 `Upgrade` 和 `Connection` 的逐跳头不会从客户端传递到代理服务器，因此为了使代理服务器知道客户端将协议切换到 WebSocket 的意图，这些头必须明确地传递：

```
location /chat/ {
    proxy_pass http://backend;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
}
```

一个更复杂的例子是，对代理服务器的请求中的 `Connection` 头字段的值取决于客户端请求头中的 `Upgrade` 字段的存在：

```
http {
    map $http_upgrade $connection_upgrade {
        default upgrade;
        ''      close;
    }

    server {
        ...

        location /chat/ {
            proxy_pass http://backend;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection $connection_upgrade;
        }
    }
```

默认情况下，如果代理务器在 60 秒内没有传输任何数据，连接将被关闭。这个超时可以通过 proxy_read_timeout 指令来增加。 或者，代理服务器可以配置为定期发送 WebSocket ping 帧以重置超时并检查连接是否仍然活跃。

原文档

http://nginx.org/en/docs/http/websocket.html

贡献变更

获取源码

Mercurial 是一个源码存储工具。可使用以下命令源码仓库：

```
hg clone http://hg.nginx.org/nginx
```

格式化变更

变更应根据 nginx 使用的代码样式进行格式化。代码格式化不应该依赖于诸如语法高亮或自动换行等编辑器功能。以下是一些基本规则：

- 最大文本宽度为 80 个字符
- 缩进为四个空格
- 没有 tab（制表符）
- 文件中的逻辑代码块用两行空行分隔

参照现有 nginx 源码的格式，在您的代码中模仿此样式。如果风格与周围的代码相一致，则更改更容易被接受。

提交更改以创建一个 Mercurial 变更集（changeset）。请确保指定的 电子邮件 和变更作者的真实姓名正确无误。

提交消息应是单行简述，后跟一行空行加描述内容。第一行最好不要超过 67 个符号。可以使用 `hg export` 命令获得一个结果变更集 patch：

```
# HG changeset patch
# User Filipe Da Silva <username@example.com>
# Date 1368089668 -7200
#      Thu May 09 10:54:28 2013 +0200
# Node ID 2220de0521ca2c0b664a8ea1e201ce1cb90fd7a2
# Parent  822b82191940ef309cd1e6502f94d50d811252a1
Mail: removed surplus ngx_close_connection() call.

It is already called for a peer connection a few lines above.

diff -r 822b82191940 -r 2220de0521ca src/mail/ngx_mail_auth_http_module.c
--- a/src/mail/ngx_mail_auth_http_module.c      Wed May 15 15:04:49 2013 +0400
+++ b/src/mail/ngx_mail_auth_http_module.c      Thu May 09 10:54:28 2013 +0200
@@ -699,7 +699,6 @@ ngx_mail_auth_http_process_headers(ngx_m

                     p = ngx_pnalloc(s->connection->pool, ctx->err.len);
                     if (p == NULL) {
-                        ngx_close_connection(ctx->peer.connection);
                         ngx_destroy_pool(ctx->pool);
                         ngx_mail_session_internal_server_error(s);
                         return;
```

提交前

提交更改前，有几点值得思考：

- 建议的变更应能在大部分的 支持平台 上正常工作。
- 尽量说明清楚为什么需要更改，如果可以的话，请提供一个用例。
- 通过测试套件传递您的更改是确保不会导致回归的好方法。可以使用以下命令克隆测试仓库：

```
hg clone http://hg.nginx.org/nginx-tests
```

提交变更

提议的更改应发送到 nginx 开发 邮件列表。提交更改集的首选便捷方法是使用 patchbomb 扩展。

许可证

提交变更意味着授予项目一个权限以在一个适当的许可证下使用它。

原文档

http://nginx.org/en/docs/contributing_changes.html

核心功能

示例配置

```
user www www;
worker_processes 2;

error_log /var/log/nginx-error.log info;

events {
    use kqueue;
    worker_connections 2048;
}

...
```

指令

accept_mutex

-	说明
语法	**accept_mutex** `on` | `off` ;
默认	accept_mutex off;
上下文	events

如果启用了 `accept_mutex` ，则工作进程将轮流接受新连接。否则，将新连接通知给所有工作进程，如果新连接数量较少，某些工作进程可能会浪费系统资源。

> 在支持 EPOLLEXCLUSIVE 标志（1.11.3）或使用 reuseport 的系统上，不需要启用 `accept_mutex` 。
>
> 在版本 1.11.3 之前，默认值为 `on` 。

accept_mutex_delay

-	说明
语法	**accept_mutex_delay** `time` ;
默认	accept_mutex_delay 500ms;
上下文	events

如果启用了 accept_mutex，则指定如果另一个工作进程正在接受新连接，则工作进程将尝试重新启动以接受新连接的最长时间。

daemon

-	说明
语法	**daemon** `on` | `off` ;
默认	daemon on;
上下文	main

决定 nginx 是否应该成为守护进程。主要用于开发。

debug_connection

-	说明
语法	**debug_connection** `address` | `CIDR` | `unix:` ;
默认	——
上下文	events

启用所选客户端连接的调试日志。其他连接将使用由 error_log 指令设置的日志级别。调试连接由 IPv4 或 IPv6 （1.3.0，1.2.1）地址或网络指定。也可以使用主机名指定连接。对于使用 UNIX 域套接字（1.3.0，1.2.1）的连接，调试日志由 `unix:` 参数启用。

```
events {
    debug_connection 127.0.0.1;
    debug_connection localhost;
    debug_connection 192.0.2.0/24;
    debug_connection ::1;
    debug_connection 2001:0db8::/32;
    debug_connection unix:;
    ...
}
```

> 为了使该指令正常工作，需要使用 `--with-debug` 来构建nginx，请参见调试日志。

debug_points

-	说明
语法	**debug_points** `abort` | `stop` ;
默认	——
上下文	main

该指令用于调试。

当检测到内部错误时，例如，在重新启动工作流程时，发生套接字泄漏，启用 `debug_points` 会导致核心文件创建（ `abort` ）或停止（ `stop` ）进程以使用系统调试器进行进一步分析。

error_log

-	说明
语法	**error_log** `file [level]`;
默认	error_log logs/error.log error;
上下文	main、http、mail、stream、server、location

配置日志。可以在同一级别指定几个日志（1.5.2）。如果在 `main` 配置级别将日志写入一个未明确定义文件，则将使用默认文件。

第一个参数定义了一个将存储日志的 `file`。特殊值 `stderr` 选择标准错误文件。可以通过指定 `syslog:` 前缀来配置记录日志到 syslog。可以通过指定 `memory:` 前缀和缓冲区 `size` 来配置记录日志到循环内存缓冲区，此通常用于调试（1.7.11）。

第二个参数确定日志记录的 `level`，可以是以下之一：`debug`、`info`、`notice`、`warn`、`error`、`crit`、`alert` 或 `emerg`。上述日志级别按严重性递增排列。设置某个日志级别将造成所有指定的消息和更严重的日志级别被记录。例如，默认级别 `error` 将导致 `error`、`crit`、`alert` 和 `emerg` 消息被记录。如果省略此参数，则使用 `error`。

> 为了使 `debug` 日志记录工作，需要使用 `--with-debug` 来构建nginx，请参见调试日志。
>
> 从 1.7.11 版本开始，该指令可以指定 `stream` 级别，从 1.9.0 版本开始可以指定 `mail` 级别。

env

-	说明
语法	**env** `variable[=value]`;
默认	env TZ;
上下文	main

默认情况下，nginx 除去 TZ 变量之外的所有继承自父进程的环境变量。该指令允许保留一些继承的变量、更改其值或创建新的环境变量。 这些变量是：

- 在可执行文件实时升级期间继承
- 被 ngx_http_perl_module 模块使用
- 被工作进程使用。应该要记住，以这种方式来控制系统库并不总是可行的，因为只有在初始化期间这些库才能检查变量。一种例外情况是可执行文件执行实时升级。

除非明确配置，否则 TZ 变量始终会继承并对 ngx_http_perl_module 模块可用。

用法示例：

```
env MALLOC_OPTIONS;
env PERL5LIB=/data/site/modules;
env OPENSSL_ALLOW_PROXY_CERTS=1;
```

> NGINX 环境变量由 nginx 内部使用，不应由用户直接设置。

events

-	说明
语法	**events** `{ ... }`
默认	——
上下文	main

提供指定影响连接处理指令的配置文件上下文。

include

-	说明
语法	**include** `file` | `mask` ;
默认	——
上下文	any

包含另一个 `file` 或匹配指定 `mask` 的文件到配置中。包含的文件应该由语法正确的指令和块组成。

使用示例：

```
include mime.types;
include vhosts/*.conf;
```

load_module

-	说明
语法	**load_module** `file` ;
默认	——
上下文	main
提示	该指令在 1.9.11 版本中出现

加载一个动态模块

使用示例：

```
load_module modules/ngx_mail_module.so;
```

-	说明
语法	**lock_file** `file` ;
默认	lock_file logs/nginx.lock;
上下文	main

nginx 使用锁机制来实现 accept_mutex 并序列化对共享内存的访问。大多数系统是使用原子操作实现锁，并忽略此指令。在其他系统上是使用“锁文件”机制。此指令用于指定一个锁文件名称的前缀。

master_process

-	说明
语法	**master_process** `on` | `off` ;
默认	master_process on;
上下文	main

用于决定是否启动工作进程。该指令适用于 nginx 开发人员。

multi_accept

-	说明
语法	**multi_accept** `on` | `off` ;
默认	multi_accept off;
上下文	events

如果 `multi_accept` 被禁用，工作进程将一次接受一个新连接。否则，工作进程将一次接受所有新连接。

> 如果使用了 kqueue 连接处理方式，则会忽略该指令，因为它报告了等待接受的新连接的数量。

pcre_jit

-	说明
语法	**pcre_jit** `on` | `off` ;
默认	pcre_jit off;
上下文	main
提示	该指令在 1.1.12 版本中出现

启用或禁用对在配置解析时已知的正则表达式使用“即时编译”（PCRE JIT）。

PCRE JIT 可以明显地加快正则表达式的处理。

> 从 8.20 版本开始，可以使用 `-enable-jit` 配置参数构建 PCRE 库启用 JIT。当使用 nginx（ `--with-pcre=` ）构建 PCRE 库时，可以通过 `--with-pcre-jit` 配置参数启用 JIT 支持。

pid

-	说明
语法	**pid** `file` ;
默认	——
上下文	main

定义一个 `file` 用于存储主进程的进程 ID 。

ssl_engine

-	说明
语法	**ssl_engine** `device` ;
默认	——
上下文	main

定义硬件 SSL 加速器的名称。

thread_pool

-	说明
语法	**thread_pool** `name threads=number [max_queue=number]` ;
默认	thread_pool default threads=32 max_queue=65536;
上下文	main
提示	该指令在 1.7.11 版本中出现

定义命名的线程池用于多线程读取和发送文件，而无需阻塞工作进程。

`threads` 参数定义池中的线程数量。

如果池中的所有线程都处于繁忙状态，则新任务将在队列中等待。`max_queue` 参数限制队列中允许等待的任务数。缺省情况下，队列中最多可以等待 65536 个任务。当队列溢出时，任务完成并出现错误。

timer_resolution

-	说明
语法	**timer_resolution** `interval` ;
默认	——
上下文	main

减少工作进程中的计时器分辨率，从而减少 `gettimeofday()` 的系统调用次数。默认情况下，每次接收到内核事件时都会调用 `gettimeofday()` 。随着分辨率的降低，`gettimeofday()` 仅在指定的时间间隔内被调用一次。

示例：

```
timer_resolution 100ms;
```

间隔的内部实现取决于使用的方法：

- 如果使用 `kqueue` ，就使用 `EVFILT_TIMER` 过滤器
- 如果使用 `eventport` ，就使用 `timer_create()`
- 否则使用 `setitimer()`

use

-	说明
语法	**use** `method` ;
默认	——
上下文	events

指定要使用的连接处理方式。通常不需要明确指定它，因为 nginx 将默认使用最有效的方式。

user

-	说明
语法	**user** `user [group]` ;
默认	——
上下文	events

定义工作进程使用的 `user` 和 `group` 凭据。如果省略 `group` ，则使用其名称与 `user` 相等的组。

worker_aio_requests

-	说明
语法	**worker_aio_requests** `number` ;
默认	worker_aio_requests 32;
上下文	events
提示	该指令在 1.1.4 和 1.0.7 版本中出现

当使用有 epoll 连接处理方式的 aio 时，为单个工作进程设置未完成异步 I/O 操作的最大 `number` （数量）。

worker_connections

-	说明
语法	**worker_connections** `number` ;
默认	worker_connections 512;
上下文	events

设置工作进程可以打开的同时连接的最大数量。

应该记住的是，这个数字包括了所有连接（例如与代理服务器的连接等），而不仅仅是客户端的连接。另一个要考虑因素是实际的并发连接数不能超过最大打开文件数的限制，可以通过 worker_rlimit_nofile 来修改。

worker_cpu_affinity

-	说明
语法	**worker_cpu_affinity** `cpumask ...` ; **worker_cpu_affinity** `auto [cpumask]` ;
默认	——
上下文	main

将工作进程绑定到一组 CPU。每个 CPU 组由允许的 CPU 的位掩码表示。应为每个工作进程定义一个单独的组。默认情况下，工作进程没有绑定任何特定的 CPU。

示例：

```
worker_processes    4;
worker_cpu_affinity 0001 0010 0100 1000;
```

将每个工作进程绑定到单独的 CPU

```
worker_processes    2;
worker_cpu_affinity 0101 1010;
```

将第一个工作进程绑定到 CPU0/CPU2，将第二个工作进程绑定到 CPU1/CPU3。第二个例子适用于超线程。

特殊值 `auto` （1.9.10）允许自动绑定工作进程到可用的 CPU：

```
worker_processes auto;
worker_cpu_affinity auto;
```

可选的掩码参数可用于限制自动绑定的可用 CPU：

```
worker_cpu_affinity auto 01010101;
```

> 该指令仅适用于 FreeBSD 和 Linux。

worker_priority

-	说明
语法	**worker_priority** `number` ;
默认	worker_priority 0;
上下文	main

定义工作进程的调度优先级，与使用 `nice` 命令完成类似：负数意味着更高的优先级。允许范围通常在 -20 到 20 之间。

示例：

```
worker_priority -10;
```

worker_processes

-	说明
语法	**worker_processes** `number` | `auto` ;
默认	worker_processes 1;
上下文	main

定义工作进程的数量。

最优值取决于许多因素，包括（但不限于）CPU 核心数、存储数据的硬盘驱动器数量以及加载模式。当您不确定时，将其设置为可用 CPU 核心的数量是一个不错的做法（值 `auto` 将尝试自动检测）。

> `auto` 参数从 1.3.8 和 1.2.5 版本开始得到支持。

worker_rlimit_core

-	说明
语法	**worker_rlimit_core** `size` ;
默认	——
上下文	main

更改工作进程核心文件（ `RLIMIT_CORE` ）最大大小限制。用于增加限制而无需重新启动主进程。

worker_rlimit_nofile

-	说明
语法	**worker_rlimit_nofile** `number` ;
默认	——
上下文	main

更改工作进程最大打开文件数（ `RLIMIT_NOFILE` ）的限制。用于增加限制而无需重新启动主进程。

worker_shutdown_timeout

-	说明
语法	**worker_shutdown_timeout** `time` ;
默认	——
上下文	main
提示	该指令在 1.11.11 版本中出现

为正常关闭工作进程配置超时。当时间到期时，nginx 将尝试关闭当前打开的所有连接以便于关闭工作进程。

working_directory

-	说明
语法	**working_directory** `directory` ;
默认	——
上下文	main

定义工作进程的当前工作目录。它主要用于编写核心文件时，在这种情况下，工作进程应有指定目录的写入权限。

原文档

http://nginx.org/en/docs/ngx_core_module.html

ngx_http_core_module

- location
- log_not_found
- log_subrequest
- max_ranges
- merge_slashes
- msie_padding
- msie_refresh
- open_file_cache
- open_file_cache_errors
- open_file_cache_min_uses
- open_file_cache_valid
- output_buffers
- port_in_redirect
- postpone_output
- read_ahead
- recursive_error_pages
- request_pool_size
- reset_timedout_connection
- resolver
- resolver_timeout
- root
- satisfy
- send_lowat
- send_timeout
- sendfile
- sendfile_max_chunk
- server
- server_name
- server_name_in_redirect
- server_names_hash_bucket_size
- server_names_hash_max_size
- server_tokens
- tcp_nodelay
- tcp_nopush
- try_files
- types
- types_hash_bucket_size
- types_hash_max_size
- underscores_in_headers
- variables_hash_bucket_size

指令

absolute_redirect

-	说明
语法	**absolute_redirect** `on` | `off` ;
默认	absolute_redirect on;
上下文	http、server、location
提示	该指令在 1.11.8 版本中出现

如果禁用，nginx 发出的重定向将是相对的。

另请参阅 server_name_in_redirect 和 port_in_redirect 指令。

aio

-	说明
语法	**aio** `on` | `off` | `threads[=pool]` ;
默认	aio off;
上下文	http、server、location
提示	该指令在 0.8.11 版本中出现

启用或禁用在 FreeBSD 和 Linux 上使用异步文件 I/O（AIO）：

```
location /video/ {
    aio            on;
    output_buffers 1 64k;
}
```

在 FreeBSD 上，AIO 从 FreeBSD 4.3 开始可用。在 FreeBSD 11.0 之前，可以将 AIO 静态链接到内核中：

```
options VFS_AIO
```

或作为一个内核可加载模块动态加载：

```
kldload aio
```

在 Linux 上，AIO 从内核版本为 2.6.22 开始可用。此外，有必要启用 directio，否则读取将被阻塞：

```
location /video/ {
    aio            on;
    directio       512;
    output_buffers 1 128k;
}
```

在 Linux 上，directio 只能用于读取 512 字节边界对齐的块（或 XFS 4K）。文件未对齐的末端以阻塞模式读取。对于字节范围请求和不是从文件开头开始的 FLV 请求也是如此：在文件的开头和结尾读取未对齐的数据将被阻塞。

当在 Linux 上启用 AIO 和 sendfile 时，AIO 用于大于或等于 directio 指令指定大小的文件，而 sendfile 用于较小的文件（禁用 directio 时也是如此）：

```
location /video/ {
    sendfile       on;
    aio            on;
    directio       8m;
}
```

最后，可以使用多线程（1.7.11）读取和发送文件，不会阻塞工作进程：

```
location /video/ {
    sendfile       on;
    aio            threads;
}
```

读取和发送文件操作被卸载到指定池中的线程。如果省略池名称，则使用名称为 `default` 的池。池名也可以用变量设置：

```
aio threads=pool$disk;
```

默认情况下，多线程是禁用状态，它应该使用 `--with-threads` 配置参数启用。目前，多线程仅与 epoll、kqueue 和 eventport 方式兼容。仅在 Linux 上支持多线程发送文件。

另请参见 sendfile 指令。

aio_write

-	说明
语法	**aio_write** `on` | `off` ;
默认	aio_write off;
上下文	http、server、location
提示	该指令在 1.9.13 版本中出现

如果启用 aio，则指定是否写入文件。目前，这仅在使用 aio 线程时有效，并且仅限于将从代理服务器接收的数据写入临时文件。

alias

-	说明
语法	**alias** `path` ;
默认	——
上下文	location

定义指定 location 的替换。例如，使用以下配置

```
location /i/ {
    alias /data/w3/images/;
}
```

`/i/top.gif` 的请求，将发送 `/data/w3/images/top.gif` 文件。

`path` 值可以包含变量，除 `$document_root` 和 `$realpath_root` 外。

如果在使用正则表达式定义的 location 内使用了别名，那么这种正则表达式应该包含捕获，并且别名应该引用这些捕获（0.7.40），例如：

```
location ~ ^/users/(.+\.(?:gif|jpe?g|png))$ {
    alias /data/w3/images/$1;
}
```

当 location 与指令值的最后一部分匹配时：

```
location /images/ {
    alias /data/w3/images/;
}
```

更好的方式是使用 root 指令：

```
location /images/ {
    root /data/w3;
}
```

chunked_transfer_encoding

-	说明
语法	**chunked_transfer_encoding** `on` | `off` ;
默认	chunked_transfer_encoding on;
上下文	http、server、location

允许在 HTTP/1.1 中禁用分块传输编码。在使用的软件未能支持分块编码时它可能会派上用场。

client_body_buffer_size

-	说明	
语法	**client_body_buffer_size** `size` ;	
默认	client_body_buffer_size 8k	16k;
上下文	http、server、location	

设置读取客户端请求体的缓冲区大小。如果请求体大于缓冲区，则整个体或仅将其部分写入临时文件。默认情况下，缓冲区大小等于两个内存页。在 x86、其他 32 位平台和 x86-64 上是 8K。在其他 64 位平台上通常为 16K。

client_body_in_file_only

-	说明
语法	**client_body_in_file_only** `on` | `clean` | `off` ;
默认	client_body_in_file_only off;
上下文	http、server、location

确定 nginx 是否应将整个客户端请求体保存到文件中。可以在调试期间或使用 `$request_body_file` 变量或模块 ngx_http_perl_module 的 $r->request_body_file 方法时使用此指令。

当设置为 `on` 值时，临时文件在请求处理后不会被删除。

值 `clean` 会将请求处理后剩下的临时文件删除。

client_body_in_single_buffer

-	说明
语法	**client_body_in_single_buffer** `on` | `off` ;
默认	client_body_in_single_buffer off;
上下文	http、server、location

确定 nginx 是否应将整个客户端请求体保存在单个缓冲区中。在使用 `$request_body` 变量时，建议使用该指令，用来保存涉及的复制操作次数。

client_body_temp_path

-	说明
语法	**client_body_temp_path** `path [level1 [level2 [level3]]]` ;
默认	client_body_temp_path client_body_temp;
上下文	http、server、location

定义用于存储持有客户端请求主体的临时文件的目录。最多可以在指定目录下使用三级子目录层次结构。例如以下配置

```
client_body_temp_path /spool/nginx/client_temp 1 2;
```

临时文件的路径可以如下：

```
/spool/nginx/client_temp/7/45/00000123457
```

client_body_timeout

-	说明
语法	**client_body_timeout** `time` ;
默认	client_body_timeout 60s;
上下文	http、server、location

定义读取客户端请求正文的超时时间。超时设置仅是在两个连续读操作之间的时间间隔，而不是整个请求体的传输过程。如果客户端在此时间内没有发送任何内容，则会将 408（请求超时）错误返回给客户端。

client_header_buffer_size

-	说明
语法	**client_header_buffer_size** `size` ;
默认	client_header_buffer_size 1k;
上下文	http、server

设置读取客户端请求头的缓冲区大小。对于大多数请求，1K 字节的缓冲区就足够了。但是，如果请求中包含长 cookie，或者来自 WAP 客户端，则可能 1K 是不适用的。如果请求行或请求头域不适合此缓冲区，则会分配由 large_client_header_buffers 指令配置的较大缓冲区。

client_header_timeout

-	说明
语法	**client_header_timeout** `time` ;
默认	client_header_timeout 60s;
上下文	http、server

定义读取客户端请求头的超时时间。如果客户端在这段时间内没有传输整个报头，则将 408（请求超时）错误返回给客户端。

client_max_body_size

-	说明
语法	**client_header_timeout** `size` ;
默认	client_max_body_size 1m;
上下文	http、server、location

在 **Content-Length** 请求头域中指定客户端请求体的最大允许大小。如果请求的大小超过配置值，则将 413（请求实体过大）错误返回给客户端。请注意，浏览器无法正确显示此错误。将 `size` 设置为 0 将禁用检查客户端请求正文大小。

connection_pool_size

-	说明	
语法	**connection_pool_size** `size` ;	
默认	connection_pool_size 256	512;
上下文	http、server	

允许精确调整每个连接的内存分配。该指令对性能影响最小，一般不建议使用。默认情况下，32 位平台上的大小为 256 字节，64 位平台上为 512 字节。

> 在 1.9.8 版本之前，所有平台上的默认值都为 256。

default_type

-	说明
语法	**default_type** `mime-type` ;
默认	default_type text/plain;
上下文	http、server、location

定义响应的默认 MIME 类型。可以使用 types 指令对 MIME 类型的文件扩展名进行映射。

directio

-	说明	
语法	**directio** `size` \	`off` ;
默认	directio off;	
上下文	http、server、location	

开启当读取大于或等于指定大小的文件时，使用 `O_DIRECT` 标志（FreeBSD、Linux）、`F_NOCACHE` 标志（macOS）或 `directio()` 函数（Solaris）。该指令自动禁用（0.7.15）给定请求使用的 sendfile。它可以用于服务大文件：

```
directio 4m;
```

或当在 Linux 上使用 aio 时。

directio_alignment

-	说明
语法	**directio_alignment** `size` ;
默认	directio_alignment 512;
上下文	http、server、location
提示	该指令在 0.8.11 版本中出现

设置 directio 的对齐方式。在大多数情况下，512 字节的对齐就足够了。但是，在 Linux 下使用 XFS 时，需要增加到 4K。

disable_symlinks

-	说明	
语法	**disable_symlinks** `off` ; **disable_symlinks on** \	`if_not_owner [from=part]` ;
默认	disable_symlinks off;	
上下文	http、server、location	
提示	该指令在 1.1.15 版本中出现	

确定打开文件时应如何处理符号链接：

- `off`

  允许路径名中的符号链接的，不执行检查。这是默认行为。

- `on`

  如果路径名存在是符号链接的组件，则拒绝对文件的访问。

- `if_not_owner`

  如果路径名存在是符号链接的组件，并且链接指向的链接和对象为不同所有者，则拒绝访问文件。

- `from=part`

  当检查符号链接（参数 `on` 和 `if_not_owner`）时，通常会检查路径名的所有组件。可以通过另外指定 `from=part` 参数来避免检查路径名的初始部分中的符号链接。在这种情况下，只能从指定的初始部分后面的路径名组件检查符号链接。如果该值不是检查的路径名的初始部分，则会检查整个路径名，相当于没有指定此参数。如果该值与整个文件名匹配，则不会检查符号链接。参数值可以包含变量。

例如：

```
disable_symlinks on from=$document_root;
```

此指令仅适用于具有 `openat()` 和 `fstatat()` 接口的系统。这样的系统包括现代版本的 FreeBSD、Linux 和 Solaris。

参数 `on` 和 `if_not_owner` 增加了处理开销。

> 在不支持打开目录仅用于搜索的系统上，要使用这些参数，需要 worker 进程对所有正在检查的目录具有读取权限。
>
> ngx_http_autoindex_module、ngx_http_random_index_module 和 ngx_http_dav_module 模块目前忽略此指令。

error_page

-	说明
语法	**error_page** `code ... [=[response]] uri`;
默认	——
上下文	http、server、location、location 中的 if

定义针对指定错误显示的 URI。`uri` 值可以包含变量。

例如：

```
error_page 404             /404.html;
error_page 500 502 503 504 /50x.html;
```

