Nginx 教程

Nginx 介绍

Nginx[读音：engine x]是 HTTP 和反向代理服务器，邮件代理服务器，以及 Igor Sysoev 最初编写的通用 TCP/UDP 代理服务器。在很长一段时间以来，它一直在许多负载重的俄罗斯网站上运行，包括：Yandex，Mail.Ru，VK 和 Rambler。 根据 Netcraft 的说法，Nginx 在 2017 年 3 月份服务或代理了 28.50％的最繁忙的网站。这里有一些成功案例：Netflix，Wordpress.com，FastMail.FM。

基本的 HTTP 服务器功能

- 提供静态和索引文件，自动索引；打开文件描述符缓存;
- 加速反向代理与缓存；负载均衡和容错;
- 通过缓存 FastCGI，uwsgi，SCGI 和 memcached 服务器来加速支持；负载均衡和容错;
- 模块化架构。 过滤器包括 gzip，字节范围，分块响应，XSLT，SSI 和图像变换过滤器。如果由代理或 FastCGI/uwsgi/SCGI 服务器处理，则单页内的多个 SSI 包含可以并行处理;
- SSL 和 TLS SNI 支持;
- 支持具有加权和依赖关系优先级的 HTTP/2。

其他 HTTP 服务器功能

- 基于名称和基于 IP 的虚拟服务器;
- 保持活动和管道连接的支持;
- 访问日志格式，缓冲日志写入，快速日志轮换和 syslog 日志记录;
- 3xx-5xx 错误代码重定向;
- 重写模块：使用正则表达式更改 URI;
- 根据客户端地址执行不同的功能;
- 根据客户端 IP 地址，密码(HTTP Basic 认证)和子请求结果进行访问控制;
- HTTP 引用的验证
- PUT，DELETE，MKCOL，COPY 和 MOVE 方法;
- FLV 和 MP4 流媒体;
- 响应速度限制;
- 限制来自一个地址的同时连接或请求的数量;
- 基于 IP 的地理定位;
- A/B 测试;

- 嵌入式 Perl
- nginScript。

邮件代理服务器功能

- 使用外部 HTTP 认证服务器将用户重定向到 IMAP 或 POP3 服务器;
- 使用外部 HTTP 认证服务器进行用户认证，并将连接重定向到内部 SMTP 服务器;
- 认证方式：
- POP3：USER / PASS，APOP，AUTH LOGIN / PLAIN / CRAM-MD5;
- IMAP：LOGIN，AUTH LOGIN / PLAIN / CRAM-MD5;
- SMTP：AUTH LOGIN / PLAIN / CRAM-MD5;
- SSL 支持;
- STARTTLS 和 STLS 支持。

TCP/UDP 代理服务器功能

- TCP 和 UDP 的通用代理;
- SSL 和 TLS SNI 支持 TCP;
- 负载均衡和容错;
- 基于客户地址的访问控制;
- 根据客户端地址执行不同的功能;
- 限制来自一个地址的同时连接数;
- 访问日志格式，缓冲日志写入，快速日志轮换和 syslog 日志记录;
- 基于 IP 的地理定位;
- A/B 测试;
- nginScript。

架构和可扩展性

- 一个主和几个工作进程; 工作进程在非特权用户下运行;
- 灵活配置;
- 重新配置和升级可执行文