这会导致客户端请求方法更改为 `GET`，内部重定向到指定的 `uri`（除 `GET` 和 `HEAD` 之外的所有方法）。

此外，可以使用 `=response` 语法将修改响应代码，例如：

```
error_page 404 =200 /empty.gif;
```

如果错误响应是由代理服务器或 FastCGI/uwsgi/SCGI 服务器处理，并且服务器可能返回不同的响应代码（例如，200、302、401 或 404），则可以使用其返回的代码进行响应：

```
error_page 404 = /404.php;
```

如果在内部重定向时不需要更改 URI 和方法，则可以将错误处理传递给一个命名了的 location：

```
location / {
    error_page 404 = @fallback;
}

location @fallback {
    proxy_pass http://backend;
}
```

> 如果 `uri` 处理导致发生错误，最后发生错误的状态代码将返回给客户端。

也可以使用 URL 重定向进行错误处理：

```
error_page 403      http://example.com/forbidden.html;
error_page 404 =301 http://example.com/notfound.html;
```

在这种情况下，默认将响应代码 302 返回给客户端。它只能是重定向状态代码其中之一（301、302、303、307 和 308）。

> 在 1.1.16 和 1.0.13 版本之前，代码 307 没有被视为重定向。
>
> 直到 1.13.0 版本，代码 308 才被视为重定向。

当且仅当没有在当前级别上定义 error_page 指令时，指令才从上一级继承这些特性。

etag

-	说明	
语法	**etag** `on` |	`off`;
默认	etag on;	
上下文	http、server、location	
提示	该指令在 1.3.3 版本中出现	

启用或禁用自动生成静态资源 **ETag** 响应头字段。

http

-	说明
语法	**http** `{ ... }`;
默认	——
上下文	main

提供指定 HTTP server 指令的配置文件上下文。

if_modified_since

-	说明		
语法	**if_modified_since** `off` |	`exact` |	`before`;
默认	if_modified_since exact;		
上下文	http、server、location		
提示	该指令在 0.7.24 版本中出现		

指定如何将响应的修改时间与 **If-Modified-Since** 请求头字段中的时间进行比较：

- `off`

  忽略 **If-Modified-Since** 请求头字段（0.7.34）

- `exact`

  完全匹配

- `before`

  响应的修改时间小于或等于 **If-Modified-Since** 请求头字段中的时间

ignore_invalid_headers

-	说明	
语法	**ignore_invalid_headers** `on` |	`off`;
默认	ignore_invalid_headers on;	
上下文	http、server	

控制是否应忽略具有无效名称的头字段。有效名称由英文字母、数字、连字符或下划线组成（由 underscores_in_headers 指令控制）。

如果在 server 级别指定了该指令，则其值仅在 server 为默认 server 时使用。指定的值也适用于监听相同地址和端口的所有虚拟服务器。

internal

-	说明
语法	**internal**;
默认	——
上下文	location

指定给定的 location 只能用于内部请求。对于外部请求，返回客户端错误 404（未找到）。内部请求如下：

- 请求由 error_page、index、random_index 和 try_files 指令重定向
- 来 upstream server 的 **X-Accel-Redirect** 响应头字段重定向的请求
- 由 ngx_http_ssi_module 模块的 `include virtual` 命令、ngx_http_addition_module 模块指令和 auth_request 和 mirror 指令组成的子请求
- 由 rewrite 指令更改的请求

示例：

```
error_page 404 /404.html;

location /404.html {
    internal;
}
```

> 每个请求限制 10 个内部重定向，以防止不正确配置引发的请求处理死循环。如果达到此限制，则返回错误 500（内部服务器错误）。在这种情况下，可以在错误日志中看到 `rewrite or internal redirection cycle` 消息。

keepalive_disable

-	说明	
语法	**keepalive_disable** `none` \	`browser ...` ;
默认	keepalive_disable msie6;	
上下文	http、server、location	

禁用与行为异常的浏览器保持连接。`browser` 参数指定哪些浏览器将受到影响。一旦接收到 POST 请求，值 `msie6` 将禁用与旧版本 MSIE 保持连接。值 `safari` 禁用与 macOS 和类似 MacOS 的操作系统上的 Safari 和 Safari 浏览器保持连接。值 `none` 启用与所有浏览器保持连接。

> 在 1.1.18 版本之前，值 `safari` 匹配所有操作系统上的所有 Safari 和类 Safari 浏览器，并且默认情况下，禁用与它们保持连接。

keepalive_requests

-	说明
语法	**keepalive_requests** `number` ;
默认	keepalive_requests 100;
上下文	http、server、location
提示	该指令在 0.8.0 版本中出现

设置通过一个保持活动（keep-alive）连接可以提供的最大请求数。在发出的请求达到最大数量后，连接将被关闭。

keepalive_timeout

-	说明
语法	**keepalive_timeout** `timeout` `[header_timeout]` ;
默认	keepalive_timeout 75s;
上下文	http、server、location

第一个参数设置一个超时时间，keep-alive 客户端连接将在服务端保持打开状态。零值将禁用 keep-alive 客户端连接。可选的第二个参数在 **Keep-Alive: timeout=** `time` 响应头域中设置一个值。两个参数可能不同。

Mozilla 和 Konqueror 会识别 **Keep-Alive: timeout= `time` 头字段。MSIE 在大约在 60 秒钟内自行关闭 keep-alive 连接。

large_client_header_buffers

-	说明
语法	**large_client_header_buffers** `number size` ;
默认	arge_client_header_buffers 4 8k;
上下文	http、server

设置用于读取大客户端请求头的缓冲区的最大 `number`（数量）和 `size`（大小）。请求行不能超过一个缓冲区的大小，否则将返回 414（请求 URI 太长）错误给客户端。请求头字段也不能超过一个缓冲区的大小，否则将返回 400（错误请求）错误给客户端。缓冲区只能按需分配。默认情况下，缓冲区大小等于 8K 字节。如果在请求处理结束之后，连接被转换为 keep-alive 状态，这些缓冲区将被释放。

limit_except

-	说明
语法	**limit_except** `method ... { ... }`;
默认	——
上下文	location

限制给定的 location 内允许的 HTTP 方法。`method` 参数可以是以下之一：`GET` 、`HEAD` 、`POST` 、`PUT` 、`DELETE` 、`MKCOL` 、`COPY` 、`MOVE` 、`OPTIONS` 、`PROPFIND` 、`PROPPATCH` 、`LOCK` 、`UNLOCK` 或 `PATCH` 。允许 `GET` 方法也将使得 `HEAD` 方法被允许。可以使用 ngx_http_access_module 和 ngx_http_auth_basic_module 模块指令来限制对其他方法的访问：

```
limit_except GET {
    allow 192.168.1.0/32;
    deny  all;
}
```

请注意，这将限制访问除 `GET` 和 `HEAD` 之外的所有方法。

limit_rate

-	说明
语法	**limit_rate** `rate`;
默认	limit_rate 0;
上下文	http、server、location、location 中的 if

限制客户端的响应传输速率。`rate` 以字节/秒为单位。零值将禁用速率限制。限制设置是针对每个请求，因此如果客户端同时打开两个连接，则整体速率将是指定限制的两倍。

也可以在 `$limit_rate` 变量中设置速率限制。根据某些条件来限制速率可能会有用：

```
server {

    if ($slow) {
        set $limit_rate 4k;
    }

    ...
}
```

也可以在代理服务器响应的 **X-Accel-Limit-Rate** 头字段中设置速率限制。可以使用 proxy_ignore_headers、fastcgi_ignore_headers、uwsgi_ignore_headers 和 scgi_ignore_headers 指令禁用此功能。

lingering_time

-	说明
语法	**lingering_time** `time` ;
默认	lingering_time 30s;
上下文	http、server、location

当 lingering_close 生效时，该指令指定 nginx 处理（读取和忽略）来自客户端的额外数据的最长时间。之后，连接将被关闭，即使还有更多的数据。

lingering_timeout

-	说明
语法	**lingering_timeout** `time` ;
默认	lingering_timeout 5s;
上下文	http、server、location

当 lingering_close 生效时，该指令指定更多的客户端数据到达的最长等待时间。如果在此期间未收到数据，则关闭连接。否则，读取和忽略数据，nginx 再次开始等待更多的数据。**wait-read-ignore** 循环被重复，但不再由 lingering_time 指令指定。

listen

-	说明				
语法	**listen** `address[:port] [default_server] [ssl] [http2 \	spdy] [proxy_protocol] [setfib=number] [fastopen=number] [backlog=number] [rcvbuf=size] [sndbuf=size] [accept_filter=filter] [deferred] [bind] [ipv6only=on\	off] [reuseport] [so_keepalive=on\	off\	[keepidle]: [keepintvl]: [keepcnt]] ; **listen** pc [default_serv [ssl] [http2 \
默认	listen *:80 \	*:8000;			
上下文	server				

设置 IP 的 `address` （地址）和 `port` （端口），或服务器接受请求的 UNIX 域套接字的 `path` （路径）。可以同时指定 `addres` （地址）和 `port` （端口），也可以只指定 `address` （地址）或 `port` （端口）。`address` 也可以是主机名，例如：

```
listen 127.0.0.1:8000;
listen 127.0.0.1;
listen 8000;
listen *:8000;
listen localhost:8000;
```

IPv6 地址（0.7.36）在方括号中指定：

```
listen [::]:8000;
listen [::1];
```

UNIX 域套接字（0.8.21）用 `unix:` 前缀指定：

```
listen unix:/var/run/nginx.sock;
```

如果只指定 `address` ，则使用 80 端口。

如果指令不存在，那么如果 nginx 是以超级用户权限运行，则使用 `*:80` ，否则使用 `*:8000` 。

`default_server` 参数（如果存在）将使得服务器成为指定 `address:port` 对的默认服务器。如果没有指令有 `default_server` 参数，那么具有 `address:port` 对的第一个服务器将是该对的默认服务器。

> 在 0.8.21 之前的版本中，此参数简单地命名为 `default` 。

`ssl` 参数（0.7.14）允许指定该端口上接受的所有连接都应该工作在 SSL 模式。这样可以为处理 HTTP 和 HTTPS 请求的服务器提供更紧凑的配置。

`http2` 参数（1.9.5）配置端口接受 HTTP/2 连接。通常，为了能够工作，还应该指定 `ssl` 参数，但也可以将 nginx 配置为接受没有 SSL 的 HTTP/2 连接。

`spdy` 参数（1.3.15 — 1.9.4）允许在此端口上接受 SPDY 连接。通常，为了能够工作，还应该指定 `ssl` 参数，但也可以将 nginx 配置为接受没有 SSL 的 SPDY 连接。

`proxy_protocol` 参数（1.5.12）允许指定此端口上接受的所有连接都应使用 PROXY 协议。

`listen` 指令可以有特定于套接字相关系统调用的几个附加参数。这些参数可以在任何 `listen` 指令中指定，但对于给定的 `address:port` 只能使用一次。

> 在 0.8.21 之前的版本中，它们只能在 `listen` 指令中与默认参数一起指定。

- `setfib=number`

  此参数（0.8.44）设置相关联的路由表，监听套接字的 FIB（`SO_SETFIB` 选项）。目前只适用于 FreeBSD。

- `fastopen=number`

  为侦听套接字启用 **TCP Fast Open**（1.5.8），并限制尚未完成三次握手的连接队列的最大长度。

  > 不要启用此功能，除非服务器可以一次处理接收多个相同的 SYN 包的数据。

- `backlog=number`

  在 `listen()` 调用中设置限制挂起连接队列的最大长度的 `backlog` 参数。默认情况下，FreeBSD、DragonFly BSD 和 macOS 上的 `backlog` 设置为 -1，在其他平台上设置为 511。

- `rcvbuf=size`

  设置侦听套接字的接收缓冲区大小（`SO_RCVBUF` 选项）。

- `sndbuf=size`

  设置侦听套接字的发送缓冲区大小（`SO_SNDBUF` 选项）。

- `accept_filter=filter`

  为侦听套接字设置接受过滤器（`SO_ACCEPTFILTER` 选项）的名称，该套接字将传入的连接在传递给 `accept()` 之前进行过滤。这只适用于 FreeBSD 和 NetBSD 5.0+。可设置值dataready 或 httpready。

- `deferred`

  指示在 Linux 上使用延迟 `accept()`（`TCP_DEFER_ACCEPT` 套接字选项）。

- `bind`

  指示为给定的 `address:port` 对单独进行 `bind()` 调用。这是有用的，因为如果有多个有相同端口但不同地址的 `listen` 指令，并且其中一个 `listen` 指令监听给定端口（`*:port`）的所有地址，则 nginx 将 `bind()` 应用到 `*:port`。应该注意的是，在这种情况下将会进行 `getsockname()` 系统调用，以确定接受该连接的地址。如果使用 `setfib`、`backlog`、`rcvbuf`、`sndbuf`、`accept_filter`、`deferred`、`ipv6only` 或 `so_keepalive` 参数，那么对于给定的 `address:port` 对将始终进行单独的 `bind()` 调用。

- `ipv6only=ON|OFF`

此参数（0.7.42）确定（通过 `IPV6_V6ONLY` 套接字选项）一个监听通配符地址 `[::]` 的 IPv6 套接字是否仅接受 IPv6 连接或 IPv6 和 IPv4 连接。此参数默认为开启。它只能在启动时设置一次。

> 在 1.3.4 版本之前，如果省略该参数，则操作系统的设置将对于套接字产生影响。

- `reuseport`

  此参数（1.9.1）指示为每个工作进程创建一个单独的监听套接字（使用 `SO_REUSEPORT` 套接字选项），允许内核在工作进程之间分配传入的连接。目前只适用于 Linux 3.9+ 和 DragonFly BSD。

  > 不当地使用此选项可能会带来安全隐患。

- `SO_KEEPALIVE=ON|OFF|[keepidle]:[keepintvl]:[keepcnt]`

  此参数（1.1.11）配置监听套接字的 **TCP keepalive** 行为。如果省略此参数，则操作系统的设置将对套接字产生影响。如果设置为 `on`，则套接字将打开 `SO_KEEPALIVE` 选项。如果设置为 `off`，则 `SO_KEEPALIVE` 选项将被关闭。一些操作系统支持在每个套接字上使用 `TCP_KEEPIDLE`、`TCP_KEEPINTVL` 和 `TCP_KEEPCNT` 套接字选项来设置 TCP keepalive 参数。在这样的系统上（目前为 Linux 2.4+、NetBSD 5+ 和 FreeBSD 9.0-STABLE），可以使用 `keepidle`、`keepintvl` 和 `keepcnt` 参数进行配置。可以省略一个或两个参数，在这种情况下，相应套接字选项的系统默认设置将生效。例如，

  ```
  SO_KEEPALIVE=30m::10
  ```

  将空闲超时（`TCP_KEEPIDLE`）设置为 30 分钟，将探测间隔（`TCP_KEEPINTVL`）设置为系统默认值，并将探测数量（`TCP_KEEPCNT`）设置为 10 个探测。

示例：

```
listen 127.0.0.1 default_server accept_filter=dataready backlog=1024;
```

location

-	说明			
语法	**location** `[= \	~ \	~* \	^~] uri { ... } ; **location** @name { ... };
默认	——			
上下文	server、location			

根据请求 URI 设置配置。

在解码以 `%xx` 形式编码的文本，解析对相对路径组件 `.` 和 `..` 的引用并且将两个或更多个相邻斜线压缩成单斜线之后，对规范化的 URI 执行匹配。

location 可以由前缀字符串定义，也可以由正则表达式定义。正则表达式前面使用 `~*` 修正符（用于不区分大小写的匹配）或 `~` 修正符（用于区分大小写的匹配）指定。要查找匹配给定请求的 location，nginx 首先检查使用前缀字符串（前缀 location）定义的 location。期间，前缀最长的 location 被匹配选中并记住。然后按照它们在配置文件中出现的顺序检查正则表达式。正则表达式的搜索将在第一次匹配时终止，并使用相应的配置。如果找不到正则表达式的匹配，则使用先前记住的前缀 location 的配置。

location 块可以嵌套，除了下面提到的一些例外。

对于不区分大小写的操作系统，如 macOS 和 Cygwin，与前缀字符串匹配忽略大小写（0.7.7）。但是，比较仅限于一个字节的区域设置。

正则表达式可以包含捕获（0.7.40），之后可以在其他指令中使用。

如果最长匹配的前缀 location 具有 `^~` 修正符，则不检查正则表达式。

另外，使用 `=` 修正符可以使 URI 和 location 的匹配精确。如果找到完全匹配，则搜索结束。例如，如果 `/` 请求频繁，那么定义 `location=/` 会加快这些请求的处理速度，因为搜索在第一次比较之后会立即终止。这样的 location 不能包含嵌套 location。

> 从 0.7.1 到 0.8.41 版本，如果请求与没有 `=` 和 `^~` 修正符的前缀 location 匹配，则搜索也会终止，并且不再检查正则表达式。

下面举一个例子来说明：

```
location = / {
    [ configuration A ]
}

location / {
    [ configuration B ]
}

location /documents/ {
    [ configuration C ]
}

location ^~ /images/ {
    [ configuration D ]
}

location ~* \.(gif|jpg|jpeg)$ {
    [ configuration E ]
}
```

`/` 请求将与配置 A 匹配，`/index.html` 请求将匹配配置 B，`/documents/document.html` 请求将匹配配置 C，`/images/1.gif` 请求将匹配配置 D，`/documents/1.jpg` 请求将匹配配置 E。

`@` 前缀定义了一个命名 location。这样的 location 不用于常规的请求处理，而是用于请求重定向。它们不能嵌套，也不能包含嵌套的 location。

如果某个 location 由以斜杠字符结尾的前缀字符串定义，并且请求由 proxy_pass、fastcgi_pass、uwsgi_pass、scgi_pass 或 memcached_pass 中的一个进行处理，则会执行特殊处理。为了响应 URI 为该字符串的请求，但没有以斜杠结尾，带有 301 代码的永久重定向将返回所请求的 URI，并附上斜线。如果不需要，可以像以下这样定义 URI 和 location 的精确匹配：

```
location /user/ {
    proxy_pass http://user.example.com;
}

location = /user {
    proxy_pass http://login.example.com;
}
```

log_not_found

-	说明	
语法	**log_not_found** `on \	off`;
默认	log_not_found on;	
上下文	http、server、location	

启用或禁用将文件未找到错误记录到 error_log 中。

log_subrequest

-	说明	
语法	**log_subrequest** `on \	off`;
默认	log_subrequest off;	
上下文	http、server、location	

启用或禁用将子请求记录到 access_log 中。

max_ranges

-	说明
语法	**merge_slashes** `number` ;
默认	——
上下文	http、server、location
提示	该指令在 1.1.2 版本中出现

限制 byte-range 请求中允许的最大范围数。如果没有指定字节范围，则处理超出限制的请求。 默认情况下，范围的数量不受限制。零值将完全禁用 byte-range 支持。

merge_slashes

-	说明	
语法	**merge_slashes** `on	off`;
默认	merge_slashes on;	
上下文	http、server	

启用或禁用将 URI 中两个或多个相邻斜线压缩为单斜线。

请注意，压缩对于正确匹配前缀字符串和正则表达式的 location 非常重要。没有它，将不匹配 `//scripts/one.php` 请求：

```
location /scripts/ {
    ...
}
```

并可能被作为一个静态文件处理。因此它需要被转换成 `/scripts/one.php` 。

因为 base64 在内部使用 `/` 字符，所以如果 URI 包含 base64 编码的名称，则可能需要关闭压缩。但是，出于安全考虑，最好不要关闭压缩。

如果该指令是在 server 级别指定的，则仅在 server 是默认 server 时才使用该指令。指定的值也适用于监听相同地址和端口的所有虚拟服务器。

msie_padding

-	说明	
语法	**msie_padding** `on	off`;
默认	msie_padding on;	
上下文	http、server、location	

启用或禁用向状态超过 400 的 MSIE 客户端响应添加注释以将响应大小增加到 512 字节。

msie_refresh

-	说明	
语法	**msie_refresh** `on	off`;
默认	msie_refresh off;	
上下文	http、server、location	

启用或禁用发布刷新替代 MSIE 客户端重定向。

open_file_cache

-	说明
语法	**open_file_cache** `off` ; **open_file_cache** `max=N [inactive=time]` ;
默认	open_file_cache off;
上下文	http、server、location

配置一个缓存，可以存储：

- 打开文件描述符、大小和修改时间
- 有关目录是否存在信息
- 文件查找错误，如找不到文件、没有读取权限等

> 应该通过 open_file_cache_errors 指令分别启用缓存错误。

该指令有以下参数：

- `max`

  设置缓存中元素的最大数量。在缓存溢出时，最近最少使用的（LRU）元素将被移除

- `inactive`

  定义一个时间，在这个时间之后，元素在缓存中将被删除，默认是 60 秒

- `off`

  禁用缓存

示例：

```
open_file_cache          max=1000 inactive=20s;
open_file_cache_valid    30s;
open_file_cache_min_uses 2;
open_file_cache_errors   on;
```

open_file_cache_errors

-	说明	
语法	**open_file_cache_errors** `off` \	`on` ;
默认	open_file_cache_errors off;	
上下文	http、server、location	

通过 open_file_cache 启用或禁用文件查找错误缓存。

open_file_cache_min_uses

-	说明
语法	**open_file_cache_min_uses** `number` ;
默认	open_file_cache_min_uses 1;
上下文	http、server、location

设置由 open_file_cache 指令的 `inactive` 参数配置的时间段内文件访问的最小 `number`（次数），这是文件描述符在缓存中保持打开状态所必需的。

open_file_cache_valid

-	说明
语法	**open_file_cache_valid** `time` ;
默认	open_file_cache_valid 60s;
上下文	http、server、location

设置 open_file_cache 元素应该验证的时间。

output_buffers

-	说明
语法	**output_buffers** `number size` ;
默认	output_buffers 2 32k;
上下文	http、server、location

设置从磁盘读取响应缓冲区的 `number` （数量）和 `size` （大小）。

> 在 1.9.5 版本之前，默认值是 `1 32k` 。

port_in_redirect

-	说明	
语法	**port_in_redirect** `on` |	`off` ;
默认	port_in_redirect on;	
上下文	http、server、location	

启用或禁用指定由 nginx 发出的绝对重定向中的端口。

> 在重定向中使用的主服务器名称由 server_name_in_redirect 指令控制。

postpone_output

-	说明
语法	**postpone_output** `size` ;
默认	postpone_output 1460;
上下文	http、server、location

如果指定的话，客户端数据的传输将被推迟，直到 nginx 至少有 `size` 字节的数据要发送。零值将禁止延迟数据传输。

read_ahead

-	说明
语法	**read_ahead** `sizef` ;
默认	read_ahead 0;
上下文	http、server、location

设置使用文件时内核的预读数量。

在 Linux 上，使用 `posix_fadvise`（`0, 0, 0, POSIX_FADV_SEQUENTIAL`）系统调用，因此 `size` 参数将被忽略。

在 FreeBSD 上，使用自 FreeBSD 9.0-CURRENT 后支持的 `fcntl`（`O_READAHEAD, size`）系统调用。FreeBSD 7 需要打补丁。

recursive_error_pages

-	说明	
语法	**recursive_error_pages** `on` \	`off`;
默认	reset_timedout_connection off;	
上下文	http、server、location	

启用或禁用使用 error_page 指令执行多个重定向。这种重定向的数量是有限的。

request_pool_size

-	说明
语法	**request_pool_size** `size`;
默认	request_pool_size 4k;
上下文	http、server

允许精确调整每个请求的内存分配。该指令对性能影响最小，一般不应该使用。

reset_timedout_connection

-	说明	
语法	**reset_timedout_connection** `on` \	`off`;
默认	reset_timedout_connection off;	
上下文	http、server、location	

启用或禁用重置超时连接。重置过程如下。在关闭一个套接字之前，`SO_LINGER` 选项超时值被设置为 0。当套接字关闭时，TCP RST 被发送到客户端，并且释放该套接字占用的所有内存。这有助于避免长时间持有一个缓冲区已经被填充 FIN_WAIT1 状态的已关闭套接字。

应该注意，超时的 keep-alive 连接被正常关闭。

resolver

-	说明	
语法	**resolver** `address ... [valid=time] [ipv6=on	off]`;
默认	——	
上下文	http、server、location	

将用于解析 upstream 服务器名称的名称服务器配置进指定的地址，例如：

```
resolver 127.0.0.1 [:: 1]:5353;
```

地址可以指定为域名或 IP 地址，也可以指定一个可选的端口（1.3.1，1.2.2）。如果没有指定端口，则使用端口 53。名称服务器以轮询方式查询。

> 在 1.1.7 版本之前，只能配置一个名称服务器。从 1.3.1 和 1.2.2 版本开始，支持使用 IPv6 地址来指定名称服务器。

默认情况下，nginx 将在解析时查找 IPv4 和 IPv6 地址。如果不想查找 IPv6 地址，则可以指定 `ipv6=off` 参数。

> 从 1.5.8 版本开始，支持将名称解析为 IPv6 地址。

默认情况下，nginx 使用响应的 TTL 值缓存回复。可选的 `valid` 参数可以覆盖它：

```
resolver 127.0.0.1 [:: 1]:5353 valid=30s;
```

> 在 1.1.9 版本之前，缓存时间是不能调整的，nginx 总是缓存 5 分钟的回复。
>
> 为了防止 DNS 欺骗，建议在安全的可信任本地网络中配置 DNS 服务器。

satisfy

-	说明	
语法	**satisfy** `all` \	`any` ;
默认	satisfy all;	
上下文	http、server、location	

如果所有（全部）或至少一个（任意一个）ngx_http_access_module、ngx_http_auth_basic_module、ngx_http_auth_request_module 或 ngx_http_auth_jwt_module 模块允许访问，则允许访问。

示例：

```
location / {
    satisfy any;

    allow 192.168.1.0/32;
    deny  all;

    auth_basic           "closed site";
    auth_basic_user_file conf/htpasswd;
}
```

send_lowat

-	说明
语法	**send_lowat** `size` ;
默认	send_lowat 0;
上下文	http、server、location

如果指令设置为非零值，nginx 将尝试通过使用 kqueue 方法的 `NOTE_LOWAT` 标志或 `SO_SNDLOWAT` 套接字选项来最小化客户端套接字上的发送操作数。在这两种情况下都使用到了指定的 `size` 。

该指令在 Linux、Solaris 和 Windows 上被忽略。

send_timeout

-	说明
语法	**send_timeout** `time` ;
默认	send_timeout 60s;
上下文	http、server、location

设置向客户端发送响应的超时时间。超时设置只限定在两次连续的写入操作之间，而不是用于整个响应的传输。如果客户端在此时间内没有收到任何内容，则连接将被关闭。

sendfile

-	说明	
语法	**sendfile** `on` \	`off` ;
默认	sendfile off;	
上下文	http、server、location、location 中的 if	

启用或禁用 `sendfile()`。

从 nginx 0.8.12 和 FreeBSD 5.2.1 开始，可以用 aio 来为 `sendfile()` 预加载数据：

```
location /video/ {
    sendfile       on;
    tcp_nopush     on;
    aio            on;
}
```

在此配置中，使用 `SF_NODISKIO` 标志调用 `sendfile()`，使其不会阻塞磁盘 I/O，而是以报告数据不在内存中的方式代替。然后 nginx 通过读取一个字节来启动异步数据加载。在第一次读取时，FreeBSD 内核会将文件的第一个 128K 字节加载到内存中，但是下一次读取只能以 16K 块加载数据。可以使用 read_ahead 指令进行修改。

> 在 1.7.11 版本之前，可以使用 `aio sendfile` 来启用预加载。

sendfile_max_chunk

-	说明
语法	**sendfile_max_chunk** `size`;
默认	sendfile_max_chunk 0;
上下文	http、server、location

设置为非零值时，可限制单个 `sendfile()` 调用时传输的数据量。如果没有限制，一个快速连接可能会完全占用工作进程。

server

-	说明
语法	**server** `{ ... }`;
默认	——
上下文	http

设置虚拟服务器的配置。基于 IP（基于 IP 地址）和基于名称（基于 `Host` 请求头字段）的虚拟服务器之间没有明确的界限。相反，listen 指令描述应接受服务器连接的所有地址和端口，server_name 指令列出所有服务器名称。如何处理请求文档中提供了配置示例。

server_name

-	说明
语法	**server_name** `name ...` ;
默认	server_name "";
上下文	server

设置虚拟服务器名称，例如：

```
server {
    server_name example.com www.example.com;
}
```

第一个名字将成为主服务器名称。

服务器名称可以包含一个星号（ `*` ）以代替名称的第一部分或最后一部分：

```
server {
    server_name example.com *.example.com www.example.*;
}
```

这样的名称被称为通配符名称。

上面提到的前两个名字可以合并成一个：

```
server {
    server_name .example.com;
}
```

也可以在服务器名称中使用正则表达式，在名称前面加上波浪号（ `~` ）：

```
server {
    server_name www.example.com ~^www\d+\.example\.com$;
}
```

正则表达式可以包含之后用于其他指令的捕获（0.7.40）：

```
server {
    server_name ~^(www\.)?(.+)$;

    location / {
        root /sites/$2;
    }
}

server {
    server_name _;

    location / {
        root /sites/default;
    }
}
```

在正则表达式中命名捕获创建变量（0.8.25），之后可在其他指令中使用：

```
server {
    server_name ~^(www\.)?(?<domain>.+)$;

    location / {
        root /sites/$domain;
    }
}

server {
    server_name _;

    location / {
        root /sites/default;
    }
}
```

如果指令参数设置为 `$hostname`（0.9.4），则将替换为机器的主机名。

也可以指定一个空的服务器名称（0.7.11）：

```
server {
    server_name www.example.com "";
}
```

它允许这个服务器对给定的 address:port 对处理没有 `Host` 头的请求，而不是默认的服务器。这是默认设置。

> 在 0.8.48 之前，默认使用机器的主机名。

在按名称搜索虚拟服务器的过程中，如果名称与多个指定变体相匹配（例如，通配符名称和正则表达式匹配），将按照以下优先顺序选择第一个匹配变体：

1. 确切的名字
2. 以星号开头的最长通配符名称，例如 `*.example.com`
3. 以星号结尾的最长通配符名称，例如 `mail.*`
4. 第一个匹配的正则表达式（按照配置文件中的出现顺序）

服务器名称的详细描述在单独的服务器名称文档中提供。

server_name_in_redirect

-	说明	
语法	**server_name** `on`	`off` ;
默认	server_name_in_redirect off;	
上下文	http、server、location	

启用或禁用在由 nginx 发出的 absolute 指令中使用由 server_name 指令指定的主服务器名称。当禁用主服务器名称时，将使用 `Host` 请求头字段中的名称。如果此字段不存在，则使用服务器的 IP 地址。

重定向中使用端口由 port_in_redirect 指令控制。

server_names_hash_bucket_size

-	说明		
语法	**server_names_hash_bucket_size** `size` ;		
默认	server_names_hash_bucket_size 32	64	128;
上下文	http		

设置服务器名称哈希表的存储桶大小。默认值取决于处理器缓存行的大小。设置哈希表的细节在单独的文档中提供。

server_names_hash_max_size

-	说明
语法	**server_names_hash_max_size** `size` ;
默认	server_names_hash_max_size 512;
上下文	http

设置服务器名称哈希表的最大大小。设置哈希表的细节在单独的文档)中提供。

server_tokens

-	说明			
语法	**server_tokens** `on` |	`off` |	`build` |	`string` ;
默认	server_tokens on;			
上下文	http、server、location			

在错误页面和 `Server` 响应头字段中启用或禁用发送 nginx 版本。

`build` 参数（1.11.10）能够发送构建名称以及 nginx 版本。

此外，作为我们的商业订阅的一部分，从1.9.13 版本开始，可以使用带有变量的字符串显式设置错误页面上的签名和 `Server` 响应头字段值。指定空字符串将禁用 `Server` 字段的发送。

tcp_nodelay

-	说明	
语法	**tcp_nodelay** `on` |	`off` ;
默认	tcp_nodelay on;	
上下文	http、server、location	

启用或禁用 `TCP_NODELAY` 选项的使用。该选项仅在连接转换到 keep-alive 状态时启用。

tcp_nopush

-	说明	
语法	**tcp_nopush** `on` |	`off` ;
默认	tcp_nopush off;	
上下文	http、server、location	

启用或禁用在 FreeBSD 上使用 `TCP_NOPUSH` 套接字选项或在 Linux 上使用 `TCP_CORK` 套接字选项。这些选项只有在使用 sendfile 时才能使用。启用该选项将允许：

- 在 Linux 和 FreeBSD 4 上，在同一包中可发送响应头和文件开头。
- 以完整包的方式发送一个文件

try_files

-	说明
语法	**try_files** `file ... uri` ; **try_files** `file ... =code` ;
默认	——
上下文	server、location

以指定顺序检查文件是否存在，并使用第一个找到的文件进行请求处理。处理将在当前上下文中执行。指向文件的路径根据 root 和 alias 指令从 `file` 参数构造。可以通过在名称末尾指定斜线来检查目录是否存在，例如，`$URI/` 。如果找不到任何文件，则内部重定向将指向最后一个参数中指定的 `uri` 。例如：

```
location /images/ {
    try_files $uri /images/default.gif;
}

location = /images/default.gif {
    expires 30s;
}
```

最后一个参数也可以指向一个命名的 location ，如以下示例。从 0.7.51 版本开始，最后一个参数也可以是一个 `code` ：

```
location / {
    try_files $uri $uri/index.html $uri.html =404;
}
```

代理 Mongrel 示例：

```
location / {
    try_files /system/maintenance.html
              $uri $uri/index.html $uri.html
              @mongrel;
}

location @mongrel {
    proxy_pass http://mongrel;
}
```

Drupal/FastCGI 示例：

```
location / {
    try_files $uri $uri/ @drupal;
}

location ~ \.php$ {
    try_files $uri @drupal;

    fastcgi_pass ...;

    fastcgi_param SCRIPT_FILENAME /path/to$fastcgi_script_name;
    fastcgi_param SCRIPT_NAME     $fastcgi_script_name;
    fastcgi_param QUERY_STRING    $args;

    ... other fastcgi_param's
}

location @drupal {
    fastcgi_pass ...;

    fastcgi_param SCRIPT_FILENAME /path/to/index.php;
    fastcgi_param SCRIPT_NAME     /index.php;
    fastcgi_param QUERY_STRING    q=$uri&$args;

    ... other fastcgi_param's
}
```

在以下示例中

```
location / {
    try_files $uri $uri/ @drupal;
}
```

try_files 指令相当于

```
location / {
    error_page 404 = @drupal;
    log_not_found off;
}
```

还有一个示例

```
location ~ \.php$ {
    try_files $uri @drupal;

    fastcgi_pass ...;

    fastcgi_param SCRIPT_FILENAME /path/to$fastcgi_script_name;

    ...
}
```

在将请求传递给 FastCGI 服务器之前，try_files 将检查 PHP 文件是否存在。

Wordpress 与 Joomla 示例：

```
location / {
    try_files $uri $uri/ @wordpress;
}

location ~ \.php$ {
    try_files $uri @wordpress;

    fastcgi_pass ...;

    fastcgi_param SCRIPT_FILENAME /path/to$fastcgi_script_name;
    ... other fastcgi_param's
}

location @wordpress {
    fastcgi_pass ...;

    fastcgi_param SCRIPT_FILENAME /path/to/index.php;
    ... other fastcgi_param's
}
```

types_hash_bucket_size

-	说明
语法	**types_hash_bucket_size** `size` ;
默认	types_hash_bucket_size 64;
上下文	http、server、location

设置类型哈希表桶大小。设置哈希表的细节在单独的文档中有提供。

> 在版 1.5.13 本之前，默认值取决于处理器缓存行的大小。

types_hash_max_size

-	说明
语法	**types_hash_max_size** `size` ;
默认	types_hash_max_size 1024;
上下文	http、server、location

设置类型哈希表的最大大小。设置哈希表的细节在单独的文档中有提供。

underscores_in_headers

-	说明	
语法	**underscores_in_headers** `on` |	`off` ;
默认	underscores_in_headers off;	
上下文	http、server	

启用或禁用在客户端请求头字段中使用下划线。当禁用下划线时，名称中包含下划线的请求头字段将被标记为无效，并受制于 ignore_invalid_headers 指令。

如果该指令是在 server 级别指定，则仅在 server 是默认 server 时才使用该指令。指定的值也适用于监听相同地址和端口的所有虚拟服务器。

variables_hash_bucket_size

-	说明
语法	**variables_hash_bucket_size** `size` ;
默认	variables_hash_bucket_size size;
上下文	http

设置变量哈希表桶大小。设置哈希表的细节在单独的文档中有提供。

variables_hash_max_size

-	说明
语法	**variables_hash_max_size** `size` ;
默认	variables_hash_max_size 1024;
上下文	http

设置变量哈哈希表的最大大小。设置哈希表的细节在单独的文档中有提供。

> 在 1.5.13 版本之前，默认值是 512。

内嵌变量

`ngx_http_core_module` 模块支持嵌入式变量名称与 Apache 服务器变量匹配。首先，这些是出现在客户端请求头字段的变量，例如 `$http_user_agent` 、 `$http_cookie` 等等。还有其他变量：

- `$arg_name`

  请求行中的 `name` 参数

- `$args`

  请求行中的参数

- `$binary_remote_addr`

  客户端地址以二进制形式表示，值的长度对于 IPv4 地址总是 4 个字节，对于 IPv6 地址总是 16 个字节

- `$body_bytes_sent`

  发送到客户端的字节数，不包括响应头。此变量与 `mod_log_config` Apache 模块的 `%B` 参数兼容

- `$bytes_sent`

  发送到客户端的字节数（1.3.8、1.2.5）

- $connection

  连接序列号（1.3.8、1.2.5）

- `$connection_requests`

  当前通过连接请求的请求数（1.3.8、1.2.5）

- `$connect_length`

  `Content-Length` 请求头字段

- `$content_type`

  `Content-Type` 请求头字段

- `$cookie_name`

  名称为 `name` 的 cookie

- `$document_root`

  根目录或别名指令的当前请求的值

- `$document_uri`

  与 `$uri` 相同

- `$host`

  按照以下优先顺序：来自请求行的主机名，来自 `Host` 请求头字段的主机名，或与请求匹配的服务器名

- `$hostname`

  主机名

- `$http_name`

  任意请求头字段,变量名称的最后一部分是将字段名称转换为小写，并用破折号替换为下划线

- `$https`

  如果连接以 SSL 模式运行，则为 `on`，否则为空字符串

- `$is_args`

  如果请求行有参数则为 `?`，否则为空字符串

- `$limit_rate`

  设置这个变量可以实现响应速率限制，见 limit_rate

- `$msec`

  当前时间以毫秒为单位（1.3.9、1.2.6）

- `$nginx_version`

  nginx 版本

- `$pid`

  工作进程的 PID

- `$pipe`

  如果请求是管道模式则为 `p`，否则为 `.`（1.3.12、1.2.7）

- `$proxy_protocol_addr`

  来自 PROXY 协议头的客户端地址，否则为空字符串（1.5.12）

  要在 listen 指令中设置 `proxy_protocol` 参数，必须先启用 PROXY 协议。

- `$proxy_protocol_port`

  PROXY 协议头中的客户端口，否则为空字符串（1.11.0）

  要在 listen 指令中设置 `proxy_protocol` 参数，必须先启用 PROXY 协议。

- `$query_string`

  与 `$args` 相同

- `$realpath_root`

  与当前请求的 root 或 alias 指令值相对应的绝对路径名，所有符号链接都将解析为实际路径

- `$remote_addr`

  客户端地址

- `$remote_port`

  客户端端口

- `$remote_user`

  基本身份验证提供的用户名

- `$request`

  完整的原始请求行

- `$request_body`

  请求正文

  当请求正文被读取到内存缓冲区时，变量的值在由proxy_pass、fastcgi_pass、uwsgi_pass 和 scgi_pass 指令处理的 location 中可用。

- `$request_body_file`

  带有请求正文的临时文件的名称

  在处理结束时，文件需要被删除。想始终将请求主体写入文件中，需要启用 client_body_in_file_only。当临时文件的名称在代理请求中或在向 FastCGI/uwsgi/SCGI 服务器的请求中传递时，应该分别通过 proxy_pass_request_body off、fastcgi_pass_request_body off、uwsgi_pass_request_body off 或 scgi_pass_request_body off 指令禁用传递请求正文。

- `$request_completion`

  如果请求已经完成，则返回 `OK`，否则返回空字符串

- `$request_filename`

  当前请求的文件路径，基于 root 或 alias 指令以及请求 URI

- `$request_id`

  由 16 个随机字节生成的唯一请求标识符，以十六进制表示（1.11.0）

- `$request_length`

  请求长度（包括请求行、头部和请求体）（1.3.12、1.2.7）

- `$request_method`

  请求方法，通常为 `GET` 或 `POST`

- `$request_time`

  请求处理时间以毫秒为单位（1.3.9、1.2.6）。自客户端读取第一个字节的时间起

- `$request_uri`

  完整的原始请求 URI（带参数）

- `$scheme`

  请求模式，`http` 或 `https`

- `$sent_http_name`

  任意响应头字段。变量名称的最后一部分是将字段名称转换为小写，并用破折号替换为下划线

- `$sent_trailer_name`

  响应结束时发送的任意字段（1.13.2）。变量名称的最后一部分是将字段名称转换为小写，并用破折号替换为下划线

- `$server_addr`

  接受请求的服务器地址

  通常需要一个系统调用来计算这个变量的值。为了避免系统调用，listen 指令必须指定地址并使用 `bind` 参数。

- `$server_name`

  接受请求的服务器名称

- `$server_port`

  接受请求的服务器端口

- `$server_protocol`

  请求协议，通常为 `HTTP/1.0` 、 `HTTP/1.1` 或 HTTP/2.0

- `$status`

  响应状态（1.3.2、1.2.2）

- `$tcpinfo_rtt` 、 `$tcpinfo_rttvar` 、 `$tcpinfo_snd_cwnd` 、 `$tcpinfo_rcv_space`

  有关客户端 TCP 连接的信息。在支持 `TCP_INFO` 套接字选项的系统上可用

- `$time_iso8601`

  本地时间采用 ISO 8601 标准格式（1.3.12、1.2.7）

- `$time_local`

  通用日志格式（Common Log Format）中的本地时间（1.3.12、1.2.7）

- `$uri`

  规范化过的当前请求 URI

  `$uri` 的值可能在请求期间会改变

原文档

http://nginx.org/en/docs/http/ngx_http_core_module.html

ngx_http_access_module

`ngx_http_access_module` 模块允许限制对某些客户端地址的访问。

访问也可以通过密码、子请求结果或 JWT 限制。可用 satisfy 指令通过地址和密码同时限制访问。

示例配置

```
location / {
    deny  192.168.1.1;
    allow 192.168.1.0/24;
    allow 10.1.1.0/16;
    allow 2001:0db8::/32;
    deny  all;
}
```

按顺序检查规则，直到找到第一个匹配项。在本例中，仅允许 IPv4 网络 `10.1.1.0/16` 和 `192.168.1.0/24` 与 IPv6 网络 `2001:0db8::/ 32` 访问，不包括地址 `192.168.1.1` 。在很多规则的情况下，最好使用 ngx_http_geo_module 模块变量。

指令

allow

-	说明
语法	**allow** `address` | `CIDR` | `unix:` | `all` ;
默认	——
上下文	http、server、location、limit_except

允许访问指定的网络或地址。如果指定了特殊值 `unix:` （1.5.1），则允许访问所有 UNIX 域套接字。

deny

-	说明
语法	**deny** `address` | `CIDR` | `unix:` | `all` ;
默认	——
上下文	http、server、location、limit_except

拒绝指定网络或地址的访问。如果指定了特殊值 `unix:`（1.5.1），则拒绝所有 UNIX 域套接字的访问。

原文档

http://nginx.org/en/docs/http/ngx_http_access_module.html

ngx_http_addition_module

`ngx_http_addition_module` 是一个过滤器，用于在响应之前和之后添加文本。该模块不是默认构建，要启用应使用 `--with-http_addition_module` 配置参数构建。

示例配置

```
location / {
    add_before_body /before_action;
    add_after_body  /after_action;
}
```

指令

add_before_body

-	说明
语法	**add_before_body** `uri` ;
默认	——
上下文	http、server、location

在响应正文之前添加文本，作为给定子请求的一个处理结果返回。空字符串（`""`）作为参数时将取消从先前配置级别继承的额外文本。

add_after_body

-	说明
语法	**add_after_body** `uri` ;
默认	——
上下文	http、server、location

在响应正文之后添加文本，作为给定子请求的一个处理结果返回。空字符串（ `""` ）作为参数时将取消从先前配置级别继承的额外文本。

addition_types

-	说明
语法	**addition_types** `mime-type ...` ;
默认	addition_types text/html;
上下文	http、server、location
提示	该指令在 0.7.9 版本中出现

除了 `text/html` 外，允许您在指定的 MIME 类型的响应中添加文本。特殊值 `*` 匹配所有MIME类型（0.8.29）。

原文档

http://nginx.org/en/docs/http/ngx_http_addition_module.html

ngx_http_auth_basic_module

`ngx_http_auth_basic_module` 模块允许通过使用 **HTTP Basic Authentication** 协议验证用户名和密码来限制对资源的访问。

也可以通过地址、子请求结果或 JWT 来限制访问。可使用 satisfy 指令通过地址和密码同时限制访问。

示例配置

```
location / {
    auth_basic           "closed site";
    auth_basic_user_file conf/htpasswd;
}
```

指令

auth_basic

-	说明
语法	**auth_basic** `string` | `off` ;
默认	auth_basic off;
上下文	http、server、location、limit_except

使用 **HTTP Basic Authentication** 协议，启用用户名和密码验证。指定的参数用作为一个 `realm` 。参数值可以包含变量（1.3.10、1.2.7）。特殊值 `off` 可以取消从先前配置级别继承的 `auth_basic` 指令的影响。

auth_basic_user_file

-	说明
语法	**auth_basic_user_file** `file`;
默认	——
上下文	http、server、location、limit_except

指定一个用于保存用户名和密码的文件，格式如下：

```
# comment
name1:password1
name2:password2:comment
name3:password3
```

`file` 的名称可以包含变量。

支持以下密码类型：

- 用 `crypt()` 函数加密；可以使用 Apache HTTP Server 分发的或 `openssl passwd` 命令中的 `htpasswd` 工具生成；
- 使用基于 MD5 密码算法（apr1）的 Apache 变体进行散列计算；可以用与上述相同的工具生成；
- RFC 2307 中指定的 `{scheme}data` 语法（1.0.3+）；目前的实现方案包括 `PLAIN`（一个不应该使用的示例）、`SHA`（1.3.13）（简单 SHA-1 散列，不应该使用）和 `SSHA`（一些软件包使用了加盐的 SHA-1 散列，特别是 OpenLDAP 和 Dovecot）。

> 增加了对 SHA 模式的支持，仅用于帮助从其他 Web 服务器迁移。它不应该用于新密码，因为它使用的未加盐的 SHA-1 散列容易受到彩虹表的攻击。

原文档

http://nginx.org/en/docs/http/ngx_http_auth_basic_module.html

ngx_http_auth_jwt_module

`ngx_http_auth_jwt_module` 模块（1.11.3）通过验证使用指定的密钥提供的 JSON Web Token（JWT）来实现客户端授权。JWT claims 必须以 JSON Web Signature（JWS）结构编码。该模块可用于 OpenID Connect 身份验证。

该模块可以通过 satisfy 指令与其他访问模块（如 ngx_http_access_module、ngx_http_auth_basic_module 和 ngx_http_auth_request_module）进行组合。

> 此模块可作为我们商业订阅的一部分。

示例配置

```
location / {
    auth_jwt          "closed site";
    auth_jwt_key_file conf/keys.json;
}
```

指令

auth_jwt

-	说明
语法	**auth_jwt** `string [token=$variable]` | `off` ;
默认	auth_jwt off;
上下文	http、server、location

启用 JSON Web Token 验证。指定的字符串作为一个 `realm` 。参数值可以包含变量。

可选的 `token` 参数指定一个包含 JSON Web Token 的变量。默认情况下，JWT 作 Bearer Token 在 **Authorization** 头中传递。JWT 也可以作为 cookie 或查询字符串的一部分传递：

```
auth_jwt "closed site" token=$cookie_auth_token;
```

特殊值 `off` 取消从上一配置级别继承的 `auth_jwt` 指令的作用。

auth_basic_user_file

-	说明
语法	**auth_jwt_header_set** `$variable name` ;
默认	——
上下文	http
提示	该指令在 1.11.10 版本中出现

将 `variable` 设置为给定的 JOSE 头参数 `name` 。

auth_jwt_claim_set

-	说明
语法	**auth_jwt_claim_set** `$variable name` ;
默认	——
上下文	http
提示	该指令在 1.11.10 版本中出现

将 `variable` 设置为给定的 JWT claim 参数 `name` 。

auth_jwt_key_file

-	说明
语法	**auth_jwt_key_file** `file` ;
默认	——
上下文	http、server、location

指定用于验证 JWT 签名的 JSON Web Key Set 格式的 `file` （文件）。参数值可以包含变量。

内嵌变量

`ngx_http_auth_jwt_module` 模块支持内嵌变量：

- `$jwt_header_name`

  返回 JOSE 头的值

- `$jwt_claim_name`

  返回 JWT claim 的值

原文档

http://nginx.org/en/docs/http/ngx_http_auth_jwt_module.html

ngx_http_auth_request_module

`ngx_http_auth_request_module` 模块（1.5.4+）基于子请求结果实现客户端授权。如果子请求返回一个 2xx 响应代码，则允许访问。如果返回 401 或 403，则拒绝访问并抛出相应的错误代码。子请求返回的任何其他响应代码被认为是一个错误。

对于 401 错误，客户端也从子请求响应中接收 **WWW-Authenticate** 头。

该模块不是默认构建，应该在构建时使用 `--with-http_auth_request_module` 配置参数启用。

该模块可以通过 satisfy 指令与其他访问模块（如 ngx_http_access_module、ngx_http_auth_basic_module 和 ngx_http_auth_jwt_module）进行组合。

> 在 1.7.3 版本之前，无法缓存对授权子请求的响应（使用 proxy_cache、proxy_store 等）。

示例配置

```
location /private/ {
    auth_request /auth;
    ...
}

location = /auth {
    proxy_pass ...
    proxy_pass_request_body off;
    proxy_set_header Content-Length "";
    proxy_set_header X-Original-URI $request_uri;
}
```

指令

auth_request

-	说明
语法	**auth_request** `uri` | `off` ;
默认	auth_request off;
上下文	http、server、location

启用基于子请求结果的授权，并设置发送子请求的 URI。

auth_request_set

-	说明
语法	**auth_request_set** `$variable value;` ;
默认	——
上下文	http

在授权请求完成后，将请求 `variable`（变量）设置为给定的 `value`（值）。该值可能包含授权请求中的变量，例如 `$upstream_http_*` 。

原文档

http://nginx.org/en/docs/http/ngx_http_auth_request_module.html

ngx_http_autoindex_module

`ngx_http_autoindex_module` 模块处理以斜线字符（`/`）结尾的请求并生成一个目录列表。当 ngx_http_index_module 模块找不到索引文件时，通常会将请求传递给 `ngx_http_autoindex_module` 模块。

示例配置

```
location / {
    autoindex on;
}
```

指令

autoindex

-	说明
语法	**autoindex** `on` | `off` ;
默认	autoindex off;
上下文	http、server、location

启用或禁用目录列表输出。

autoindex_exact_size

-	说明
语法	**autoindex_exact_size** `on` | `off` ;
默认	autoindex_exact_size on;
上下文	http、server、location

对于 HTML 格式，指定是否应在目录列表中输出确切的文件大小，或者四舍五入到千字节、兆字节和千兆字节。

autoindex_format

-	说明
语法	**autoindex_format** `html` | `xml` | `json` | `jsonp` ;
默认	autoindex_format html;
上下文	http、server、location
提示	该指令在 1.7.9 版本中出现

设置目录列表的格式。

当使用 JSONP 格式时，使用 `callback` 请求参数设置回调函数的名称。如果没有参数或为空值，则使用 JSON 格式。

XML 输出可以使用 ngx_http_xslt_module 模块进行转换。

autoindex_localtime

-	说明
语法	**autoindex_localtime** `on` | `off` ;
默认	autoindex_localtime off;
上下文	http、server、location

对于 HTML 格式，指定目录列表中的时间是否应使用本地时区或 UTC 输出。

原文档

http://nginx.org/en/docs/http/ngx_http_autoindex_module.html

ngx_http_browser_module

`ngx_http_browser_module` 模块创建值由 **User-Agent** 请求头域决定的变量：

- `$modern_browser`

  如果浏览器被识别为现代，则等于 modern_browser_value 指令设置的值

- `$ancient_browser`

  如果浏览器被识别为古代，则等于由 ancient_browser_value 指令设置的值

- `$MSIE`

  如果浏览器被识别为任何版本的 MSIE，则等于 `1`

示例配置

选择一个索引文件：

```
modern_browser_value "modern.";

modern_browser msie      5.5;
modern_browser gecko     1.0.0;
modern_browser opera     9.0;
modern_browser safari    413;
modern_browser konqueror 3.0;

index index.${modern_browser}html index.html;
```

旧浏览器重定向：

```
modern_browser msie      5.0;
modern_browser gecko     0.9.1;
modern_browser opera     8.0;
modern_browser safari    413;
modern_browser konqueror 3.0;

modern_browser unlisted;

ancient_browser Links Lynx netscape4;

if ($ancient_browser) {
    rewrite ^ /ancient.html;
}
```

指令

ancient_browser

-	说明
语法	**ancient_browser** `string ...`;
默认	——
上下文	http、server、location

如果在 **User-Agent** 请求头域中找到任何特殊的子字符串，浏览器将被视为传统类型。特殊字符串 `netscape4` 对应于正则表达式 `^Mozilla/[1-4]`。

ancient_browser_value

-	说明
语法	**ancient_browser_value** `string`;
默认	ancient_browser_value 1;
上下文	http、server、location

设置 `$ancient_browser` 变量的值。

modern_browser

-	说明
语法	**modern_browser** `browser version` ; **modern_browser** `unlisted` ;
默认	——
上下文	http、server、location

指定将浏览器视为现代版本开始的版本。浏览器可以是以下任何一种：`msie` 、 `gecko` （基于 Mozilla 的浏览器）、 `opera` 、 `safari` 或 `konqueror` 。

版本可以是以下列格式：X、X.X、X.X.X 或 X.X.X.X。每种格式的最大值分别为 4000、4000.99、4000.99.99 和 4000.99.99.99。

未列出的特殊值如果未被 `modern_browser` 和 ancient_browser 指令指定，则将其视为现代浏览器。否则被认为是传统浏览器。如果请求没有在头中提供 **User-Agent** 域，则浏览器被视为未列出。

modern_browser_value

-	说明
语法	**modern_browser_value** `string` ;
默认	modern_browser_value 1;
上下文	http、server、location

设置 `$modern_browser` 变量的值。

原文档

http://nginx.org/en/docs/http/ngx_http_browser_module.html

ngx_http_charset_module

`ngx_http_charset_module` 模块将指定的字符集添加到 **Content-Type** 响应头域。此外，该模块可以将数据从一个字符集转换为另一个字符集，但也存在一些限制：

- 转换工作只能是从服务器到客户端
- 只能转换单字节字符集
- 或转为/来自 UTF-8 的单字节字符集。

示例配置

```
include        conf/koi-win;

charset        windows-1251;
source_charset koi8-r;
```

指令

charset

-	说明
语法	**charset** `charset` | `off` ;
默认	charset off;
上下文	http、server、location、location 中的 if

将指定的字符集添加到 **Content-Type** 响应头域。如果此字符集与 source_charset 指令指定的字符集不同，则执行转换。

参数 `off` 取消将字符集添加到 **Content-Type** 响应头。

可以使用一个变量来定义字符集：

```
charset $charset;
```

在这种情况下，变量的值至少要在 charset_map、charset 或 source_charset 其中一个指令配置一次。对于 `utf-8`、`windows-1251` 和 `koi8-r` 字符集，将 `conf/koi-win`、`conf/koi-utf` 和 `conf/win-utf` 文件包含到配置中就足够了。对于其他字符集，只需制作一个虚构的转换表即可，例如：

```
charset_map iso-8859-5 _ { }
```

另外，可以在 **X-Accel-Charset** 响应头域中设置一个字符集。可以使用proxy_ignore_headers、fastcgi_ignore_headers、uwsgi_ignore_headers 和 scgi_ignore_headers 指令禁用此功能。

charset_map

-	说明
语法	**charset_map** `charset1 charset2 { ... }`;
默认	——
上下文	http

描述转换表，将一个字符集转换到另一个字符集。反向转换表也使用相同的数据构建。字符代码是十六进制格式。不在 80-FF 范围内的字符将被替换为 `?`。当从 UTF-8 转换时，一个字节字符集中丢失的字符将被替换为 `&#XXXX;`。

示例：

```
charset_map koi8-r windows-1251 {
    C0 FE ; # small yu
    C1 E0 ; # small a
    C2 E1 ; # small b
    C3 F6 ; # small ts
    ...
}
```

在将转换表描述为 UTF-8 时，应在第二列中给出 UTF-8 字符集代码，例如：

```
charset_map koi8-r utf-8 {
    C0 D18E ; # small yu
    C1 D0B0 ; # small a
    C2 D0B1 ; # small b
    C3 D186 ; # small ts
    ...
}
```

在分发文件 `conf/koi-win` 、 `conf/koi-utf` 和 `conf/win-utf` 中提供了从 `koi8-r` 到 `windows-1251` 以及从 `koi8-r` 和 `windows-1251` 到 `utf-8` 的完整转换表。

charset_types

-	说明
语法	**charset_types** `mime-type ...` ;
默认	charset_types text/html text/xml text/plain text/vnd.wap.wml

application/javascript application/rss+xml;| |上下文|http、server、location| |提示|该指令在 0.7.9 版本中出现|

除了 `text/html` 之外，还可以使用指定了 MIME 类型的响应中的模块处理。特殊值 `*` 可匹配任何 MIME 类型（0.8.29）。

> 直到 1.5.4 版本， `application/x-javascript` 被作为默认的 MIME 类型，而不是 `application/javascript` 。

override_charset

-	说明
语法	**override_charset** `on` | `off` ;
默认	override_charset off;
上下文	http、server、location、location 中的 if

当应答已经在 **Content-Type** 响应头域中携带字符集时，确定是否应该对从代理或 FastCGI/uwsgi/SCGI 服务器接收的应答执行转换。如果启用转换，则在接收到的响应中指定的字符集将用作源字符集。

> 应该注意的是，如果在子请求中接收到响应，则始终执行从响应字符集到主请求字符集的转换，而不管 `override_charset` 指令如何设置。

source_charset

-	说明
语法	**source_charset** `charset` ;
默认	——
上下文	http、server、location、location 中的 if

定义响应的源字符集。如果此字符集与 charset 指令中指定的字符集不同，则执行转换。

原文档

http://nginx.org/en/docs/http/ngx_http_charset_module.html

ngx_http_dav_module

`ngx_http_dav_module` 模块用于通过 WebDAV 协议进行文件管理自动化。该模块处理 HTTP 和 WebDAV 的 PUT、DELETE、MKCOL、COPY 和 MOVE 方法。

该模块不是默认构的，您可以在构建时使用 `--with-http_dav_module` 配置参数启用。

> 需要其他 WebDAV 方法进行操作的 WebDAV 客户端将无法使用此模块。

示例配置

```
location / {
    root                  /data/www;

    client_body_temp_path /data/client_temp;

    dav_methods PUT DELETE MKCOL COPY MOVE;

    create_full_put_path  on;
    dav_access            group:rw  all:r;

    limit_except GET {
        allow 192.168.1.0/32;
        deny  all;
    }
}
```

指令

dav_access

-	说明
语法	**dav_access** `users:permissions ...` ;
默认	dav_access user:rw;
上下文	http、server、location

设置新创建的文件和目录的访问权限，例如：

```
dav_access user:rw group:rw all:r;
```

如果指定了任何 `group` （组）或所有访问权限，则可以省略 `user` 权限：

```
dav_access group:rw all:r;
```

dav_methods

-	说明	
语法	**dav_methods** `off` \	`method ...` ;
默认	dav_methods off;	
上下文	http、server、location	

允许指定的 HTTP 方法和 WebDAV 方法。参数 `off` 将拒绝本模块处理的所有方法。支持以下方法：PUT、DELETE、MKCOL、COPY 和 MOVE。

使用 PUT 方法上传的文件首先需要写入一个临时文件，然后重命名该文件。从 0.8.9 版本开始，临时文件和持久存储可以放在不同的文件系统上。但是，请注意，在这种情况下，文件复制需要跨越两个文件系统，而不是简单的重命名操作。因此，建议通过 client_body_temp_path 指令对临时文件设置存放目录，与保存文件的目录设置在同一文件系统上。

当使用 PUT 方法创建文件时，可以通过在 **Date** 头域中传递日期来指定修改日期。

create_full_put_path

-	说明	
语法	**create_full_put_path** `on` \	`off` ;
默认	create_full_put_path off;	
上下文	http、server、location	

WebDAV 规范仅允许在已存在的目录中创建文件。开启该指令允许创建所有需要的中间目录。

min_delete_depth

-	说明
语法	**min_delete_depth** `number` ;
默认	min_delete_depth 0;
上下文	http、server、location

允许 DELETE 方法删除文件，只要请求路径中的元素数不少于指定的数字。例如，指令：

```
min_delete_depth 4;
```

允许删除请求中的文件

```
/users/00/00/name
/users/00/00/name/pic.jpg
/users/00/00/page.html
```

拒绝删除的文件

```
/users/00/00
```

原文档

http://nginx.org/en/docs/http/ngx_http_dav_module.html

ngx_http_empty_gif_module

`ngx_http_empty_gif_module` 模块发送单像素透明 GIF。

示例配置

```
location = /_.gif {
    empty_gif;
}
```

指令

empty_gif

-	说明
语法	**empty_gif**;
默认	——
上下文	location

开启针对 `location` 的模块处理。

原文档

http://nginx.org/en/docs/http/ngx_http_empty_gif_module.html

ngx_http_f4f_module

`ngx_http_f4f_module` 模块为 Adobe HTTP 动态流（HDS）提供服务端支持。

该模块以 `/videoSeg1-Frag1` 形式处理 HTTP 动态流请求，使用 `videoSeg1.f4x` 索引文件从 `videoSeg1.f4f` 文件中提取所需的片段。该模块是 Apache 的 Adobe f4f 模块（HTTP Origin Module）的替代品。

它需要通过 Adobe 的 f4fpackager 进行预处理，有关详细信息，请参阅相关文档。

> 此模块作为我们商业订阅的一部分。

示例配置

```
location /video/ {
    f4f;
    ...
}
```

指令

f4f

-	说明
语法	**f4f**;
默认	——
上下文	location

开启针对 `location` 的模块处理。

f4f_buffer_size

-	说明
语法	**f4f4f_buffer_sizef** `size` ;
默认	f4f_buffer_size 512k;
上下文	http、server、location

设置用于读取 `.f4x` 索引文件的缓冲区的 `size` （大小）。

原文档

http://nginx.org/en/docs/http/ngx_http_f4f_module.html

ngx_http_fastcgi_module

`ngx_http_fastcgi_module` 模块允许将请求传递给 FastCGI 服务器。

示例配置

```
location / {
    fastcgi_pass  localhost:9000;
    fastcgi_index index.php;

    fastcgi_param SCRIPT_FILENAME /home/www/scripts/php$fastcgi_script_name;
    fastcgi_param QUERY_STRING    $query_string;
    fastcgi_param REQUEST_METHOD  $request_method;
    fastcgi_param CONTENT_TYPE    $content_type;
    fastcgi_param CONTENT_LENGTH  $content_length;
}
```

指令

fastcgi_bind

-	说明
语法	**fastcgi_bind** `address [transparent]` | `off` ;
默认	——
上下文	http、server、location
提示	该指令在 0.8.22 版本中出现

通过一个可选的端口（1.11.2）从指定的本地 IP 地址发出到 FastCGI 服务器的传出连接。参数值可以包含变量（1.3.12）。特殊值 `off`（1.3.12）取消从上层配置级别继承到的 `fastcgi_bind` 指令作用，这允许系统自动分配本地 IP 地址和端口。

`transparent` 参数（1.11.0）允许从非本地 IP 地址（例如来自客户端的真实 IP 地址）的到 FastCGI 服务器的传出连接：

```
fastcgi_bind $remote_addr transparent;
```

为了使这个参数起作用，有必要以超级用户权限运行 nginx 工作进程，并配置内核路由来拦截来自 FastCGI 服务器的网络流量。

fastcgi_buffer_size

-	说明
语法	**fastcgi_buffer_size** `size` ;
默认	fastcgi_buffer_size 4k|8k;
上下文	http、server、location

设置读取 FastCGI 服务器收到的响应的第一部分的缓冲区的 `size`（大小）。该部分通常包含一个小的响应头。默认情况下，缓冲区大小等于一个内存页。为 4K 或 8K，因平台而异。但是，它可以设置得更小。

fastcgi_buffering

-	说明
语法	**fastcgi_buffering** `on` | `off` ;
默认	fastcgi_buffering on;
上下文	http、server、location
提示	该指令在 1.5.6 版本中出现

启用或禁用来自 FastCGI 服务器的响应缓冲。

当启用缓冲时，nginx 会尽可能快地收到接收来自 FastCGI 服务器的响应，并将其保存到由 fastcgi_buffer_size 和 fastcgi_buffers 指令设置的缓冲区中。如果内存放不下整个响应，响应的一部分可以保存到磁盘上的临时文件中。写入临时文件由 fastcgi_max_temp_file_size 和 fastcgi_temp_file_write_size 指令控制。

当缓冲被禁用时，nginx 在收到响应时立即同步传递给客户端，不会尝试从 FastCGI 服务器读取整个响应。nginx 一次可以从服务器接收的最大数据量由 fastcgi_buffer_size 指令设置。

通过在 `X-Accel-Buffering` 响应头字段中通过 `yes` 或 `no` 也可以启用或禁用缓冲。可以使用 fastcgi_ignore_headers 指令禁用此功能。

fastcgi_buffes

-	说明
语法	**fastcgi_buffes** `number size` ;
默认	fastcgi_buffers 8 4k|8k;
上下文	http、server、location

设置单个连接从 FastCGI 服务器读取响应的缓冲区的 `number` （数量）和 `size` （大小）。默认情况下，缓冲区大小等于一个内存页。为 4K 或 8K，因平台而异。

fastcgi_busy_buffers_size

-	说明
语法	**fastcgi_busy_buffers_size** `size` ;
默认	fastcgi_busy_buffers_size 8k|16k;
上下文	http、server、location

当启用 FastCGI 服务器响应缓冲时，限制缓冲区的总大小（ `size` ）在当响应尚未被完全读取时可向客户端发送响应。同时，其余的缓冲区可以用来读取响应，如果需要的话，缓冲部分响应到临时文件中。默认情况下， `size` 受 fastcgi_buffer_size 和 fastcgi_buffers 指令设置的两个缓冲区的大小限制。

fastcgi_cache

-	说明
语法	**fastcgi_cache** `zone` | `off` ;
默认	fastcgi_cache off;
上下文	http、server、location

定义用于缓存的共享内存区域。同一个区域可以在几个地方使用。参数值可以包含变量（1.7.9）。`off` 参数将禁用从上级配置级别继承的缓存配置。

fastcgi_cache_background_update

-	说明
语法	**fastcgi_cache_background_update** `on` | `off` ;
默认	fastcgi_cache_background_update off;
上下文	http、server、location
提示	该指令在 1.11.10. 版本中出现

允许启动后台子请求来更新过期的缓存项，而过时的缓存响应则返回给客户端。请注意，有必要在更新时允许使用陈旧的缓存响应。

fastcgi_cache_bypass

-	说明
语法	**fastcgi_cache_bypass** `string ...` ;
默认	——
上下文	http、server、location

定义不从缓存中获取响应的条件。如果字符串参数中有一个值不为空且不等于 `0`，则不会从缓存中获取响应：

```
fastcgi_cache_bypass $cookie_nocache $arg_nocache$arg_comment;
fastcgi_cache_bypass $http_pragma    $http_authorization;
```

可以和 fastcgi_no_cache 指令一起使用。

fastcgi_cache_key

-	说明
语法	**fastcgi_cache_key** `string` ;
默认	——
上下文	http、server、location

为缓存定义一个 key，例如：

```
fastcgi_cache_key localhost:9000$request_uri;
```

fastcgi_cache_lock

-	说明
语法	**fastcgi_cache_lock** `on` | `off` ;
默认	fastcgi_cache_lock off;
上下文	http、server、location
提示	该指令在 1.1.12 版本中出现

当启用时，同一时间只允许一个请求通过将请求传递给 FastCGI 服务器来填充 fastcgi_cache_key 指令标识的新缓存元素。同一缓存元素的其他请求将等待响应出现在缓存中，或等待此元素的缓存锁释放，直到 fastcgi_cache_lock_timeout 指令设置的时间。

fastcgi_cache_lock_age

-	说明
语法	**fastcgi_cache_lock_age** `time` ;
默认	fastcgi_cache_lock_age 5s;
上下文	http、server、location
提示	该指令在 1.7.8 版本中出现

如果传递给 FastCGI 服务器的最后一个请求填充新缓存元素没能在指定的 `time` 内完成，则可能会有其他另一个请求被传递给 FastCGI 服务器。

fastcgi_cache_lock_timeout

-	说明
语法	**fastcgi_cache_lock_timeout** `time` ;
默认	fastcgi_cache_lock_timeout 5s;
上下文	http、server、location
提示	该指令在 1.1.12 版本中出现

设置 fastcgi_cache_lock 的超时时间。当时间到期时，请求将被传递给 FastCGI 服务器，但是，响应不会被缓存。

> 在 1.7.8 之前，响应可以被缓存。

fastcgi_cache_max_range_offset

-	说明
语法	**fastcgi_cache_max_range_offset** `number` ;
默认	——
上下文	http、server、location
提示	该指令在 1.11.6 版本中出现

为 byte-range 请求设置字节偏移量。如果 range 超出 `number` （偏移量），range 请求将被传递给 FastCGI 服务器，并且不会缓存响应。

fastcgi_cache_methods

-	说明
语法	**fastcgi_cache_methods** `GET` | `HEAD` | `POST ...` ;
默认	fastcgi_cache_methods GET HEAD;
上下文	http、server、location
提示	该指令在 0.7.59 版本中出现

如果此指令中存在当前客户端请求方法，那么响应将被缓存。虽然 `GET` 和 `HEAD` 方法总是在该列表中，但我们还是建议您明确指定它们。另请参阅 fastcgi_no_cache 指令。

fastcgi_cache_min_uses

-	说明
语法	**fastcgi_cache_min_uses** `number` ;
默认	fastcgi_cache_min_uses 1;
上下文	http、server、location

设置指定数量（ `number` ）请求后响应将被缓存。

fastcgi_cache_path

-	说明
语法	**fastcgi_cache_path** `path [levels=levels] [use_temp_path=on|off] keys_zone=name:size [inactive=time] [max_size=size] [manager_files=number] [manager_sleep=time] [manager_threshold=time] [loader_files=number] [loader_sleep=time] [loader_threshold=time] [purger=on|off] [purger_files=number] [purger_sleep=time] [purger_threshold=time]` ;
默认	——
上下文	http

设置缓存的路径和其他参数。缓存数据存储在文件中。缓存中的 key 和文件名是代理 URL 经过 MD5 函数处理后得到的值。`levels` 参数定义缓存的层次结构级别：范围从 `1` 到 `3`，每个级别可接受值为 `1` 或 `2`。例如，在以下配置中

```
fastcgi_cache_path /data/nginx/cache levels=1:2 keys_zone=one:10m;
```

缓存中的文件名如下所示：

```
/data/nginx/cache/c/29/b7f54b2df7773722d382f4809d65029c
```

首先将缓存的响应写入临时文件，然后重命名该文件。从 0.8.9 版本开始，临时文件和缓存可以放在不同的文件系统上。但是，请注意，在这种情况下，文件复制将要跨两个文件系统，而不是简单的重命名操作。因此建议，对于任何给定的位置，缓存和保存临时文件的目录都应该放在同一个文件系统上。临时文件的目录根据 `use_temp_path` 参数（1.7.10）设置。如果忽略此参数或将其设置为 `on`，则将使用由 fastcgi_temp_path 指令设置的目录。如果该值设置为 `off`，临时文件将直接放在缓存目录中。

另外，所有活跃的 key 和有关数据的信息都存储在共享内存区中，其名称和大小由 `keys_zone` 参数配置。一个兆字节的区域可以存储大约 8 千个 key。

> 作为商业订阅的一部分，共享内存区还存储其他缓存信息，因此，需要为相同数量的 key区域大小。例如，一个兆字节区域可以存储大约 4 千个 key。

在 `inactive` 参数指定的时间内未被访问的缓存数据将从缓存中删除。默认情况下，`inactive` 设置为 10 分钟。

“缓存管理器”（cache manager）进程监视的最大缓存大小由 `max_size` 参数设置。当超过此大小时，它将删除最近最少使用的数据。数据在由 `manager_files`、`manager_threshold` 和 `manager_sleep` 参数（1.11.5）配置下进行迭代删除。在一次迭代中，不会超过

`manager_files` 项被删除（默认为 100）。一次迭代的持续时间受到 `manager_threshold` 参数（默认为 200 毫秒）的限制。在每次迭代之间存在间隔时间，由 `manager_sleep` 参数（默认为 50 毫秒）配置。

开始后一分钟，“缓存加载器”（cache loader）进程被激活。它将先前存储在文件系统中的缓存数据的有关信息加载到缓存区中。加载也是在迭代中完成。在每一次迭代中，不会加载 `loader_files` 个项（默认情况下为 100）。此外，每一次迭代的持续时间受到 `loader_threshold` 参数的限制（默认情况下为 200 毫秒）。在迭代之间存在间隔时间，由 `loader_sleep` 参数（默认为 50 毫秒）配置。

此外，以下参数作为我们商业订阅的一部分：

- `purger=on|off` 指明缓存清除程序（1.7.12）是否将与通配符键匹配的缓存条目从磁盘中删除。将该参数设置为 `on`（默认为 `off`）将激活“缓存清除器”（cache purger）进程，该进程不断遍历所有缓存条目并删除与通配符匹配的条目。
- `purger_files=number` 设置在一次迭代期间将要扫描的条目数量（1.7.12）。默认情况下，`purger_files` 设置为 10。
- `purger_threshold=number` 设置一次迭代的持续时间（1.7.12）。默认情况下，`purger_threshold` 设置为 50 毫秒。
- `purger_sleep=number` 在迭代之间设置暂停时间（1.7.12）。默认情况下，`purger_sleep` 设置为 50 毫秒。

> 在 1.7.3、1.7.7 和 1.11.10 版本中，缓存头格式发生了更改。升级到更新的 nginx 版本后，以前缓存的响应将视为无效。

fastcgi_cache_purge

-	说明
语法	**fastcgi_cache_purge** `string ...`;
默认	——
上下文	http、server、location
提示	该指令在 1.5.7 版本中出现

定义将请求视为缓存清除请求的条件。如果 string 参数中至少有一个不为空的值并且不等于“0”，则带有相应缓存键的缓存条目将被删除。通过返回 204（无内容）响应来表示操作成功。

如果清除请求的缓存键以星号（`*`）结尾，则将匹配通配符键的所有缓存条目从缓存中删除。但是，这些条目仍然保留在磁盘上，直到它们因为不活跃而被删除或被缓存清除程序（1.7.12）处理，或者客户端尝试访问它们。

配置示例：

```
fastcgi_cache_path /data/nginx/cache keys_zone=cache_zone:10m;

map $request_method $purge_method {
    PURGE   1;
    default 0;
}

server {
    ...
    location / {
        fastcgi_pass        backend;
        fastcgi_cache       cache_zone;
        fastcgi_cache_key   $uri;
        fastcgi_cache_purge $purge_method;
    }
}
```

> 该功能可作为我们商业订阅的一部分。

fastcgi_cache_revalidate

-	说明
语法	**fastcgi_cache_purge** `on` | `off` ;
默认	fastcgi_cache_revalidate off;
上下文	http、server、location
提示	该指令在 1.5.7 版本中出现

开启使用带有 `If-Modified-Since` 和 `If-None-Match` 头字段的条件请求对过期缓存项进行重新验证。

fastcgi_cache_use_stale

-	说明
语法	**fastcgi_cache_use_stale** `error` | `timeout` | `invalid_header` | `updating` | `http_500` | `http_503` | `http_403` | `http_404` | `http_429` | `off` ... ;
默认	fastcgi_cache_use_stale off;
上下文	http、server、location

当在与 FastCGI 服务器通信期间发生错误时可以使用陈旧的缓存响应。该指令的参数与 fastcgi_next_upstream 指令的参数相匹配。

如果无法选择使用 FastCGI 服务器处理请求，则 `error` 参数还允许使用陈旧的缓存响应。

此外，如果它当前正在更新，`updating` 参数允许使用陈旧的缓存响应。这样可以在更新缓存数据时最大限度地减少对 FastCGI 服务器的访问次数。

也可以在响应头中直接启用在响应变为陈旧的指定秒数后使用陈旧的缓存响应（1.11.10）。这比使用指令参数的优先级低。

- `Cache-Control` 头字段的 `stale-while-revalidate` 扩展允许使用陈旧的缓存响应当它正在更新。
- `Cache-Control` 头字段的 `stale-if-error` 扩展允许在发生错误时使用陈旧的缓存响应。

为了最大限度地减少填充新缓存元素时对 FastCGI 服务器的访问次数，可以使用 fastcgi_cache_lock 指令。

fastcgi_cache_valid

-	说明
语法	**fastcgi_cache_valid** `[code ...] time`;
默认	——
上下文	http、server、location

为不同的响应码设置缓存时间。例如：

```
fastcgi_cache_valid 200 302 10m;
fastcgi_cache_valid 404      1m;
```

对响应码为 200 和 302 的响应设置 10 分钟缓存，对响应码为 404 的响应设置为 1 分钟。

如果只指定缓存时间（`time`）：

```
fastcgi_cache_valid 5m;
```

那么只缓存 200 、301 和 302 响应。

另外，可以指定 `any` 参数来缓存任何响应：

```
fastcgi_cache_valid 200 302 10m;
fastcgi_cache_valid 301      1h;
fastcgi_cache_valid any      1m;
```

缓存参数也可以直接在响应头中设置。这比使用指令设置缓存时间具有更高的优先级。

- `X-Accel-Expires` 头字段以秒为单位设置响应的缓存时间。零值会禁用响应缓存。如果该值以 `@` 前缀开头，则它会设置自 Epoch 以来的绝对时间（以秒为单位），最多可以缓存该时间段内的响应。
- 如果头中不包含 `X-Accel-Expires` 字段，则可以在头字段 `Expires` 或 `Cache-Control` 中设置缓存参数。
- 如果头中包含 `Set-Cookie` 字段，则不会缓存此类响应。
- 如果头中包含具有特殊值 `*` 的 `Vary` 字段，则这种响应不会被缓存（1.7.7）。如果头中包含带有另一个值的 `Vary` 字段，考虑到相应的请求头字段（1.7.7），这样的响应将被缓存。

使用 fastcgi_ignore_headers 指令可以禁用一个或多个响应头字段的处理。

fastcgi_catch_stderr

-	说明
语法	**fastcgi_catch_stderr** `string` ;
默认	——
上下文	http、server、location

设置一个字符串，用于在从 FastCGI 服务器接收到的响应的错误流中搜索匹配。如果找到该字符串，则认为 FastCGI 服务器返回无效响应。此时将启用 nginx 中的应用程序错误处理，例如：

```
location /php {
    fastcgi_pass backend:9000;
    ...
    fastcgi_catch_stderr "PHP Fatal error";
    fastcgi_next_upstream error timeout invalid_header;
}
```

fastcgi_connect_timeout

-	说明
语法	**fastcgi_connect_timeout** `time` ;
默认	fastcgi_connect_timeout 60s;
上下文	http、server、location

设置与 FastCGI 服务器建立连接的超时时间。需要注意的是，这个超时通常不能超过 75 秒。

fastcgi_force_ranges

-	说明
语法	**fastcgi_force_ranges** `on` | `off` ;
默认	fastcgi_force_ranges off;
上下文	http、server、location
提示	该指令在 1.7.7 版本中出现

启用来自 FastCGI 服务器的缓存和未缓存响应的 byte-range 支持，忽略响应中的 `Accept-Ranges` 头字段。

fastcgi_hide_header

-	说明
语法	**fastcgi_hide_header** `field` ;
默认	——
上下文	http、server、location

默认情况下，nginx 不会将 FastCGI 服务器响应中的头字段 `Status` 和 `X-Accel-...` 传递给客户端。`fastcgi_hide_header` 指令设置不会被传递的附加字段。但是，如果需要允许传递字段，则可以使用 fastcgi_pass_header 指令。

fastcgi_ignore_client_abort

-	说明
语法	**fastcgi_ignore_client_abort** `on` | `off` ;
默认	fastcgi_ignore_client_abort off;
上下文	http、server、location

确定当客户端关闭连接而不等待响应时是否关闭与 FastCGI 服务器的连接。

fastcgi_ignore_headers

-	说明
语法	**fastcgi_ignore_headers** `field ...` ;
默认	——
上下文	http、server、location

禁止处理来自 FastCGI 服务器的某些响应头字段。以下字段将被忽略：`X-Accel-Redirect`、`X-Accel-Expires`、`X-Accel-Limit-Rate`（1.1.6）、`X-Accel-Buffering`（1.1.6）、`X-Accel-Charset`（1.1.6）、`Expires`、`Cache-Control`、`Set-Cookie`（0.8.44）和 `Vary`（1.7.7）。

如果未禁用，则处理这些头字段产生以下效果：

- `X-Accel-Expires`、`Expires`、`Cache-Control`、`Set-Cookie` 和 `Vary` 设置响应缓存的参数
- `X-Accel-Redirect` 执行内部重定向到指定的 URI
- `X-Accel-Limit-Rate` 设置响应的传送速率限制回客户端
- `X-Accel-Buffering` 启用或禁用缓冲响应
- `X-Accel-Charset` 设置所需的响应字符集

fastcgi_index

-	说明
语法	**fastcgi_index** `name` ;
默认	——
上下文	http、server、location

在 `$fastcgi_script_name` 变量的值中设置一个文件名，该文件名追加到 URL 后面并以一个斜杠结尾。例如以下设置

```
fastcgi_index index.php;
fastcgi_param SCRIPT_FILENAME /home/www/scripts/php$fastcgi_script_name;
```

和 `/page.php` 请求，`SCRIPT_FILENAME` 参数将等于 `/home/www/scripts/php/page.php`，并且 `/` 请求将等于 `/home/www/scripts/php/index.php`。

fastcgi_intercept_errors

-	说明
语法	**fastcgi_intercept_errors** `on` | `off` ;
默认	fastcgi_intercept_errors off;
上下文	http、server、location

确定当 FastCGI 服务器响应码大于或等于 300 时是否应传递给客户端，或者拦截并重定向到 nginx 以便使用 error_page 指令进行处理。

fastcgi_keep_conn

-	说明
语法	**fastcgi_keep_conn** `on` | `off` ;
默认	fastcgi_keep_conn off;
上下文	http、server、location
提示	该指令在 1.1.4 版本中出现

默认情况下，FastCGI 服务器将在发送响应后立即关闭连接。但是，如果当此指令设置为 `on` 值，则 nginx 将指示 FastCGI 服务器保持连接处于打开状态。这对保持 FastCGI 服务器连接 keepalive 尤为重要。

fastcgi_limit_rate

-	说明
语法	**fastcgi_limit_rate** `rate` ;
默认	fastcgi_limit_rate 0;
上下文	http、server、location
提示	该指令在 1.7.7 版本中出现

限制读取 FastCGI 服务器响应的速度。`rate` 以每秒字节数为单位。零值则禁用速率限制。该限制是针对每个请求设置的，因此如果 nginx 同时打开两个连接到 FastCFI 服务器的连接，则整体速率将是指定限制的两倍。该限制仅在启用缓冲来自 FastCGI 服务器的响应时才起作用。

fastcgi_max_temp_file_size

-	说明
语法	**fastcgi_max_temp_file_size** `size` ;
默认	fastcgi_max_temp_file_size 1024m;
上下文	http、server、location

当启用缓冲](#fastcgi_buffering)来自 FastCGI 服务器的响应时并且整个响应不适合由 {fastcgi_buffer_size 和 fastcgi_buffers 指令设置的缓冲时，响应的一部分可以保存到临时文件中。该指令用于设置临时文件的最大大小（ `size` ）。一次写入临时文件的数据大小由 fastcgi_temp_file_write_size 指令设置。

零值将禁用临时文件响应缓冲。

> 此限制不适用于将要缓存或存储在磁盘上的响应。

fastcgi_next_upstream

-	说明
语法	**fastcgi_next_upstream** `error` | `timeout` | `invalid_header` | `http_500` | `http_503` | `http_403` | `http_404` | `http_429` | `non_idempotent` | `off ...` ;
默认	fastcgi_next_upstream error timeout;
上下文	http、server、location

指定在哪些情况下请求应传递给下一台服务器：

- `erorr`

  在与服务器建立连接、传递请求或读取响应头时发生错误

- `timeout`

  在与服务器建立连接、传递请求或读取响应头时发生超时

- `invalid_header`

  服务器返回了空的或无效的响应

- `http_500`

  服务器返回 500 响应码

- `http_503`

  服务器返回 503 响应码

- `http_403`

  服务器返回 403 响应码

- `http_404`

  服务器返回 404 响应码

- `http_429`

  服务器返回 429 响应码（1.11.13）

- `non_idempotent`

通常，如果请求已发送到上游服务器（1.9.13），则具有非幂等方法（POST、LOCK、PATCH）的请求不会传递到下一个服务器，使这个选项明确允许重试这样的请求

- `off`

  禁用将请求传递给下一个服务器

我们应该记住，只有在没有任何内容发送给客户端的情况下，才能将请求传递给下一台服务器。也就是说，如果在响应传输过程中发生错误或超时，要修复是不可能的。

该指令还定义了与服务器进行通信的不成功尝试。`erorr`、`timeout` 和 `invalid_header` 的情况总是被认为是不成功的尝试，即使它们没有在指令中指定。只有在指令中指定了 `http_500`、`http_503` 和 `http_429` 的情况下，它们才被视为不成功尝试。`http_403` 和 `http_404` 的情况永远不会被视为不成功尝试。

将请求传递给下一台服务器可能受到尝试次数和时间的限制。

fastcgi_next_upstream_timeout

-	说明
语法	**fastcgi_next_upstream_timeout** `time` ;
默认	fastcgi_next_upstream_timeout 0;
上下文	http、server、location
提示	该指令在 1.7.5 版本中出现

限制请求可以传递到下一个服务器的时间。`0` 值关闭此限制。

fastcgi_next_upstream_tries

-	说明
语法	**fastcgi_next_upstream_tries** `number` ;
默认	fastcgi_next_upstream_tries 0;
上下文	http、server、location
提示	该指令在 1.7.5 版本中出现

限制将请求传递到下一个服务器的尝试次数。`0` 值关闭此限制。

fastcgi_no_cache

-	说明
语法	**fastcgi_no_cache** `string ...` ;
默认	——
上下文	http、server、location

定义响应不会保存到缓存中的条件。如果 `string` 参数中有一个值不为空且不等于 `0`，则不会保存响应：

```
fastcgi_no_cache $cookie_nocache $arg_nocache$arg_comment;
fastcgi_no_cache $http_pragma    $http_authorization;
```

可以与 fastcgi_cache_bypass 指令一起使用。

fastcgi_param

-	说明
语法	**fastcgi_param** `parameter value [if_not_empty]` ;
默认	——
上下文	http、server、location

设置应传递给 FastCGI 服务器的 `parameter` (参数)。该值可以包含文本、变量及其组合。当且仅当在当前级别上没有定义 fastcgi_param 指令时，这些指令才从前一级继承。

以下示例展示了 PHP 的最小要求配置：

```
fastcgi_param SCRIPT_FILENAME /home/www/scripts/php$fastcgi_script_name;
fastcgi_param QUERY_STRING    $query_string;
```

`SCRIPT_FILENAME` 参数在 PHP 中用于确定脚本名称，`QUERY_STRING` 参数用于传递请求参数。

对于处理 POST 请求的脚本，还需要以下三个参数：

```
fastcgi_param REQUEST_METHOD  $request_method;
fastcgi_param CONTENT_TYPE    $content_type;
fastcgi_param CONTENT_LENGTH  $content_length;
```

如果 PHP 使用了 `--enable-force-cgi-redirect` 配置参数构建，则还应该使用值 `200` 传递 `REDIRECT_STATUS` 参数：

```
fastcgi_param REDIRECT_STATUS 200;
```

如果该指令是通过 `if_not_empty` （1.1.11）指定的，那么只有当它的值不为空时，这个参数才会被传递给服务器：

```
fastcgi_param HTTPS           $https if_not_empty;
```

fastcgi_pass

-	说明
语法	**fastcgi_pass** `address` ;
默认	——
上下文	http、server、location

设置 FastCGI 服务器的地址。该地址可以指定为域名或 IP 地址，以及端口：

```
fastcgi_pass localhost:9000;
```

或者作为 UNIX 域套接字路径：

```
fastcgi_pass unix:/tmp/fastcgi.socket;
```

如果域名解析为多个地址，则所有这些地址都将以循环方式使用。另外，地址可以被指定为服务器组。

参数值可以包含变量。在这种情况下，如果地址被指定为域名，则在所描述的服务器组中搜索名称，如果未找到，则使用解析器来确定。

fastcgi_pass_header

-	说明
语法	**fastcgi_pass_header** `field` ;
默认	——
上下文	http、server、location

允许从 FastCGI 服务器向客户端传递隐藏禁用的头字段。

fastcgi_pass_request_body

-	说明
语法	**fastcgi_pass_request_body** `on` | `off` ;
默认	fastcgi_pass_request_body on;
上下文	http、server、location

指示是否将原始请求主体传递给 FastCGI 服务器。另请参阅 fastcgi_pass_request_headers 指令。

fastcgi_pass_request_headers

-	说明
语法	**fastcgi_pass_request_headers** `on` | `off` ;
默认	fastcgi_pass_request_headers on;
上下文	http、server、location

指示原始请求的头字段是否传递给 FastCGI 服务器。另请参阅 fastcgi_pass_request_body 指令。

fastcgi_read_timeout

-	说明
语法	**fastcgi_read_timeout** `time` ;
默认	fastcgi_read_timeout 60s;
上下文	http、server、location

定义从 FastCGI 服务器读取响应的超时时间。超时设置在两次连续的读操作之间，而不是传输整个响应的过程。如果 FastCGI 服务器在此时间内没有发送任何内容，则连接将被关闭。

fastcgi_request_buffering

-	说明
语法	**fastcgi_request_buffering** `on` | `off` ;
默认	fastcgi_request_buffering on;
上下文	http、server、location
提示	该指令在 1.7.11 版本中出现

启用或禁用客户端请求体缓冲。

启用缓冲时，在将请求发送到 FastCGI 服务器之前，将从客户端读取整个请求体。

当缓冲被禁用时，请求体在收到时立即发送到 FastCGI 服务器。在这种情况下，如果 nginx 已经开始发送请求体，则请求不能传递到下一个服务器。

fastcgi_send_lowat

-	说明
语法	**fastcgi_send_lowat** `size` ;
默认	fastcgi_send_lowat 0;
上下文	http、server、location

如果指令设置为非零值，则 nginx 将尝试通过使用 kqueue 方式的 `NOTE_LOWAT` 标志或 `SO_SNDLOWAT` 套接字选项，以指定的 `size` （大小）来最小化传出连接到 FastCGI 服务器上的发送操作次数。

该指令在 Linux、Solaris 和 Windows 上被忽略。

fastcgi_send_timeout

-	说明
语法	**fastcgi_send_timeout** `time` ;
默认	fastcgi_send_timeout 60s;
上下文	http、server、location

设置向 FastCGI 服务器发送请求的超时时间。超时设置在两次连续写入操作之间，而不是传输整个请求的过程。如果 FastCGI 服务器在此时间内没有收到任何内容，则连接将关闭。

fastcgi_split_path_info

-	说明
语法	**fastcgi_split_path_info** `regex` ;
默认	fastcgi_send_timeout 60s;
上下文	location

定义一个捕获 `$fastcgi_path_info` 变量值的正则表达式。正则表达式应该有两个捕获：第一个为 `$fastcgi_script_name` 变量的值，第二个为 `$fastcgi_path_info` 变量的值。例如以下设置

```
location ~ ^(.+\.php)(.*)$ {
    fastcgi_split_path_info       ^(.+\.php)(.*)$;
    fastcgi_param SCRIPT_FILENAME /path/to/php$fastcgi_script_name;
    fastcgi_param PATH_INFO       $fastcgi_path_info;
```

和 `/show.php/article/0001` 请求，`SCRIPT_FILENAME` 参数等于 `/path/to/php/show.php`，并且 `PATH_INFO` 参数等于 `/article/0001`。

fastcgi_store

-	说明
语法	**fastcgi_store** `on` | `off` | `string` ;
默认	fastcgi_store off;
上下文	http、server、location

启用将文件保存到磁盘。`on` 参数将文件保存为与指令 alias 或 root 相对应的路径。`off` 参数禁用保存文件。另外，可以使用带变量的字符串显式设置文件名：

```
fastcgi_store /data/www$original_uri;
```

文件的修改时间根据收到的 `Last-Modified` 响应头字段设置。首先将响应写入临时文件，然后重命名该文件。从 0.8.9 版本开始，临时文件和持久存储可以放在不同的文件系统上。但是，请注意，在这种情况下，文件将跨两个文件系统进行复制，而不是简单地进行重命名操作。因此建议，对于任何给定位置，保存的文件和由 fastcgi_temp_path 指令设置的保存临时文件的目录都放在同一个文件系统上。

该指令可用于创建静态不可更改文件的本地副本，例如：

```
location /images/ {
    root                /data/www;
    error_page          404 = /fetch$uri;
}

location /fetch/ {
    internal;

    fastcgi_pass        backend:9000;
    ...

    fastcgi_store       on;
    fastcgi_store_access user:rw group:rw all:r;
    fastcgi_temp_path   /data/temp;

    alias               /data/www/;
}
```

fastcgi_store_access

-	说明
语法	**fastcgi_store_access** `users:permissions ...` ;
默认	fastcgi_store_access user:rw;
上下文	http、server、location

为新创建的文件和目录设置访问权限，例如：

```
fastcgi_store_access user:rw group:rw all:r;
```

如果指定了任何组或所有访问权限，则可以省略用户权限

```
fastcgi_store_access group:rw all:r;
```

fastcgi_temp_file_write_size

-	说明	
语法	**fastcgi_temp_file_write_size** `size` ;	
默认	fastcgi_temp_file_write_size 8k	16k;
上下文	http、server、location	

设置当开启缓冲 FastCGI 服务器响应到临时文件时，限制写入临时文件的数据 `size`（大小）。默认情况下，大小受 fastcgi_buffer_size 和 fastcgi_buffers 指令设置的两个缓冲区限制。临时文件的最大大小由 fastcgi_max_temp_file_size 指令设置。

fastcgi_temp_path

-	说明
语法	**fastcgi_temp_path** `path [level1 [level2 [level3]]]`;
默认	astcgi_temp_path fastcgi_temp;
上下文	http、server、location

定义一个目录，用于存储从 FastCGI 服务器接收到的数据的临时文件。指定目录下最多可有三级子目录。例如以下配置

```
fastcgi_temp_path /spool/nginx/fastcgi_temp 1 2;
```

临时文件如下所示：

```
/spool/nginx/fastcgi_temp/7/45/00000123457
```

另请参见 fastcgi_cache_path 指令的 `use_temp_path` 参数。

传参到 FastCGI 服务器

HTTP 请求头字段作为参数传递给 FastCGI 服务器。在作为 FastCGI 服务器运行的应用程序和脚本中，这些参数通常作为环境变量提供。例如，`User-Agent` 头字段作为 `HTTP_USER_AGENT` 参数传递。除 HTTP 请求头字段外，还可以使用 fastcgi_param 指令传递任意参数。

内嵌变量

`ngx_http_fastcgi_module` 模块支持在 fastcgi_param 指令设置参数时使用内嵌变量：

- `$fastcgi_script_name`

  请求 URI，或者如果 URI 以斜杠结尾，则请求 URI 的索引文件名称由 fastcgi_index 指令配置。该变量可用于设置 `SCRIPT_FILENAME` 和 `PATH_TRANSLATED` 参数，以确定 PHP 中的脚本名称。例如，对 `/info/` 请求的指令设置

```
fastcgi_index index.php;
fastcgi_param SCRIPT_FILENAME /home/www/scripts/php$fastcgi_script_name;
```

`SCRIPT_FILENAME` 参数等于 `/home/www/scripts/php/info/index.php` 。

使用 fastcgi_split_path_info 指令时， `$fastcgi_script_name` 变量等于指令设置的第一个捕获值。

- `$fastcgi_path_info`

  由 fastcgi_split_path_info 指令设置的第二个捕获值。这个变量可以用来设置 `PATH_INFO` 参数。

原文档

http://nginx.org/en/docs/http/ngx_http_fastcgi_module.html

ngx_http_flv_module

`ngx_http_flv_module` 模块为 Flash 视频（FLV）文件提供伪流服务端支持。

它通过发送回文件的内容来处理请求 URI 查询字符串中带有特定 `start` 参数的请求，文件的内容从请求字节偏移开始的且 FLV 头为前缀。

该模块不是默认构的，您可以在构建时使用 `--with-http_flv_module` 配置参数启用。

示例配置

```
location ~ \.flv$ {
    flv;
}
```

指令

flv

-	说明
语法	**flv**;
默认	——
上下文	location

开启针对 `location` 的模块处理。

原文档

http://nginx.org/en/docs/http/ngx_http_flv_module.html

ngx_http_geo_module

`ngx_http_geo_module` 模块创建带值变量需依赖客户端 IP 地址。

示例配置

```
geo $geo {
    default        0;

    127.0.0.1      2;
    192.168.1.0/24 1;
    10.1.0.0/16    1;

    ::1            2;
    2001:0db8::/32 1;
}
```

指令

geo

-	说明
语法	**geo** `[$address] $variable { ... }`；
默认	——
上下文	http

描述指定变量的值对客户端 IP 地址的依赖。默认情况下，地址来自 `$remote_addr` 变量，但也可以从另一个变量（0.7.27）中获取，例如：

```
geo $arg_remote_addr $geo {
    ...;
}
```

> 由于变量仅在使用时才生效，因此即使存在大量已声明的 `geo` 变量也不会增加请求处理开销。

如果变量的值不是有效的 IP 地址，则使用 `255.255.255.255` 地址。

> 从1.3.10 版本和 1.2.7 版本开始支持 IPv6 前缀。

也支持以下特殊参数：

- `delete`

  删除指定的网络（0.7.23）。

- `default`

  如果客户端地址与所有指定地址都不匹配，则将设置为该变量的值。当以 CIDR 表示法指定地址时，可以使用 `0.0.0.0/0` 和 `::/0` 代替默认值。未指定默认值时，默认值为空字符串。

- `include`

  包含一个包含地址和值的文件。可包含多个。

- `proxy`

  定义可信地址（0.8.7、0.7.63）。当请求来自可信地址时，将使用来自 `X-Forwarded-For` 请求头字段的地址。与常规地址相比，可信地址是按顺序检查的。

  > 从 1.3.0 版本和 1.2.1 版本开始支持 IPv6 地址。

- `proxy_recursive`

  启用递归地址搜索（1.3.0、1.2.1）。如果递归搜索被禁用，将使用在 `X-Forwarded-For` 中发送的最后一个地址，而不是匹配其中一个可信地址的原始客户端地址。如果启用递归搜索，则将使用在 `X-Forwarded-For` 中发送的最后一个不可信地址，而不是匹配其中一个可信地址的原始客户端地址。

- `ranges`

  表示地址被指定为范围形式（0.7.23）。这个参数应该放在首位。想要加快加载地理区域，地址应按升序排列。

示例：

```
geo $country {
    default         ZZ;
    include         conf/geo.conf;
    delete          127.0.0.0/16;
    proxy           192.168.100.0/24;
    proxy           2001:0db8::/32;

    127.0.0.0/24    US;
    127.0.0.1/32    RU;
    10.1.0.0/16     RU;
    192.168.1.0/24 UK;
}
```

`conf/geo.conf` 文件可能包含以下内容：

```
10.2.0.0/16     RU;
192.168.2.0/24 RU;
```

使用最明确匹配的值。例如，对于 127.0.0.1 地址，将选择 `RU` 值，而不是 `US` 。

范围示例：

```
geo $country {
    ranges;
    default                    ZZ;
    127.0.0.0-127.0.0.0        US;
    127.0.0.1-127.0.0.1        RU;
    127.0.0.1-127.0.0.255      US;
    10.1.0.0-10.1.255.255      RU;
    192.168.1.0-192.168.1.255 UK;
}
```

原文档

http://nginx.org/en/docs/http/ngx_http_geo_module.html

ngx_http_geoip_module

`ngx_http_geoip_module` 模块（0.8.6+）使用预编译的 MaxMind 数据库，其创建带值的变量依赖客户端 IP 地址。

当使用支持 IPv6 的数据库时（1.3.12、1.2.7），IPv4 地址将被视为 IPv4 映射的 IPv6 地址。

此模块不是默认构建的，可以使用 `--with-http_geoip_module` 配置参数启用。

> 该模块需要 MaxMind GeoIP 库。

示例配置

```
http {
    geoip_country         GeoIP.dat;
    geoip_city            GeoLiteCity.dat;
    geoip_proxy           192.168.100.0/24;
    geoip_proxy           2001:0db8::/32;
    geoip_proxy_recursive on;
    ...
```

指令

geoip_country

-	说明
语法	**geoip_country** `file` ;
默认	——
上下文	http

指定一个用于根据客户端 IP 地址确定国家的数据库。使用此数据库时，以下变量可用：

- `$geoip_country_code`

  双字母国家代码，例如 `RU` 、 `US`

- `$geoip_country_code3`

  三个字母的国家代码，例如 `RUS` 、 `USA`

- `$geoip_country_name`

  国家名称，例如 `Russian Federation` 、 `United States`

geoip_city

-	说明
语法	**geoip_city** `file`;
默认	——
上下文	http

指定一个用于根据客户端 IP 地址确定国家、地区和城市的数据库。使用此数据库时，以下变量可用：

- `$geoip_area_code`

  电话区号（仅限美国）

  > 由于相应的数据库字段已弃用，因此此变量可能包含过时的信息

- `$geoip_city_continent_code`

  双字母的大陆码，例如 `EU` 、 `NA`

- `$geoip_city_country_code`

  双字母国家代码，例如 `RU` 、 `US`

- `$geoip_city_country_code3`

  三个字母的国家代码，例如 `RUS` 、 `USA`

- `$geoip_city_country_name`

  国家名称，例如 `Russian Federation` 、 `United States`

- `$geoip_dma_code`

美国的 DMA 地区代码（也称为城市代码），根据 Google AdWords API 中的地理位置定位

- `$geoip_latitude`

  纬度

- `$geoip_longitude`

  经度

- `$geoip_region`

  双符号国家地区代码（地区、领土、州、省、联邦土地等），例如 `48` 、 `DC`

- `$geoip_region_name`

  国家地区名称（地区，领土，州，省，联邦土地等），例如 `Moscow City` 、 `District of Columbia`

- `$geoip_city`

  城市名称，例如 `Moscow` 、 `Washington`

- `$geoip_postal_code`

  邮政编码

geoip_org

-	说明
语法	**geoip_org** `file` ;
默认	——
上下文	http
提示	该指令在 1.0.3 版本中出现

指定用于根据客户端 IP 地址确定组织的数据库。使用此数据库时，以下变量可用：

- `$geoip_org`

  组织名称，例如 `The University of Melbourne`

geoip_proxy

-	说明
语法	**geoip_proxy** `address` | `CIDR` ;
默认	——
上下文	http
提示	该指令在 1.3.0 版本和 1.2.1. 版本中出现

定义可信地址。当请求来自可信地址时，将使用来自 `X-Forwarded-For` 请求头字段的地址。

geoip_proxy_recursive

-	说明
语法	**geoip_proxy_recursive** `on` | `off` ;
默认	geoip_proxy_recursive off;
上下文	http
提示	该指令在 1.3.0 版本和 1.2.1. 版本中出现

如果递归搜索被禁用，那么将使用在 `X-Forwarded-For` 中发送的最后一个地址，而不是匹配其中一个可信地址的原始客户端地址。如果启用递归搜索，则将使用在 `X-Forwarded-For` 中发送的最后一个不可信地址，而不是匹配其中一个可信地址的原始客户端地址。

原文档

http://nginx.org/en/docs/http/ngx_http_geoip_module.html

- grpc_bind

ngx_http_grpc_module

`ngx_http_grpc_module` 模块允许将请求传递给 gRPC 服务器（1.13.10）。该模块需要 ngx_http_v2_module 模块的支持。

示例配置

```
server {
    listen 9000 http2;

    location / {
        grpc_pass 127.0.0.1:9000;
    }
}
```

指令

grpc_bind

-	说明
语法	**map** `address [transparent ]` | `off` ;
默认	——
上下文	http、server、location

连接到一个指定了本地 IP 地址和可选端口的 gRPC 服务器。参数值可以包含变量。特殊值 `off` 取消从上层配置级别继承的 `grpc_bind` 指令的作用，其允许系统自动分配本地 IP 地址和端口。

`transparent` 参数允许出站从非本地 IP 地址到 gRPC 服务器的连接（例如，来自客户端的真实 IP 地址）：

```
grpc_bind $remote_addr transparent;
```

为了使这个参数起作用，通常需要以超级用户权限运行 nginx worker 进程。在 Linux 上，不需要指定 `transparent` 参数，工作进程会继承 master 进程的 `CAP_NET_RAW` 功能。此外，还要配置内核路由表来拦截来自 gRPC 服务器的网络流量。

grpc_buffer_size

-	说明
语法	**grpc_buffer_size** `size` ;
默认	grpc_buffer_size 4k|8k;
上下文	http、server、location

设置用于读取从 gRPC 服务器收到的响应的缓冲区的大小（`size`）。一旦收到响应，响应便会同步传送给客户端。

grpc_connect_timeout

-	说明
语法	**grpc_connect_timeout** `time` ;
默认	grpc_connect_timeout 60s;
上下文	http、server、location

定义与 gRPC 服务器建立连接的超时时间。需要说明的是，超时通常不能超过 75 秒。

grpc_hide_header

-	说明
语法	**grpc_hide_header** `field` ;
默认	——
上下文	http、server、location

默认情况下，nginx 不会将 gRPC 服务器响应中的头字段 `Date` 、 `Server` 和 `X-Accel-...` 传送给客户端。 `grpc_hide_header` 指令设置了不会被传送的附加字段。相反，如果需要允许传送字段，则可以使用 grpc_pass_header 指令设置。

grpc_ignore_headers

-	说明
语法	**grpc_ignore_headers** `field ...` ;
默认	——
上下文	http、server、location

禁用处理来自 gRPC 服务器的某些响应头字段。以下字段可以忽略： `X-Accel-Redirect` 和 `X-Accel-Charset` 。

如果未禁用，处理这些头字段将产生以下作用：

- `X-Accel-Redirect` 执行内部重定向到指定的 URI
- `X-Accel-Charset` 设置所需的响应字符集

grpc_intercept_errors

-	说明
语法	**grpc_intercept_errors** `on` | `off` ;
默认	grpc_intercept_errors off;
上下文	http、server、location

确定状态码大于或等于 300 的 gRPC 服务器响应是应该传送给客户端还是拦截并重定向到 nginx 使用 error_page 指令进行处理。

grpc_next_upstream

-	说明
语法	**grpc_next_upstream** `error` | `timeout` | `invalid_header` | `http_500` | `http_502` | `http_503` | `http_504` | `http_403` | `http_404` | `http_429` | `non_idempotent` | `off ...` ;
默认	grpc_next_upstream error timeout;
上下文	http、server、location

指定在哪些情况下请求应传递给下一个服务器：

- `error`

  在与服务器建立连接、传递请求或读取响应头时发生错误

- `timeout`

  在与服务器建立连接、传递请求或读取响应头时发生超时

- `invalid_header`

  服务器返回了空的或无效的响应

- `http_500`

  服务器返回状态码为 500 的响应

- `http_502`

  服务器返回状态码为 502 的响应

- `http_503`

  服务器返回状态码为 503 的响应

- `http_504`

  服务器返回状态码为 504 的响应

- `http_403`

  服务器返回状态码为 403 的响应

- `http_404`

  服务器返回状态码为 404 的响应

- `http_429`

  服务器返回状态码为 429 的响应

- `non_idempotent`

  通常，如果请求已发送到上游服务器，请求方法为非幂等（POST、LOCK、PATCH）的请求是不会传送到下一个服务器，使这个选项将明确允许重试这样的请求

- `off`

  禁止将请求传递给下一个服务器

我们应该记住，只有在没有任何内容发送给客户端的情况下，才能将请求传递给下一个服务器。也就是说，如果在响应传输过程中发生错误或超时，修复这样的错误是不可能的。

该指令还定义了与服务器进行通信的失败尝试。`error` 、`timeout` 和 `invalid_header` 的情况总是被认为是失败尝试，即使它们没有在指令中指定。只有在指令中指定了 `http_500` 、`http_502` 、`http_503` 、`http_504` 和 `http_429` 的情况下，才会将其视为失败尝试。`http_403` 和 `http_404` 的情况永远不会被视为失败尝试。

将请求传递给下一个服务器可能受到尝试次数和时间的限制。

grpc_next_upstream_timeout

-	说明
语法	**grpc_next_upstream_timeout** `time` ;
默认	grpc_next_upstream_timeout 0;
上下文	http、server、location

限制请求可以传递到下一个服务器的时间。`0` 值表示关闭此限制。

grpc_next_upstream_tries

-	说明
语法	**grpc_next_upstream_tries** `number` ;
默认	grpc_next_upstream_tries 0;
上下文	http、server、location

限制尝试将请求传递到下一个服务器的次数。`0` 值表示关闭此限制。

grpc_pass

-	说明
语法	**grpc_pass** `address` ;
默认	——
上下文	location、location 中的 if

设置 gRPC 服务器地址。该地址可以指定为域名或 IP 地址以及端口：

```
grpc_pass localhost:9000;
```

或使用 UNIX 域套接字路径：

```
grpc_pass unix:/tmp/grpc.socket;
```

或使用 `grpc://` scheme：

```
grpc_pass grpc://127.0.0.1:9000;
```

要 gRPC 配合 SSL，应该使用 `grpcs://` scheme：

```
grpc_pass grpcs://127.0.0.1:443;
```

如果域名解析为多个地址，则这些地址将以循环方式使用。另外，地址可以被指定为服务器组。

grpc_pass_header

-	说明
语法	**grpc_pass_header** `field` ;
默认	——
上下文	http、server、location

允许从 gRPC 服务器向客户端传递忽略的头字段。

grpc_read_timeout

-	说明
语法	**grpc_read_timeout** `time` ;
默认	grpc_read_timeout 60s;
上下文	http、server、location

定义从 gRPC 服务器读取响应的超时时间。超时间隔只在两次连续的读操作之间，而不是整个响应的传输过程。如果 gRPC 服务器在此时间内没有发送任何内容，则连接关闭。

grpc_send_timeout

-	说明
语法	**grpc_send_timeout** `time` ;
默认	grpc_send_timeout 60s;
上下文	http、server、location

设置向 gRPC 服务器发送请求的超时时间。超时间隔只在两次连续写入操作之间，而不是整个请求的传输过程。如果 gRPC 服务器在此时间内没有收到任何内容，则连接将关闭。

grpc_set_header

-	说明
语法	**grpc_set_header** `field value` ;
默认	grpc_set_header Content-Length $content_length;
上下文	http、server、location

允许重新定义或附加字段到传递给 gRPC 服务器的请求头。该值可以包含文本、变量及其组合。当且仅当在当前级别上没有定义 `grpc_set_header` 指令时，这些指令才从上一级继承。

如果头字段的值是一个空字符串，那么这个字段将不会被传递给 gRPC 服务器：

```
grpc_set_header Accept-Encoding "";
```

grpc_ssl_certificate

-	说明
语法	**grpc_ssl_certificate** `file` ;
默认	——
上下文	http、server、location

指定一个带有 PEM 格式证书的文件（ `file` ），用于向 gRPC SSL 服务器进行身份验证。

grpc_ssl_certificate_key

-	说明
语法	**grpc_ssl_certificate_key** `file` ;
默认	——
上下文	http、server、location

指定一个文件（ `file` ），其包含 PEM 格式的密钥，用于对 gRPC SSL 服务器进行身份验证。

可以指定值 `engine:name:id` 来代替 `file` ，其从 OpenSSL 引擎 `name` 加载具有指定 `id` 的密钥。

grpc_ssl_ciphers

-	说明
语法	**grpc_ssl_ciphers** `ciphers` ;
默认	grpc_ssl_ciphers DEFAULT;
上下文	http、server、location

指定对 gRPC SSL 服务器的请求启用的密码。密码格式能被 OpenSSL 库所支持。

完整的列表详情可以使用 `openssl ciphers` 命令查看。

grpc_ssl_crl

-	说明
语法	**grpc_ssl_crl** `file` ;
默认	——
上下文	http、server、location

指定一个带有 PEM 格式的撤销证书（CRL）的文件（ `file` ），用于验证 gRPC SSL 服务器的证书。

grpc_ssl_name

-	说明
语法	**grpc_ssl_name** `name` ;
默认	grpc_ssl_name host from grpc_pass;
上下文	http、server、location

允许重写用于验证 gRPC SSL 服务器的证书并在与 gRPC SSL 服务器建立连接时通过 SNI 传递的服务器名称。

默认情况下，使用 grpc_pass 的主机部分。

grpc_ssl_password_file

-	说明
语法	**grpc_ssl_password_file** `file` ;
默认	——
上下文	http、server、location

指定一个密码为密钥的文件，每个密码在单独的行上指定。加载密钥时会依次尝试每个密码。

grpc_ssl_server_name

-	说明
语法	**grpc_ssl_server_name** `on` | `off` ;
默认	grpc_ssl_server_name off;
上下文	http、server、location

在建立与 gRPC SSL 服务器的连接时，启用或禁用通过 TLS 服务器名称指示扩展（SNI，RFC 6066）传送服务器名称。

grpc_ssl_session_reuse

-	说明
语法	**grpc_ssl_session_reuse** `on` | `off` ;
默认	grpc_ssl_session_reuse on;
上下文	http、server、location

确定在使用 gRPC 服务器时是否可以重用 SSL 会话。如果错误 `SSL3_GET_FINISHED:digest check fialed` 出现在日志中，尝试禁用会话重用。

grpc_ssl_protocols

-	说明
语法	**grpc_ssl_protocols** `[SSLv2] [SSLv3] [TLSv1] [TLSv1.1] [TLSv1.2] [TLSv1.3]` ;
默认	grpc_ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
上下文	http、server、location

对 gRPC SSL 服务器的请求启用指定的协议。

grpc_ssl_trusted_certificate

-	说明
语法	**grpc_ssl_trusted_certificate** `file` ;
默认	——
上下文	http、server、location

指定一个带有 PEM 格式的可信 CA 证书的文件，用于验证 gRPC SSL 服务器的证书。

grpc_ssl_verify

-	说明
语法	**grpc_ssl_verify** `on` | `off` ;
默认	grpc_ssl_verify off;
上下文	http、server、location

启用或禁用验证 gRPC SSL 服务器证书。

grpc_ssl_verify_depth

-	说明
语法	**grpc_ssl_verify_depth** `number` ;
默认	grpc_ssl_verify_depth 1;
上下文	http、server、location

设置 gRPC SSL 服务器证书链中的验证深度。

原文档

http://nginx.org/en/docs/http/ngx_http_grpc_module.html

ngx_http_gunzip_module

`ngx_http_gunzip_module` 模块是一个过滤器，用于对不支持 **gzip** 编码方法的客户端解压缩 `Content-Encoding：gzip` 的响应。当需要存储压缩数据以节省空间并降低 I/O 成本时，该模块将非常有用。

此模块不是默认构建，您可以使用 `--with-http_gunzip_module` 配置参数启用。

示例配置

```
location /storage/ {
    gunzip on;
    ...
}
```

指令

gunzip

-	说明
语法	**gunzip** `on` | `off` ;
默认	gunzip off;
上下文	http、server、location

对缺少 gzip 支持的客户端启用或禁用 gzip 响应解压缩。如果开启，在确定客户端是否支持 gzip 时还会考虑以下指令：gzip_http_version、gzip_proxied 和 gzip_disable。另请参阅 gzip_vary 指令。

gunzip_buffers

-	说明
语法	**gunzip_buffers** `number size` ;
默认	gunzip_buffers 32 4k|16 8k;
上下文	http、server、location

设置用于解压响应的缓冲区的数量（ `number` ）和大小（ `size` ）。默认情况下，缓冲区大小等于一个内存页（4K 或 8K，取决于平台）。

原文档

http://nginx.org/en/docs/http/ngx_http_gunzip_module.html

ngx_http_gzip_module

`ngx_http_gzip_module` 模块是一个使用了 **gzip** 方法压缩响应的过滤器。有助于将传输数据的大小减少一半甚至更多。

示例配置

```
gzip            on;
gzip_min_length 1000;
gzip_proxied    expired no-cache no-store private auth;
gzip_types      text/plain application/xml;
```

`$gzip_ratio` 变量可用于记录实现的压缩比率。

指令

gzip

-	说明
语法	**gzip** `on` | `off` ;
默认	gzip off;
上下文	http、server、location、location 中的 if

启用或禁用响应的 gzip 压缩。

gzip_buffers

-	说明
语法	**gzip_buffers** `number size` ;
默认	gzip_buffers 32 4k|16 8k;
上下文	http、server、location

设置用于压缩响应的缓冲区的数量（ `number` ）和大小（ `size` ）。默认情况下，缓冲区大小等于一个内存页（4K 或 8K，取决于平台）。

> 在 0.7.28 版本之前，默认使用 4 个 4K 或 8K 缓冲区。

gzip_comp_level

-	说明
语法	**gzip_comp_level** `level` ;
默认	gzip_comp_level 1;
上下文	http、server、location

设置响应的 gzip 压缩级别（ `level` ）。值的范围为 1 到 9。

gzip_disable

-	说明
语法	**gzip_disable** `regex ...` ;
默认	——
上下文	http、server、location
提示	该指令在 0.6.23 版本中出现

禁用对与任何指定正则表达式匹配的 `User-Agent` 头字段的请求响应做 gzip 处理。

特殊掩码 `msie6` （0.7.12）对应正则表达式 `MSIE [4-6]\.` ，但效率更高。从 0.8.11 版本开始， `MSIE 6.0; ... SV1` 不包含在此掩码中。

gzip_min_length

-	说明
语法	**gzip_min_length** `length` ;
默认	gzip_min_length 20;
上下文	http、server、location

设置被压缩响应的最小长度。该长度仅由 `Content-Length` 响应头字段确定。

gzip_http_version

-	说明
语法	**gzip_http_version** `1.0` | `1.1` ;
默认	gzip_http_version 1.1;
上下文	http、server、location

设置压缩响应一个请求所需的最小 HTTP 版本。

gzip_proxied

-	说明
语法	**gzip_proxied** `off` | `expired` | `no-cache` | `no-store` | `private` | `no_last_modified` | `no_etag` | `auth` | `any ...` ;
默认	gzip_proxied off;
上下文	http、server、location

根据请求和响应，启用或禁用针对代理请求的响应的 gzip。事实上请求被代理取决于 `Via` 请求头字段是否存在。该指令接受多个参数：

- `off`

  禁用所有代理请求压缩，忽略其他参数

- `expired`

  如果响应头包含 `Expires"` 字段并且其值为禁用缓存，则启用压缩

- `no-cache`

  如果响应头包含具有 `no-cache` 参数的 `Cache-Control` 字段，则启用压缩

- `no-store`

如果响应头包含具有 `no-store` 参数的 `Cache-Control` 字段，则启用压缩

- `private`

  如果响应头包含带有 `private` 参数的 `Cache-Control` 字段，则启用压缩

- `no_last_modified`

  如果响应头不包含 `Last-Modified` 字段，则启用压缩

- `no_etag`

  如果响应头不包含 `ETag` 字段，则启用压缩

- `auth`

  如果请求头包含 `Authorization` 字段，则启用压缩

- `any`

  为所有代理请求启用压缩

gzip_types

-	说明
语法	**gzip_types** `mime-type ...` ;
默认	gzip_types text/html;
上下文	http、server、location

除了 `text/html` 之外，还可以针对指定的 MIME 类型启用 gzip 响应。特殊值 `*` 匹配任何 MIME 类型（0.8.29）。对 `text/html` 类型的响应始终启用压缩。

gzip_vary

-	说明
语法	**gzip_vary** `on` | `off` ;
默认	gzip_vary off;
上下文	http、server、location

如果指令 gzip、gzip_static 或 gunzip 处于激活状态，则启用或禁用插入 `Vary:Accept-Encoding` 响应头字段。

内嵌变量

- `$gzip_ratio`

  实现压缩比率，计算为原始压缩响应大小与压缩后响应大小之间的比率。

原文档

http://nginx.org/en/docs/http/ngx_http_gzip_module.html

ngx_http_gzip_static_module

`ngx_http_gzip_static_module` 模块允许发送以 **.gz** 结尾的预压缩文件替代普通文件。 该模块默认不会被构建到 nginx 中，需要在编译时加入 `--with-http_gzip_static_module` 配置参数启用。

配置示例

```
gzip_static on;
gzip_proxied expired no-cache no-store private auth;
```

指令

gzip_static

-	说明
语法	**gzip_static** `on` | `off` | `always` ;
默认	gzip_static off;
上下文	http、server、location

开启(**on**)或禁用(**off**)会检查预压缩文件是否存在。下列指令也会被影响到 gzip_http_verson，gzip_proxied， gzip_disable， gzip_vary。

值为 **always** (1.3.6)，在所有情况下都会使用压缩文件，不检查客户端是否支持。如果磁盘上没有未被压缩的文件或者 ngx_http_gunzip_module 模块被启用，这个参数非常有用。

文件可以使用 `gzip` 命令，或者任何兼容文件进行压缩。建议压缩文件和源文件的修改日期和时间保持一致。

ngx_http_headers_module

`ngx_http_headers_module` 模块允许将 `Expires` 和 `Cache-Control` 头字段以及任意字段添加到响应头中。

示例配置

```
expires    24h;
expires    modified +24h;
expires    @24h;
expires    0;
expires    -1;
expires    epoch;
expires    $expires;
add_header Cache-Control private;
```

指令

add_header

-	说明
语法	**add_header** `name value [always]` ;
默认	——
上下文	http、server、location、location 中的 if

如果响应代码等于 200、201（1.3.10）、204、206、301、302、303、304、307（1.1.16、1.0.13）或 308（1.13.0），则将指定的字段添加到响应报头中。该值可以包含变量。

可以存在几个 `add_header` 指令。当且仅当在当前级别上没有定义 `add_header` 指令时，这些指令才从上一级继承。

如果指定了 `always` 参数（1.7.5），则无论响应代码为何值，头字段都将被添加。

add_trailer

-	说明
语法	**add_trailer** `number size` ;
默认	——
上下文	http、server、location、location 中的 if
提示	该指令在 1.13.2 版本中出现

如果响应代码等于 200、201、206、301、302、303、307 或 308，则将指定的字段添加到响应的末尾。该值可以包含变量。

可以存在多个 `add_trailer` 指令。当且仅当在当前级别上没有定义 `add_trailer` 指令时，这些指令才从上一级继承。

如果指定 `always` 参数，则无论响应代码为何值，都会添加指定的字段。

expires

-	说明
语法	**expires** `[modified] time` ; **expires** `epoch` | `max` | `off` ;
默认	expires off;
上下文	http、server、location、location 中的 if

如果响应代码等于 200、201（1.3.10）、204、206、301、302、303、304 307（1.1.16、1.0.13）或 308（1.13.0），则启用或禁用添加或修改 `Expires` 和 `Cache-Control` 响应头字段。参数可以是正值或负值。

`Expires` 字段中的时间计算为指令中指定的 `time` 和当前时间的总和。如果使用 `modified` 参数（0.7.0、0.6.32），则计算时间为文件修改时间与指令中指定的 `time` 之和。

另外，可以使用 `@` 前缀指定一天的时间（0.7.9、0.6.34）：

```
expires @15h30m;
```

`epoch` 参数对应于绝对时间 **Thu, 01 Jan 1970 00:00:01 GMT**。`Cache-Control` 字段的内容取决于指定时间的符号：

- 时间为负值 — `Cache-Control:no-cache`
- 时间为正值或为零 — `Cache-Control:max-age=t`，其中 `t` 是指令中指定的时间，单位为秒

`max` 参数将 `Expires` 的值设为 `Thu, 2037 Dec 23:55:55 GMT`，`Cache-Control` 设置为 10 年。

`off` 参数禁止添加或修改 `Expires` 和 `Cache-Control` 响应头字段。

最后一个参数值可以包含变量（1.7.9）：

```
map $sent_http_content_type $expires {
    default         off;
    application/pdf 42d;
    ~image/         max;
}

expires $expires;
```

原文档

http://nginx.org/en/docs/http/ngx_http_headers_module.html

ngx_http_headers_module

`ngx_http_hls_module` 模块为 MP4 和 MOV 媒体文件提供 HTTP Live Streaming（HLS）服务器端支持。这些文件通常具有 `.mp4`、`.m4v`、`.m4a`、`.mov` 或 `.qt` 扩展名。该模块支持 H.264 视频编解码器、AAC 和 MP3 音频编解码器。

对于每个媒体文件，支持两种 URI：

- 带有 `.m3u8` 文件扩展名的播放列表 URI。该 URI 可以接受可选参数：
  - `start` 和 `end` 以秒为单位定义播放列表范围（1.9.0）。
  - `offset` 将初始播放位置移动到以秒为单位的时间偏移（1.9.0）。正值设置播放列表开头的时间偏移量。负值设置播放列表中最后一个片段末尾的时间偏移量。
  - `len` 以秒为单位定义片段长度。
- 带有 `.ts` 文件扩展名的片段 URI。该 URI 可以接受可选参数：
  - `start` 和 `end` 以秒为单位定义片段范围。

> 该模块可作为我们商业订阅的一部分。

示例配置

```
location / {
    hls;
    hls_fragment             5s;
    hls_buffers              10 10m;
    hls_mp4_buffer_size      1m;
    hls_mp4_max_buffer_size 5m;
    root /var/video/;
}
```

在此配置中，`/var/video/test.mp4` 文件支持以下 URI：

```
http://hls.example.com/test.mp4.m3u8?offset=1.000&start=1.000&end=2.200
http://hls.example.com/test.mp4.m3u8?len=8.000
http://hls.example.com/test.mp4.ts?start=1.000&end=2.200
```

指令

hls

-	说明
语法	**hls**;
默认	——
上下文	location

为当前 location 打开 HLS 流。

hls_buffers

-	说明
语法	**hls_buffers** `number size` ;
默认	hls_buffers 8 2m;
上下文	http、server、location

设置用于读取和写入数据帧的缓冲区的最大数量（ `number` ）和大小（ `size` ）。

hls_forward_args

-	说明
语法	**hls_forward_args** `on` | `off` ;
默认	hls_forward_args off;
上下文	http、server、location
提示	该指令在 1.5.12 版本中出现

将播放列表请求中的参数添加到片段的 URI 中。这对于在请求片段时或在使用 ngx_http_secure_link_module 模块保护 HLS 流时执行客户端授权非常有用。

例如，如果客户端请求播放列表 `http://example.com/hls/test.mp4.m3u8?a=1&b=2` ，参数 `a=1` 和 `b=2` 将在参数 `start` 和 `end` 后面添加到片段 URI 中：

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-TARGETDURATION:15
#EXT-X-PLAYLIST-TYPE:VOD

#EXTINF:9.333,
test.mp4.ts?start=0.000&end=9.333&a=1&b=2
#EXTINF:7.167,
test.mp4.ts?start=9.333&end=16.500&a=1&b=2
#EXTINF:5.416,
test.mp4.ts?start=16.500&end=21.916&a=1&b=2
#EXTINF:5.500,
test.mp4.ts?start=21.916&end=27.416&a=1&b=2
#EXTINF:15.167,
test.mp4.ts?start=27.416&end=42.583&a=1&b=2
#EXTINF:9.626,
test.mp4.ts?start=42.583&end=52.209&a=1&b=2

#EXT-X-ENDLIST
```

如果 HLS 流受到 ngx_http_secure_link_module 模块的保护，则不应在 secure_link_md5 表达式中使用 `$uri`，因为这会在请求片段时触发错误。应该使用 Base URI 而不是 `$uri`（在示例中为 `$hls_uri`）：

```
http {
    ...

    map $uri $hls_uri {
        ~^(?<base_uri>.*).m3u8$ $base_uri;
        ~^(?<base_uri>.*).ts$   $base_uri;
        default                 $uri;
    }

    server {
        ...

        location /hls {
            hls;
            hls_forward_args on;

            alias /var/videos;

            secure_link $arg_md5,$arg_expires;
            secure_link_md5 "$secure_link_expires$hls_uri$remote_addr secret";

            if ($secure_link = "") {
                return 403;
            }

            if ($secure_link = "0") {
                return 410;
            }
        }
    }
}
```

hls_fragment

-	说明
语法	**hls_fragment** `time` ;
默认	hls_fragment 5s;
上下文	http、server、location

定义未使用 `len` 参数请求的播放列表 URI 的默认片段长度。

hls_mp4_buffer_size

-	说明
语法	**hls_mp4_buffer_size** `size` ;
默认	hls_mp4_buffer_size 512k;
上下文	http、server、location

设置用于处理 MP4 和 MOV 文件的缓冲区的初始大小（ `size` ）。

hls_mp4_max_buffer_size

-	说明
语法	**hls_mp4_max_buffer_size** `size` ;
默认	hls_mp4_max_buffer_size 10m;
上下文	http、server、location

在元数据处理期间，可能需要更大的缓冲区。其大小不能超过指定的大小（ `size` ），否则 nginx 将返回 500 状态码（内部服务器错误），并记录以下消息：

```
"/some/movie/file.mp4" mp4 moov atom is too large:
12583268, you may want to increase hls_mp4_max_buffer_size
```

原文档

http://nginx.org/en/docs/http/ngx_http_hls_module.html

ngx_http_image_filter_module

`ngx_http_image_filter_module` 模块（0.7.54+）是一个可以转换 JPEG、GIF、PNG 和 WebP 格式图像的过滤器。

此模块不是默认构建的，可以使用 `--with-http_image_filter_module` 配置参数启用。

> 该模块使用了 libgd 库。建议使用该库的最新版本。
>
> WebP 格式支持出现在 1.11.6 版本中。要转换成此格式的图像，必须在编译 `libgd` 库时启用 WebP 支持。

示例配置

```
location /img/ {
    proxy_pass   http://backend;
    image_filter resize 150 100;
    image_filter rotate 90;
    error_page   415 = /empty;
}

location = /empty {
    empty_gif;
}
```

指令

image_filter

-	说明
语法	**image_filter** `off` ; **image_filter** `test` ; **image_filter** `size` ; **image_filter** `rotate 90	180	270` ; **image_filter** `resize width height` ; **image_filter** `crop width height` ;
默认	image_filter off;
上下文	location

设置图片执行的转换类型：

- `off`

  关闭对 location 模块的处理

- `test`

  确保响应是 JPEG、GIF、PNG 或 WebP 格式的图片。否则，返回 415（不支持的媒体类型）错误。

- `size`

  以 JSON 格式输出图片的信息，例如：

  ```
  { "img" : { "width": 100, "height": 100, "type": "gif" } }
  ```

  发生错误时，输出如下：

  ```
  {}
  ```

- `rotate 90|180|270`

  将图片逆时针旋转指定的度数。参数值可以包含变量。此模式可以单独使用，也可以与调整大小和裁剪转换一起使用。

- `resize width height`

  按比例将图片缩小到指定的尺寸。要只指定一个维度，可以将另一个维度指定为 `-`。当发生错误，服务器将返回 415 状态码（不支持的媒体类型）。参数值可以包含变量。当与 `rotate` 参数一起使用时，旋转变换将在缩小变换之后执行。

- `crop width height`

按比例将图片缩小到较大的一边，并裁剪另一边多余的边缘。要只指定一个维度，可以将另一个维度指定为 `-`。当发生错误，服务器将返回 415 状态码（不支持的媒体类型）。参数值可以包含变量。当与 `rotate` 参数一起使用时，旋转变换将在缩小变换之前执行。

image_filter_buffer

-	说明
语法	**image_filter_buffer** `size` ;
默认	image_filter_buffer 1M;
上下文	http、server、location

设置用于读取图片的缓冲区的最大大小。当超过指定大小时，服务器返回 415 错误状态码（不支持的媒体类型）。

image_filter_interlace

-	说明
语法	**image_filter_interlace** `on` | `off` ;
默认	image_filter_interlace off;
上下文	http、server、location
提示	该指令在 1.3.15 版本中出现

如果启用此选项，图片最后将被逐行扫描。对于 JPEG，图片最终将采用逐行 **JPEG** 格式。

image_filter_jpeg_quality

-	说明
语法	**image_filter_jpeg_quality** `quality` ;
默认	image_filter_jpeg_quality 75;
上下文	http、server、location

设置 JPEG 图片的转换质量。可接受的值范围在 1 到 100 之间。较小的值意味着较低的图片质量和较少的数据传输。最大的推荐值是 95，参数值可以包含变量。

image_filter_sharpen

-	说明
语法	**image_filter_sharpen** `percent` ;
默认	image_filter_sharpen 0;
上下文	http、server、location

增加最终图像的清晰度。锐度百分比可以超过 100。零值将禁用锐化。参数值可以包含变量。

image_filter_transparency

-	说明
语法	**image_filter_transparency** `on` | `off` ;
默认	image_filter_transparency on;
上下文	http、server、location

定义在使用调色板指定的颜色转换 GIF 图像或 PNG 图像时是否保留透明度。透明度的丧失使图像的质量更好的。PNG 中的 alpha 通道透明度始终保留。

image_filter_webp_quality

-	说明
语法	**image_filter_webp_quality** `quality` ;
默认	image_filter_webp_quality 80;
上下文	http、server、location
提示	该指令在 1.11.6 版本中出现

设置 WebP 图片的转换质量。可接受的值在 1 到 100 之间。较小的值意味着较低的图片质量和较少的数据传输。参数值可以包含变量。

原文档

http://nginx.org/en/docs/http/ngx_http_image_filter_module.html

ngx_http_mirror_module# ngx_http_index_module

`ngx_http_index_module` 模块处理以斜线字符（`/`）结尾的请求。这些请求也可以由ngx_http_autoindex_module 和 ngx_http_random_index_module 模块来处理。

示例配置

```
location / {
    index index.$geo.html index.html;
}
```

指令

index

-	说明
语法	**index** `file ...`;
默认	index index.html;
上下文	http、server、location

定义将用作索引的文件。文件名可以包含变量。以指定的顺序检查文件。列表的最后一个元素可以是一个具有绝对路径的文件。例：

```
index index.$geo.html index.0.html /index.html;
```

应该注意的是，使用索引文件发起内部重定向，可以在不同的 location 处理请求。例如，使用以下配置：

```
location = / {
    index index.html;
}

location / {
    ...
}
```

`/` 请求实际上是将在第二个 location 处理为 `/index.html` 。

原文档

http://nginx.org/en/docs/http/ngx_http_flv_module.html

ngx_http_js_module

`ngx_http_js_module` 模块用于在 nginScript 中实现 location 和变量处理器 — 它是 JavaScript 语言的一个子集。

此模块不是默构建，可以使用 `--add-module` 配置参数与 nginScript 模块一起编译：

```
./configure --add-module=path-to-njs/nginx
```

可以使用以下命令克隆 nginScript 模块仓库（需要 Mercurial 客户端）：

```
hg clone http://hg.nginx.org/njs
```

该模块也可以构建为动态形式：

```
./configure --add-dynamic-module=path-to-njs/nginx
```

示例配置

```
js_include http.js;

js_set $foo     foo;
js_set $summary summary;

server {
    listen 8000;

    location / {
        add_header X-Foo $foo;
        js_content baz;
    }

    location /summary {
        return 200 $summary;
    }
}
```

`http.js` 文件：

```
function foo(req, res) {
    req.log("hello from foo() handler");
    return "foo";
}

function summary(req, res) {
    var a, s, h;

    s = "JS summary\n\n";

    s += "Method: " + req.method + "\n";
    s += "HTTP version: " + req.httpVersion + "\n";
    s += "Host: " + req.headers.host + "\n";
    s += "Remote Address: " + req.remoteAddress + "\n";
    s += "URI: " + req.uri + "\n";

    s += "Headers:\n";
    for (h in req.headers) {
        s += "  header '" + h + "' is '" + req.headers[h] + "'\n";
    }

    s += "Args:\n";
    for (a in req.args) {
        s += "  arg '" + a + "' is '" + req.args[a] + "'\n";
    }

    return s;
}

function baz(req, res) {
    res.headers.foo = 1234;
    res.status = 200;
    res.contentType = "text/plain; charset=utf-8";
    res.contentLength = 15;
    res.sendHeader();
    res.send("nginx");
    res.send("java");
    res.send("script");

    res.finish();
}
```

指令

js_include

-	说明
语法	**js_include** `file`;
默认	——
上下文	http

指定一个在 nginScript 中实现 location 和变量处理器的文件。

hls_buffers

-	说明
语法	**js_content** `function`;
默认	——
上下文	location、limit_except

将 nginScript 函数设置为 location 内容处理器。

js_set

-	说明
语法	**js_set** `$variable function`;
默认	——
上下文	http
提示	该指令在 1.5.12 版本中出现

为指定变量设置 nginScript 函数。

请求与响应参数

每个 HTTP nginScript 处理器接收两个参数，请求和响应。

请求对象具有以下属性：

- `uri`

  请求的当前 URI，只读

- `method`

  请求方法，只读

- `httpVersion`

  HTTP 版本，只读

- `remoteAddress`

  客户端地址，只读

- `headers{}`

  请求头对象，只读

  例如，可以使用语法 `headers['Header-Name']` 或 `headers.Header_name` 来访问 `Header-Name` 头

- `args{}`

  请求参数对象，只读

- `variables{}`

  nginx 变量对象，只读

- `log(string)`

  将 `string` 写入错误日志

响应对象具有以下属性：

- `status`

  响应状态，可写

- `headers{}`

  响应头对象

- `contentType`

  响应的 `Content-Type` 头字段值，可写

- `contentLength`

  响应的 `Content-Length` 头字段值，可写

响应对象具有以下方法：

- `sendHeader()`

  将 HTTP 头发送到客户端

- `send(string)`

将部分响应体的发送给客户端

- `finish()`

  完成向客户端发送响应

原文档

http://nginx.org/en/docs/http/ngx_http_js_module.html

ngx_http_keyval_module

`ngx_http_keyval_module` 模块（1.13.3）创建的带值变量从 API 管理的键值对中获取。

> 该模块可作为我们商业订阅的一部分。

示例配置

```
http {

    keyval_zone zone=one:32k state=one.keyval;
    keyval $arg_text $text zone=one;
    ...
    server {
        ...
        location / {
            return 200 $text;
        }

        location /api {
            api write=on;
        }
    }
}
```

指令

keyval

-	说明
语法	**keyval** `key $variable zone=name` ;
默认	——
上下文	http

创建一个新的变量 `$variable`，该变量的值从键值数据库中通过 `key` 查找。字符串匹配忽略大小写。数据库存储在 `zone` 参数指定的共享内存区域中。

keyval_zone

-	说明
语法	**keyval_zone** `zone=name:size [state=file]`;
默认	——
上下文	http

设置保存键值数据库的共享内存区域的名称（ `name` ）和大小（ `size` ）。键值对由 API 管理。

可选的 `state` 参数指定一个文件，该文件将键值数据库的当前状态保持为 JSON 格式，并使其在 nginx 重启时保持不变。

原文档

http://nginx.org/en/docs/http/ngx_http_keyval_module.html

ngx_http_limit_conn_module

`ngx_http_limit_conn_module` 模块用于限制每个已定义的 key 的连接数量，特别是来自单个 IP 地址的连接数量。

并非所有的连接都会被计数。只有当服务器处理了请求并且已经读取了整个请求头时，连接才被计数。

示例配置

```
http {
    limit_conn_zone $binary_remote_addr zone=addr:10m;

    ...

    server {

        ...

        location /download/ {
            limit_conn addr 1;
        }
```

指令

limit_conn

-	说明
语法	**limit_conn** `zone number` ;
默认	——
上下文	http、server、location

设置给定键值的共享内存区域和最大允许连接数。当超过此限制时，服务器将返回错误响应请求。例如：

```
limit_conn_zone $binary_remote_addr zone=addr:10m;

server {
    location /download/ {
        limit_conn addr 1;
    }
```

同一时间只允许一个 IP 地址一个连接。

> 在 HTTP/2和 SPDY 中，每个并发请求都被视为一个单独的连接。

可以有多个 `limit_conn` 指令。例如，以下配置将限制每个客户端 IP 连接到服务器的数量，同时限制连接到虚拟服务器的总数：

```
limit_conn_zone $binary_remote_addr zone=perip:10m;
limit_conn_zone $server_name zone=perserver:10m;

server {
    ...
    limit_conn perip 10;
    limit_conn perserver 100;
}
```

当且仅当在当前级别上没有 `limit_conn` 指令时，这些指令才从前一级继承。

limit_conn_log_level

-	说明
语法	**limit_conn_log_level** `info` | `notice` | `warn` | `error` ;
默认	limit_conn_log_level error;
上下文	http、server、location
提示	该指令在 0.8.18 版本中出现

当服务器限制连接数时，设置所需的日志记录级别。

limit_conn_status

-	说明
语法	**limit_conn_status** `code` ;
默认	limit_conn_status 503;
上下文	http、server、location
提示	该指令在 1.3.15 版本中出现

设置响应拒绝请求返回的状态码。

limit_conn_zone

-	说明
语法	**limit_conn_zone** `key zone=name:size` ;
默认	——
上下文	http

为共享内存区域设置参数，该区域将保留各种键的状态。特别是，该状态包含当前的连接数。`key` 可以包含文本、变量及其组合。不包括有空键值的请求。

> 在 1.7.6 版本之前，一个 key 可能只包含一个变量。

用法示例：

```
limit_conn_zone $binary_remote_addr zone=addr:10m;
```

在这里，客户端 IP 地址作为 key。请注意，不是 `$remote_addr` ，而是使用 `$binary_remote_addr` 变量。`$remote_addr` 变量的大小可以为 7 到 15 个字节不等。存储状态在 32 位平台上占用 32 或 64 字节的内存，在 64 位平台上总是占用 64 字节。对于 IPv4 地址，`$binary_remote_addr` 变量的大小始终为 4 个字节，对于 IPv6 地址则为 16 个字节。存储状态在 32 位平台上始终占用 32 或 64 个字节，在 64 位平台上占用 64 个字节。一兆字节的区域可以保持大约 32000 个 32 字节的状态或大约 16000 个 64 字节的状态。如果区域存储耗尽，服务器会将错误返回给所有其余的请求。

limit_zone

-	说明
语法	**limit_zone** `name $variable size` ;
默认	——
上下文	http

该指令在 1.1.8 版本中已过时，并在 1.7.6 版本中被删除。请使用等效 limit_conn_zone 指令代替：

```
limit_conn_zone $variable zone=name:size;
```

原文档

http://nginx.org/en/docs/http/ngx_http_limit_conn_module.html

ngx_http_limit_req_module

`ngx_http_limit_req_module` 模块（0.7.21）用于限制每个已定义 key 的请求处理速率，特别是来自单个 IP 地址请求的处理速率。限制机制采用了 **leaky bucket** （漏桶算法）方法完成。

示例配置

```
http {
    limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;

    ...

    server {

        ...

        location /search/ {
            limit_req zone=one burst=5;
        }
```

指令

limit_req

-	说明
语法	**limit_req** `zone=name [burst=number] [nodelay]` ;
默认	——
上下文	http、server、location

设置共享内存区域和请求的最大突发大小。如果请求速率超过为某个区域配置的速率，则它们的处理会延迟，从而使请求以定义的速率处理。过多的请求被延迟，直到它们的数量超过最大突发大小，在这种情况下请求被终止并出现错误。 默认情况下，最大突发大小等于零。

例如：

```
limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;

server {
    location /search/ {
        limit_req zone=one burst=5;
    }
```

平均每秒不超过 1 个请求，并且突发不超过 5 个请求。

如果在限制期间延迟请求过多，则不需要使用参数 `nodelay`：

```
limit_req zone=one burst=5 nodelay;
```

可以存在多个 `limit_req` 指令。例如，以下配置将限制来自单个 IP 地址请求的处理速率，同时限制虚拟服务器的请求处理速率：

```
limit_req_zone $binary_remote_addr zone=perip:10m rate=1r/s;
limit_req_zone $server_name zone=perserver:10m rate=10r/s;

server {
    ...
    limit_req zone=perip burst=5 nodelay;
    limit_req zone=perserver burst=10;
}
```

当且仅当在当前级别上没有 `limit_req` 指令时，这些指令才从上一级继承。

limit_req_log_level

-	说明
语法	**limit_req_log_level** `info` | `notice` | `warn` | `error` ;
默认	limit_req_log_level error;
上下文	http、server、location
提示	该指令在 0.8.18 版本中出现

当服务器由于速率超出而拒绝处理请求或延迟请求处理时，设置所需的日志记录级别。延误情况的记录等级比拒绝情况的记录低一些。例如，如果指定了 `limit_req_log_level notice`，则延迟情况将会在 `info` 级别记录。

limit_req_status

-	说明
语法	**limit_req_status** `code` ;
默认	limit_req_status 503;
上下文	http、server、location
提示	该指令在 1.3.15 版本中出现

设置响应拒绝请求返回的状态码。

limit_req_zone

-	说明
语法	**limit_req_zone** `key zone=name:size rate=rate` ;
默认	——
上下文	http

为共享内存区域设置参数，该区域将保留各种键的状态。特别是，该状态包含当前的连接数。`key` 可以包含文本、变量及其组合。不包括有空键值的请求。

> 在 1.7.6 版本之前，一个 `key` 可能只包含一个变量。

用法示例：

```
limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
```

在这里，状态保持在 10 兆字节的区域 **one**，并且该区域的平均请求处理速率不能超过每秒 1 个请求。

客户端 IP 地址作为 key。请注意，不是 `$remote_addr` ，而是使用 `$binary_remote_addr` 变量。`$binary_remote_addr` 变量的大小始终为 4 个字节，对于 IPv6 地址则为 16 个字节。存储状态在 32 位平台上始终占用 32 或 64 个字节，在 64 位平台上占用 64 个字节。一兆字节的区域可以保持大约 32000 个 32 字节的状态或大约 16000 个 64 字节的状态或大约 8000 个 128 字节的状态。

如果区域存储耗尽，最近最少使用的状态将被删除。即使在此之后无法创建新状态，该请求也会因错误而终止。

速率以每秒请求数（r/s）指定。如果需要每秒小于一个请求的速率，则按每分钟请求（r/m）指定。例如，每秒半请求是 30r/m。

原文档

http://nginx.org/en/docs/http/ngx_http_limit_req_module.html

ngx_http_log_module

`ngx_http_log_module` 模块可让请求日志以指定的格式写入。

请求会在处理结束的 location 的上下文中记录。如果在请求处理期间发生内部重定向，可能会造成与原始 location 不同。

示例配置

```
log_format compression '$remote_addr - $remote_user [$time_local] '
                       '"$request" $status $bytes_sent '
                       '"$http_referer" "$http_user_agent" "$gzip_ratio"';

access_log /spool/logs/nginx-access.log compression buffer=32k;
```

指令

access_log

-	说明
语法	**access_log** `path [format [buffer=size] [gzip[=level]] [flush=time] [if=condition]]` ; **access_log** `off` ;
默认	access_log logs/access.log combined;
上下文	http、server、location、location 中的 if、limit_except

设置缓冲日志写入的路径、格式和配置。可以在同一级别指定多个日志。可以通过在第一个参数中指定 `syslog:` 前缀配置将日志记录到 syslog。特殊值 `off` 取消当前级别的所有 `access_log` 指令。如果未指定格式，则使用预定义的 `combined` 格式。

如果使用缓冲区或 `gzip` 参数（1.3.10、1.2.7），则日志写入将被缓冲。

> 缓冲区大小不得超过原子写入磁盘文件的大小。对于 FreeBSD，该大小是无限的。

启用缓冲时，以下情况数据将被写入文件中：

- 如果下一个日志行不适合放入缓冲区
- 如果缓冲数据比 `flush参数` 指定的更旧（1.3.10,1.2.7）
- 当 worker 进程重新打开日志文件或正在关闭时

如果使用 `gzip` 参数，那么在写入文件之前，缓冲的数据将被压缩。压缩级别可以设置在 1（最快、较少压缩）和 9（最慢、最佳压缩）之间。默认情况下，缓冲区大小等于 64K 字节，压缩级别设置为 1。由于数据是以原子块的形式压缩的，因此日志文件可以随时解压缩或由 `zcat` 读取。

示例：

```
access_log /path/to/log.gz combined gzip flush=5m;
```

> 要使 gzip 压缩起作用，必须使用 zlib 库构建 nginx。

文件路径可以包含变量（0.7.6+），但存在一些限制：

- 被 woker 进程使用凭据的用户应有在此类日志在目录中创建文件的权限
- 缓冲写入不起作用
- 该文件在每次日志写入都要打开和关闭。然而，由于常用文件的描述符可以存储在缓存中，因此可以在 open_log_file_cache 指令的 `valid` 参数指定的时间内继续写入旧文件
- 在每次写入日志期间，检查请求根目录是否存在，如果不存在，则不创建日志。因此，在同一级别指定 root 和 access_log 是一个好方法：

```
server {
    root       /spool/vhost/data/$host;
    access_log /spool/vhost/logs/$host;
    ...
```

`if` 参数（1.7.0）启用条件日志记录。如果 `condition` 为 `0` 或空字符串，则不会记录请求。在以下示例中，不会记录响应代码为 2xx 和 3xx 的请求：

```
map $status $loggable {
    ~^[23]  0;
    default 1;
}

access_log /path/to/access.log combined if=$loggable;
```

log_format

-	说明
语法	**log_format** `name [escape=default|json|none] string ...` ;
默认	log_format combined "...";
上下文	http

指定日志格式。

`escape` 参数（1.11.8）允许设置 `json` 或 `default` 字符在变量中转义，默认情况下使用 `default` 转义。 `none` 参数（1.13.10）禁用转义。

日志格式可以包含公共变量和仅在日志写入时存在的变量：

- `$bytes_sent`

  发送给客户端的字节数

- `$connection`

  连接序列号

- `$connection_requests`

  当前通过连接发出的请求数量（1.1.18）

- `$msec`

  以秒为单位的时间，日志写入时的毫秒精度

- `$pipe`

  如果请求是 pipe，则为 **p**，否则为 **.**

- `$request_length`

  请求长度（包括请求行、头部和请求体）

- `$request_time`

  以毫秒为精度的请求处理时间，以秒为单位。从客户端读取第一个字节到最后一个字节发送到客户端并写入日志过程的时间

- `$status`

  响应状态

- `$time_iso8601`

  本地时间采用 ISO 8601 标准格式

- `$time_local`

本地时间采用通用日志格式（Common Log Format）

> 在现代 nginx 版本中，变量 $status（1.3.2、1.2.2）、$bytes_sent（1.3.8、1.2.5）、$connection（1.3.8、1.2.5）、$connection_requests（1.3.8、1.2.5）、$msec（1.3.9、1.2.6）、$request_time（1.3.9、1.2.6）、$pipe（1.3.12、1.2.7）、`$request_length`（1.3.12、1.2.7）、$time_iso8601（1.3.12、1.2.7）和 $time_local（1.3.12、1.2.7）也作为公共变量。

发送到客户端的头字段前缀为 `sent_http_`，例如 `$sent_http_content_range`。

配置始终包含预定义的 `combined` 格式：

```
log_format combined '$remote_addr - $remote_user [$time_local] '
                    '"$request" $status $body_bytes_sent '
                    '"$http_referer" "$http_user_agent"';
```

open_log_file_cache

-	说明
语法	**open_log_file_cache** `max=N [inactive=time] [min_uses=N] [valid=time]`; **open_log_file_cache** `off`;
默认	open_log_file_cache off;
上下文	http、server、location

定义一个缓存，用于存储名称中包含变量的常用日志的文件描述符。该指令有以下参数：

- `max`

  设置缓存中描述符的最大数量。如果缓存变满，则最近最少使用（LRU）的描述符将被关闭

- `inactive`

  如果在此时间后缓存描述符没有被访问，则被关闭，默认为 10 秒

- `min_uses`

  在 `inactive` 参数定义的时间内设置文件使用的最小数量，以使描述符在缓存中保持打开状态，默认为 1

- `valid`

  设置检查文件是否仍然存在同名的时间，默认为 60 秒

- `off`

禁用缓存

使用示例：

```
open_log_file_cache max=1000 inactive=20s valid=1m min_uses=2;
```

原文档

http://nginx.org/en/docs/http/ngx_http_log_module.html

ngx_http_map_module

`ngx_http_map_module` 模块创建的变量的值取决于其他变量值。

示例配置

```
map $http_host $name {
    hostnames;

    default       0;

    example.com   1;
    *.example.com 1;
    example.org   2;
    *.example.org 2;
    .example.net  3;
    wap.*         4;
}

map $http_user_agent $mobile {
    default       0;
    "~Opera Mini" 1;
}
```

指令

map

-	说明
语法	**map** `string $variable { ... }`;
默认	——
上下文	http

创建一个新变量，其值取决于第一个参数中指定的一个或多个源变量的值。

> 在 0.9.0 版本之前，只能在第一个参数中指定一个变量。
>
> 由于只有在使用变量时才计算变量的值，因此仅仅声明大量的 `map` 变量也不会增加额外请求处理负荷。

`map` 块内的参数指定源和结果值之间的映射。

源值可以是字符串或正则表达式（0.9.6）。

字符串匹配将忽略大小写。

正则表达式应该以区分大小写匹配的 `~` 符号开始，或者以区分大小写匹配的 `~*` 符号开始（1.0.4）。正则表达式可以包含命名和位置捕获，之后可以将其用于其他指令和作为结果变量。

如果源值与下面描罗列的特殊参数名称之一相匹配，则应该以 `\` 符号为前缀转义。

结果值可以包含文本，变量（0.9.0）及其组合（1.11.0）。

还支持以下特殊参数：

- `default`

  如果源值不匹配指定变体，则设置结果值。如果未指定 `default` ，则默认结果值为空字符串。

- `hostname`

  表示源值可以是具有前缀或后缀掩码的主机名：

  ```
  *.example.com 1;
  example.*     1;
  ```

  以下两条记录

  ```
  example.com 1;
  * .example.com 1;
  `
  ```

  可以合并：

  ```
  .example.com 1;
  ```

  这个参数应该在值列表之前指定。

- `include file`

  包含一个包含值的文件。可以有多个包含。

- `volatile`

  表示该变量不可缓存（1.11.7）

如果源值匹配多于一个指定的变体，例如 掩码和正则表达式匹配时，将按照以下优先级顺序选择第一个匹配变体：

1. 没有掩码的字符串值
2. 带有前缀掩码的最长字符串值，例如 `*.example.com`
3. 带有后缀掩码的最长字符串值，例如 `mail.*`
4. 首先匹配正则表达式（按照在配置文件中出现的顺序）
5. 默认值

map_hash_bucket_size

-	说明
语法	**map_hash_bucket_size** `size` ;
默认	map_hash_bucket_size 32|64|128;
上下文	http

设置 map 变量哈希表的桶大小。默认值取决于处理器的缓存行大小。设置哈希表的详细内容可在单独的文档中找到。

map_hash_max_size

-	说明
语法	**map_hash_max_size** `size` ;
默认	map_hash_max_size 2048;
上下文	http

设置 map 变量哈希表的最大大小（ `size` ）。设置哈熟表的详细内容可在单独的文档中找到。

原文档

http://nginx.org/en/docs/http/ngx_http_map_module.html

ngx_http_memcached_module

`ngx_http_memcached_module` 模块用于从 memcached 服务器上获取响应。key 设置在 `$memcached_key` 变量中。应通过 nginx 之外的方式提前将响应放入 memcached。

示例配置

```
server {
    location / {
        set             $memcached_key "$uri?$args";
        memcached_pass  host:11211;
        error_page      404 502 504 = @fallback;
    }

    location @fallback {
        proxy_pass      http://backend;
    }
}
```

指令

memcached_bind

-	说明
语法	**memcached_bind** `address [transparent ]` | `off` ;
默认	——
上下文	http、server、location
提示	该指令在 0.8.22 版本中出现

连接到一个指定了本地 IP 地址和可选端口（1.11.2）的 memcached 服务器。参数值可以包含变量（1.3.12）。特殊值 `off` （1.3.12）取消从上层配置级别继承的 `memcached_bind` 指令的作用，其允许系统自动分配本地 IP 地址和端口。

`transparent` 参数（1.11.0）允许出站从非本地 IP 地址到 memcached 服务器的连接（例如，来自客户端的真实 IP 地址）：

```
memcached_bind $remote_addr transparent;
```

为了使这个参数起作用，通常需要以超级用户权限运行 nginx worker 进程。在 Linux 上，不需要指定 `transparent` 参数，工作进程会继承 master 进程的 `CAP_NET_RAW` 功能。此外，还要配置内核路由表来拦截来自 memcached 服务器的网络流量。

memcached_buffer_size

-	说明
语法	**memcached_buffer_size** `size` ;
默认	memcached_buffer_size 4k|8k
上下文	http、server、location

设置用于读取从 memcached 服务器收到的响应的缓冲区的大小（ `size` ）。一旦收到响应，响应便会同步传送给客户端。

memcached_connect_timeout

-	说明
语法	**memcached_connect_timeout** `time` ;
默认	memcached_connect_timeout 60s
上下文	http、server、location

定义与 memcached 服务器建立连接的超时时间。需要说明的是，超时通常不能超过 75 秒。

memcached_force_ranges

-	说明
语法	**memcached_force_ranges** `on` | `off` ;
默认	memcached_force_ranges off;
上下文	http、server、location
提示	该指令在 1.7.7 版本中出现

无论响应中的 **Accept-Ranges** 字段如何，都对来自 memcached 服务器的缓存和未缓存的响应启用 byte-range 支持。

memcached_gzip_flag

-	说明
语法	**memcached_gzip_flag** `flag` ;
默认	——
上下文	http、server、location
提示	该指令在 1.3.6 版本中出现

启用对 memcached 服务器响应中的 `flag` 存在测试，并在 flag 设置时将 **Content-Encoding** 响应头字段设置为 **gzip**。

memcached_next_upstream

-	说明
语法	**memcached_next_upstream** `error` | `timeout` | `invalid_response` | `not_found` | `off ...` ;
默认	memcached_next_upstream error timeout;
上下文	http、server、location

指定在哪些情况下请求应传递给下一台服务器：

- `error`

  在与服务器建立连接、传递请求或读取响应头时发生错误

- `timeout`

  在与服务器建立连接、传递请求或读取响应头时发生超时

- `invalid_response`

  服务器返回空或无效的响应

- `not_found`

  在服务器上未找到响应

- `off`

  禁用将请求传递给下一个服务器。

我们应该记住，只有在没有任何内容发送给客户端的情况下，才能将请求传递给下一个服务器。也就是说，如果在响应传输过程中发生错误或超时，修复这样的错误是不可能的。

该指令还定义了与服务器进行通信的失败尝试。`error`、`timeout` 和 `invalid_response` 的情况始终被视为失败尝试，即使它们没有在指令中指定。`not_found` 的情况永远不会被视为失败尝试。

将请求传递给下一个服务器可能受到尝试次数和时间的限制。

memcached_next_upstream_tries

-	说明
语法	**memcached_next_upstream_tries** `number` ;
默认	memcached_next_upstream_tries 0;
上下文	http、server、location
提示	该指令在 1.7.5 版本中出现

限制尝试将请求传递到下一个服务器的次数。`0` 值表示关闭此限制。

memcached_pass

-	说明
语法	**memcached_pass** `address` ;
默认	——
上下文	http、location 中 if

设置 memcached 服务器地址。该地址可以指定为域名或 IP 地址以及端口：

```
memcached_pass localhost:11211;
```

或使用 UNIX 域套接字路径：

```
memcached_pass unix:/tmp/memcached.socket;
```

如果域名解析为多个地址，则这些地址将以循环方式使用。另外，地址可以被指定为服务器组。

memcached_read_timeout

-	说明
语法	**memcached_read_timeout** `time` ;
默认	memcached_read_timeout 60s;
上下文	http、server、location

memcached 定义从 gRPC 服务器读取响应的超时时间。超时间隔只在两次连续的读操作之间，而不是整个响应的传输过程。如果 memcached 服务器在此时间内没有发送任何内容，则连接关闭。

memcached_send_timeout

-	说明
语法	**memcached_send_timeout** `time` ;
默认	memcached_send_timeout 60s;
上下文	http、server、location

设置将请求传输到 memcached 服务器的超时时间。超时间隔只在两次连续写入操作之间，而不是整个请求的传输过程。如果 memcached 服务器在此时间内没有收到任何内容，则连接将关闭。

内嵌变量

- `$memcached_key`

  定义从 memcached 服务器获取响应的密钥

原文档

http://nginx.org/en/docs/http/ngx_http_memcached_module.html

ngx_http_mirror_module

`ngx_http_mirror_module` 模块（1.13.4）通过创建后台镜像子请求来实现原始请求的镜像。镜像子请求的响应将被忽略。

示例配置

```
location / {
    mirror /mirror;
    proxy_pass http://backend;
}

location /mirror {
    internal;
    proxy_pass http://test_backend$request_uri;
}
```

指令

mirror

-	说明
语法	**mirror** `uri` | `off` ;
默认	mirror off;
上下文	http、server、location

设置将做成镜像的原始请求的 URI。可以在同一层级上指定多个镜像。

mirror_request_body

-	说明
语法	**mirror_request_body** `on` | `off` ;
默认	mirror_request_body on;
上下文	http、server、location

指示是否将客户端请求体做成镜像。启用后，将在创建镜像子请求之前读取客户端请求体。在这种情况下，将禁用由 proxy_request_buffering、fastcgi_request_buffering、scgi_request_buffering 和 uwsgi_request_buffering 指令设置的未缓冲的客户端请求正代理。

```
location / {
    mirror /mirror;
    mirror_request_body off;
    proxy_pass http://backend;
}

location /mirror {
    internal;
    proxy_pass http://log_backend;
    proxy_pass_request_body off;
    proxy_set_header Content-Length "";
    proxy_set_header X-Original-URI $request_uri;
}
```

原文档

http://nginx.org/en/docs/http/ngx_http_mirror_module.html

ngx_http_mp4_module

`ngx_http_mp4_module` 模块为 MP4 文件提供伪流服务端支持。这些文件的扩展名通常为 `.mp4` 、 `.m4v` 或 `.m4a` 。

伪流与兼容的 Flash 播放器可以很好地配合工作。播放器在查询字符串参数中指定的开始时间向服务器发送 HTTP 请求（简单地以 `start` 命名并以秒为单位），服务器以流响应方式使其起始位置与请求的时间相对应，例如：

```
http://example.com/elephants_dream.mp4?start=238.88
```

这将允许随时执行随机查找，或者在时间线中间开始回放。

为了支持搜索，基于 H.264 的格式将元数据存储在所谓的 **moov atom** 中。它是保存整个文件索引信息文件的一部分。

要开始播放，播放器首先需要读取元数据。通过发送一个有 `start=0` 参数的特殊请求来完成的。许多编码软件在文件的末尾插入元数据。这对于伪流播来说很糟糕，因为播放器必须在开始播放之前下载整个文件。如果元数据位于文件的开头，那么 nginx 就可以简单地开始发回文件内容。如果元数据位于文件末尾，nginx 必须读取整个文件并准备一个新流，以便元数据位于媒体数据之前。这涉及到一些 CPU、内存和磁盘 I/O 开销，所以最好事先准备一个用于伪流传输的原始文件，而不是让 nginx 在每个这样的请求上都这样处理。

该模块还支持设置播放结束点的 HTTP 请求（1.5.13）的 `end` 参数。 `end` 参数可以与 `start` 参数一起指定或单独指定：

```
http://example.com/elephants_dream.mp4?start=238.88&end=555.55
```

对于有非零 `start` 或 `end` 参数的匹配请求，nginx 将从文件中读取元数据，准备有所需时间范围的流并将其发送到客户端 这与上面描述的开销相同。

如果匹配请求不包含 `start` 和 `end` 参数，则不会有开销，并且文件仅作为静态资源发送。有些播放器也支持 byte-range 请求，因此不需要这个模块。

该模块不是默认构建的，可以使用 `--with-http_mp4_module` 配置参数启用。

> 如果以前使用过第三方 mp4 模块，则应该禁用它。

ngx_http_flv_module 模块提供了对 FLV 文件的类伪流式的支持。

示例配置

```
location /video/ {
    mp4;
    mp4_buffer_size       1m;
    mp4_max_buffer_size   5m;
    mp4_limit_rate        on;
    mp4_limit_rate_after  30s;
}
```

指令

mp4

-	说明
语法	**mp4**;
默认	——
上下文	location

启用对 location 模块处理。

mp4_buffer_size

-	说明
语法	**mp4_buffer_size** `size` ;
默认	mp4_buffer_size 512K;
上下文	http、server、location

设置用于处理 MP4 文件的缓冲区的初始大小。

mp4_max_buffer_size

-	说明
语法	**mp4_max_buffer_size** `time` ;
默认	mp4_max_buffer_size 10M;
上下文	http、server、location

在元数据处理期间，可能需要更大的缓冲区。它的大小不能超过指定的大小，否则 nginx 将返回 500（内部服务器错误）错误状态码，并记录以下消息：

```
"/some/movie/file.mp4" mp4 moov atom is too large:
12583268, you may want to increase mp4_max_buffer_size
```

mp4_limit_rate

-	说明
语法	**mp4_limit_rate** `on` | `off` | `factor` ;
默认	p4_limit_rate off;
上下文	http、server、location

限制对客户响应的传输速率。速率限制基于所提供 MP4 文件的平均比特率。要计算速率，比特率将乘以指定的 `factor` 。特殊值 `on` 对应于因子 1.1 。特殊值 `off` 禁用速率限制。限制是根据请求设置的，所以如果客户端同时打开两个连接，总体速率将是指定限制的两倍。

> 该指令可作为我们商业订阅的一部分。

mp4_limit_rate_after

-	说明
语法	**mp4_limit_rate_after** `time` ;
默认	mp4_limit_rate_after 60s;
上下文	http、server、location

设置媒体数据的初始数量（在回放时计算），之后进一步传输到客户端的响应将受到速率限制。

> 该指令可作为我们商业订阅的一部分。

原文档

http://nginx.org/en/docs/http/ngx_http_mp4_module.html

ngx_http_perl_module

`ngx_http_perl_module` 模块用于在 Perl 中实现 location 和变量处理器，并将 Perl 调用插入到 SSI中。

此模块不是默认构建，可以在构建时使用 `--with-http_perl_module` 配置参数启用。

> 该模块需要 Perl 5.6.1 或更高版本。C 编译器应该与用于构建Perl 的编译器兼容。

已知的问题

该模块还处于实验阶段，以下是一些注意事项。

为了让 Perl 能在重新配置过程中重新编译已修改的模块，应使用 `-Dusemultiplicity=yes` 或 `-Dusethreads=yes` 参数来构建它。另外，为了让 Perl 在运行时泄漏更少的内存，应使用 `-Dusemymalloc=no` 参数来构建它。要检查已构建的 Perl 中这些参数值（在示例中已指定首选值），请运行：

```
$ perl -V:usemultiplicity -V:usemymalloc
usemultiplicity='define';
usemymalloc='n';
```

请注意，在使用新的 `-Dusemultiplicity=yes` 或 `-Dusethreads=yes` 参数重新构建 Perl 之后，所有二进制 Perl 模块也必须重新构建 — 否则它们将停止使用新的 Perl。

在每次重新配置后，master 进程和 worker 进程都有可能增加。如果 master 进程增加到不可接受的大小，则可以使用实时升级流程而无需更改可执行文件。

当 Perl 模块执行长时间运行的操作时，例如解析域名、连接到另一台服务器或查询数据库时，将不会处理分配给当前 worker 进程的其他请求。因此，建议仅执行可预测且执行时间短的操作，例如访问本地文件系统。

示例配置

```
http {

    perl_modules perl/lib;
    perl_require hello.pm;

    perl_set $msie6 '

        sub {
            my $r = shift;
            my $ua = $r->header_in("User-Agent");

            return "" if $ua =~ /Opera/;
            return "1" if $ua =~ / MSIE [6-9]\.\d+/;
            return "";
        }

    ';

    server {
        location / {
            perl hello::handler;
        }
    }
```

The perl/lib/hello.pm module:

```
package hello;

use nginx;

sub handler {
    my $r = shift;

    $r->send_http_header("text/html");
    return OK if $r->header_only;

    $r->print("hello!\n<br/>");

    if (-f $r->filename or -d _) {
        $r->print($r->uri, " exists!\n");
    }

    return OK;
}

1;
__END__
```

指令

perl

-	说明
语法	**perl** `module::function` | `'sub { ... }'`;
默认	——
上下文	location、limit_except

给指定的 location 设置一个 Perl 处理程序。

perl_modules

-	说明
语法	**perl_modules** `path`;
默认	——
上下文	http

为 Perl 模块设置额外的路径。

perl_require

-	说明
语法	**perl_require** `module`;
默认	——
上下文	http

定义每次重新配置期间将要加载的模块的名称。可存在多个 `perl_require` 指令。

perl_set

-	说明
语法	**perl_set** `$variable module::function` | `'sub { ... }'`;
默认	——
上下文	http

为指定的变量安装一个 Perl 处理程序。

从 SSI 调用 Perl

使用 SSI 命令调用 Perl 的格式如下：

```
<!--# perl sub="module::function" arg="parameter1" arg="parameter2" ...
-->
```

$r 请求对象方法

- `$r->args`

  返回请求参数。

- `$r->filename`

  返回与请求 URI 相对应的文件名。

- `$r->has_request_body(handler)`

  如果请求中没有请求体，则返回 0。如果存在，则为请求设置指定的处理程序，并返回 1。在读取请求体后，nginx 将调用指定的处理程序。请注意，处理函数应该通过引用传递。例：

```
package hello;

use nginx;

sub handler {
    my $r = shift;

    if ($r->request_method ne "POST") {
        return DECLINED;
    }

    if ($r->has_request_body(\&post)) {
        return OK;
    }

    return HTTP_BAD_REQUEST;
}

sub post {
    my $r = shift;

    $r->send_http_header;

    $r->print("request_body: \"", $r->request_body, "\"<br/>");
    $r->print("request_body_file: \"", $r->request_body_file, "\"<br/>\n");

    return OK;
}

1;

__END__
```

- `$r->allow_ranges`

  在发送响应时启用字节范围。

- `$r->discard_request_body`

  指示 nginx 放弃请求体。

- `$r->header_in(field)`

  返回指定的客户端请求头字段的值。

- `$r->header_only`

  确定整个响应还是仅将头部发送给客户端。

- `$r->header_out(field, value)`

  为指定的响应头字段设置一个值。

- `$r->internal_redirect(uri)`

  做一个内部重定向到指定的 uri。在 Perl 处理程序执行完成后重定向。

  > 目前不支持重定向到具名 location。

- `$r->log_error(errno, message)`

  将指定的消息写入 error_log。如果 `errno` 不为零，则错误码及其描述将被附加到消息中。

- `$r->print(text, ...)`

  将数据传递给客户端。

- `$r->request_body`

  如果尚未将请求体写入临时文件中，则返回客户端请求体。为了确保客户端请求体在内存中，其大小应该由 client_max_body_size 来限制，并且应该使用 client_body_buffer_size 来设置足够的缓冲区大小。

- `$r->request_body_file`

  客户端请求体返回文件的名称。处理完成后，该文件将被删除。要始终将请求体写入文件，应启用 client_body_in_file_only。

- `$r->request_method`

  返回客户端请求的 HTTP 方法。

- `$r->remote_addr`

  返回客户端 IP 地址。

- `$r->flush`

  立即向客户端发送数据。

- `$r->sendfile(name[, offset[, length]])` 将指定的文件内容发送到客户端。可选参数指定要传输的数据的初始偏移量（ `offset` ）和长度（ `length` ）。数据在 Perl 处理程序完成之后开始传输。

- `$r->send_http_header([type])`

  将响应头发送给客户端。可选的 `type` 参数用于设置 **Content-Type** 响应头字段的值。如果该值为空字符串，则不会发送 **Content-Type** 头字段。

- `$r->status(code)`

  设置响应状态码。

- `$r->sleep(milliseconds, handler)`

  设置指定的处理程序（`handler`）和指定停止请求处理的时间（`milliseconds`）。在此期间，nginx 继续处理其他请求。在经过指定的时间后，nginx 将调用已安装的处理程序。请注意，处理函数应该通过引用传递。为了在处理程序之间传递数据，应使用 `$r->variable()`。例：

  ```
  package hello;

  use nginx;

  sub handler {
      my $r = shift;

      $r->discard_request_body;
      $r->variable("var", "OK");
      $r->sleep(1000, \&next);

      return OK;
  }

  sub next {
      my $r = shift;

      $r->send_http_header;
      $r->print($r->variable("var"));

      return OK;
  }

  1;

  __END__
  ```

- `$r->unescape(text)`

  解码 %**XX** 格式编码的文本。

- `$r->uri`

  返回请求 URI。

- `$r->variable(name[, value])`

  返回或设置指定变量的值。对于每个请求来说这些变量都是本地变量。

原文档

http://nginx.org/en/docs/http/ngx_http_perl_module.html

ngx_http_perl_module

`ngx_http_proxy_module` 模块允许将请求传递给另一台服务器。

示例配置

```
location / {
    proxy_pass       http://localhost:8000;
    proxy_set_header Host      $host;
    proxy_set_header X-Real-IP $remote_addr;
}
```

指令

proxy_bind

-	说明
语法	**proxy_bind** `address [transparent]` | `off` ;
默认	——
上下文	http、server、location
提示	该指令在 0.8.22 版本中出现

连接到一个指定了本地 IP 地址和可选端口（1.11.2）的代理服务器。参数值可以包含变量（1.3.12）。特殊值 `off` （1.3.12）取消从上层配置级别继承的 `proxy_bind` 指令的作用，其允许系统自动分配本地 IP 地址和端口。

`transparent` 参数（1.11.0）允许出站从非本地 IP 地址到代理服务器的连接（例如，来自客户端的真实 IP 地址）：

```
proxy_bind $remote_addr transparent;
```

为了使这个参数起作用，通常需要以超级用户权限运行 nginx worker 进程。在 Linux 上，不需要指定 `transparent` 参数（1.13.8），工作进程会继承 master 进程的 `CAP_NET_RAW` 功能。此外，还要配置内核路由表来拦截来自代理服务器的网络流量。

proxy_buffer_size

-	说明
语法	**proxy_buffer_size** `size` ;
默认	proxy_buffer_size 4k|8k;
上下文	http、server、location

设置用于读取从代理服务器收到的第一部分响应的缓冲区大小（ `size` ）。这部分通常包含一个小的响应头。默认情况下，缓冲区大小等于一个内存页。4K 或 8K，因平台而异。但是，它可以设置得更小。

proxy_buffering

-	说明
语法	**proxy_buffering** `on` | `off` ;
默认	proxy_buffering on;
上下文	http、server、location

启用或禁用来自代理服务器的响应缓冲。

当启用缓冲时，nginx 会尽可能快地收到接收来自代理服务器的响应，并将其保存到由 proxy_buffer_size 和 proxy_buffers 指令设置的缓冲区中。如果内存放不下整个响应，响应的一部分可以保存到磁盘上的临时文件中。写入临时文件由 proxy_max_temp_file_size 和 proxy_temp_file_write_size 指令控制。

当缓冲被禁用时，nginx 在收到响应时立即同步传递给客户端，不会尝试从代理服务器读取整个响应。nginx 一次可以从服务器接收的最大数据量由 proxy_buffer_size 指令设置。

通过在 `X-Accel-Buffering` 响应头字段中通过 `yes` 或 `no` 也可以启用或禁用缓冲。可以使用 proxy_ignore_headers 指令禁用此功能。

proxy_buffers

-	说明
语法	**proxy_buffers** `number size` ;
默认	proxy_buffers 8 4k|8k;
上下文	http、server、location

设置单个连接从代理服务器读取响应的缓冲区的 `number` （数量）和 `size` （大小）。默认情况下，缓冲区大小等于一个内存页。为 4K 或 8K，因平台而异。

proxy_busy_buffers_size

-	说明
语法	**proxy_busy_buffers_size** `size` ;
默认	proxy_buffer_size 8k|16k;
上下文	http、server、location

当启用代理服务器响应缓冲时，限制缓冲区的总大小（ `size` ）在当响应尚未被完全读取时可向客户端发送响应。同时，其余的缓冲区可以用来读取响应，如果需要的话，缓冲部分响应到临时文件中。默认情况下， `size` 受 proxy_buffer_size 和 proxy_buffers 指令设置的两个

缓冲区的大小限制。

proxy_cache

-	说明
语法	**proxy_cache** `zone` | `off` ;
默认	proxy_cache off;
上下文	http、server、location

定义用于缓存的共享内存区域。同一个区域可以在几个地方使用。参数值可以包含变量（1.7.9）。`off` 参数将禁用从上级配置级别继承的缓存配置。

proxy_cache_background_update

-	说明
语法	**proxy_cache_background_update** `on` | `off` ;
默认	proxy_cache_background_update off;
上下文	http、server、location
提示	该指令在 1.11.10 版本中出现

允许启动后台子请求来更新过期的缓存项，而过时的缓存响应则返回给客户端。请注意，有必要在更新时允许使用陈旧的缓存响应。

proxy_cache_bypass

-	说明
语法	**proxy_cache_bypass** `string ...` ;
默认	——
上下文	http、server、location

定义不从缓存中获取响应的条件。如果字符串参数中有一个值不为空且不等于 `0`，则不会从缓存中获取响应：

```
proxy_cache_bypass $cookie_nocache $arg_nocache$arg_comment;
proxy_cache_bypass $http_pragma    $http_authorization;
```

可以与 proxy_no_cache 指令一起使用。

proxy_cache_convert_head

-	说明
语法	**proxy_cache_convert_head** `on` | `off` ;
默认	proxy_cache_convert_head on;
上下文	http、server、location
提示	该指令在 1.9.7 版本中出现

启用或禁用将 **HEAD** 方法转换为 **GET** 进行缓存。禁用转换时，应将缓存键配置为包含 `$request_method` 。

proxy_cache_key

-	说明
语法	**proxy_cache_key** `string` ;
默认	proxy_cache_key $scheme$proxy_host$request_uri;
上下文	http、server、location

为缓存定义一个 key，例如：

```
proxy_cache_key "$host$request_uri $cookie_user";
```

默认情况下，指令的值与字符串相近：

```
proxy_cache_key $scheme$proxy_host$uri$is_args$args;
```

待续.....

原文档

http://nginx.org/en/docs/http/ngx_http_proxy_module.html

ngx_http_random_index_module

`ngx_http_random_index_module` 模块处理以 `/` 结尾的请求，然后随机选择目录中的一个文件作为索引文件展示，该模块优先于 ngx_http_index_module 之前处理。

该模块默认不会被构建到 nginx 中，需要在编译时加入 `--with-http_random_index_module` 配置参数启用。

配置示例

```
location / {
    random_index on;
}
```

random_index

-	说明
语法	**random_index** `on` | `off` ;
默认	random_index off;
上下文	location

启用或禁用 `location` 周边的模块处理。

原文档

- http://nginx.org/en/docs/http/ngx_http_random_index_module.html

ngx_mail_imap_module

指令

imap_auth

-	说明
语法	**imap_auth** `method ...` ;
默认	imap_auth plain;
上下文	mail、server

为 IMAP 客户端设置允许的认证方法。支持的方法有：

- `login`

  AUTH=LOGIN

- `plain`

  AUTH=PLAIN

- `cram-md5`

  AUTH=CRAM-MD5。为了使此方法正常工作，密码不能加密存储。

- `external`

  AUTH EXTERNAL（1.11.6）。

imap_capabilities

-	说明
语法	**imap_capabilities** `extension ...` ;
默认	imap_capabilities IMAP4 IMAP4rev1 UIDPLUS;
上下文	mail、server

设置响应 `CAPABILITY` 命令传递给客户端的 IMAP 协议 扩展列表。根据 starttls 指令值，imap_auth 指令和 STARTTLS 中指定的认证方法将自动添加到此列表中。

指定被代理客户端的 IMAP 后端支持的扩展（当 nginx 透明地代理到后端的客户端连接，如果这些扩展与认证后使用的命令相关），则是有意义的。

目前的标准扩展名单已发布在 www.iana.org。

imap_client_buffer

-	说明	
语法	**imap_client_buffer** `size` ;	
默认	imap_client_buffer 4k	8k;
上下文	mail、server	

设置 IMAP 命令读取缓冲区大小。默认情况下，缓冲区大小等于一个内存页面。4K 或 8K，取决于平台。

原文档

http://nginx.org/en/docs/mail/ngx_mail_imap_module.html

ngx_mail_pop3_module

指令

pop3_auth

-	说明
语法	**pop3_auth** `method ...` ;
默认	pop3_auth plain;
上下文	mail、server

为 POP3 客户端设置允许的认证方法。支持的方法有：

- `plain`

  USER/PASS、AUTH PLAIN、AUTH LOGIN。不可能禁用这些方法。

- `apop`

  APOP。为了使此方法正常工作，密码不能加密存储。

- `cram-md5`

  AUTH CRAM-MD5。为了使此方法正常工作，密码不能加密存储。

- `external`

  AUTH EXTERNAL（1.11.6）。

pop3_capabilities

-	说明
语法	**pop3_capabilities** `extension ...` ;
默认	pop3_capabilities TOP USER UIDL;
上下文	mail、server

设置响应 `CAPA` 命令传送给客户端的 POP3 协议 扩展列表。根据 starttls 指令值，pop3_auth 指令（SASL 扩展）和 STLS 中指定的认证方法将自动添加到此列表。

指定客户端代理的 POP3 后端支持的扩展（当 nginx 透明地代理到后端的客户端连接，如果这些扩展与认证后使用的命令相关），则是有意义的。

原文档

http://nginx.org/en/docs/mail/ngx_mail_pop3_module.html

ngx_mail_smtp_module

指令

smtp_auth

-	说明
语法	**smtp_auth** `method ...` ;
默认	smtp_auth login plain;
上下文	mail、server

为 SMTP 客户端设置 SASL 认证 的允许方法。支持的方法有：

- `login`

  AUTH LOGIN

- `plain`

  AUTH PLAIN

- `cram-md5`

  AUTH CRAM-MD5。为了使此方法正常工作，密码不加密存储。

- `external`

  AUTH EXTERNAL（1.11.6）。

- `none`

  不需要验证

smtp_capabilities

-	说明
语法	**smtp_capabilities** `extension ...`;
默认	——
上下文	mail、server

设置传送给客户端响应 `EHLO` 命令的 SMTP 协议扩展列表。根据 starttls 指令值，smtp_auth 指令和 STARTTLS 中指定的认证方法将自动添加到此列表中。

指定被代理客户端的 MTA 支持扩展是有意义的（当 nginx 透明地将客户端连接代理到后端，如果这些扩展与认证后使用的命令相关）。

目前的标准扩展名单已发布在 www.iana.org。

原文档

http://nginx.org/en/docs/mail/ngx_mail_smtp_module.html

ngx_stream_realip_module

`ngx_stream_realip_module` 模块用于将客户端地址和端口更改为 PROXY 协议头（1.11.4）中发送。必须先在 `listen` 指令中设置 proxy_protocol 参数才能启用 PROXY 协议。

该模块不是默认构建的，您可以在构建时使用 `--with-stream_realip_module` 配置参数启用。

示例配置

```
listen 12345 proxy_protocol;

set_real_ip_from  192.168.1.0/24;
set_real_ip_from  192.168.2.1;
set_real_ip_from  2001:0db8::/32;
```

指令

set_real_ip_from

-	说明		
语法	**set_real_ip_from** `address` \	`CIDR` \	`unix:` ;
默认	——		
上下文	stream、server		

定义已知可发送正确替换地址的受信任地址。如果指定了特殊值 `unix:`，则所有 UNIX 域套接字将被信任。

内嵌变量

- `$realip_remote_addr`

  保留原始客户端地址

- `$realip_remote_port`

  保留原始的客户端端口

原文档

http://nginx.org/en/docs/stream/ngx_stream_realip_module.html

ngx_stream_return_module

`ngx_stream_return_module` 模块（1.11.2）允许向客户端发送指定的值，然后关闭连接。

示例配置

```
server {
    listen 12345;
    return $time_iso8601;
}
```

指令

return

-	说明
语法	**return** `value` ;
默认	——
上下文	server

指定要发送给客户端的值。该值可以包含文本、变量及其组合。

原文档

http://nginx.org/en/docs/stream/ngx_stream_return_module.html

ngx_stream_split_clients_module

`ngx_stream_split_clients_module` 模块（1.11.3）创建适用于 A/B 测试的变量，也称为拆分测试。

示例配置

```
stream {
    ...
    split_clients "${remote_addr}AAA" $upstream {
                  0.5%                feature_test1;
                  2.0%                feature_test2;
                  *                   production;
    }

    server {
        ...
        proxy_pass $upstream;
    }
}
```

指令

split_clients

-	说明
语法	**split_clients** `string $variable { ... }`
默认	——
上下文	stream

为 A/B 测试创建一个变量，例如：

```
split_clients "${remote_addr}AAA" $variant {
               0.5%               .one;
               2.0%               .two;
               *                  "";
}
```

使用 MurmurHash2 对原始字符串的值进行哈希处理。在给出的例子中，从 0 到 21474835（0.5%）的哈希值对应于 `$variant` 变量的值 `.one`，从 21474836 到 107374180（2%）的哈希值对应于值 `.two`，哈希值从 107374181 到 4294967295 对应于值 `""`（空字符串）。

原文档

http://nginx.org/en/docs/stream/ngx_stream_split_clients_module.html

ngx_stream_ssl_preread_module

`ngx_stream_ssl_preread_module` 模块（1.11.5）允许从 ClientHello 消息中提取信息，而不会终止 SSL/TLS，例如提取通过 SNI 请求的服务器名称。默认情况下不构建此模块，您可以在构建时使用 `--with-stream_ssl_preread_module` 配置参数启用此模块。

示例配置

```
map $ssl_preread_server_name $name {
    backend.example.com      backend;
    default                  backend2;
}

upstream backend {
    server 192.168.0.1:12345;
    server 192.168.0.2:12345;
}

upstream backend2 {
    server 192.168.0.3:12345;
    server 192.168.0.4:12345;
}

server {
    listen      12346;
    proxy_pass  $name;
    ssl_preread on;
}
```

指令

google_perftools_profiles

-	说明	
语法	**ssl_preread** `on` \	`off` ;
默认	ssl_preread off;	
上下文	stream、server	

启用在预读阶段从 ClientHello 消息中提取信息。

内嵌变量

- `$ssl_preread_server_name`

  返回通过 SNI 请求的服务器名称

原文档

http://nginx.org/en/docs/ngx_google_perftools_module.html

ngx_google_perftools_module

`ngx_google_perftoos_module` 模块（0.6.29）可以使用 Google 性能工具 对 nginx 的 worker 进程进行分析。该模块适用于 nginx 开发人员。

默认情况下不构建此模块，您可以在构建时使用 `--with-google_perftools_module` 配置参数启用此模块。

> 该模块需要 gperftools 库。

示例配置

```
google_perftools_profiles /path/to/profile;
```

profile 文件将被存储为 `/path/to/profile.<worker_pid>` 。

指令

google_perftools_profiles

-	说明
语法	**google_perftools_profiles** `file ...` ;
默认	——
上下文	main

设置保存 nginx worker 进程分析信息的文件的名名。worker 进程的 ID 始终是文件名的一部分，并追加在文件名的末尾。

原文档

http://nginx.org/en/docs/ngx_google_perftools_module.html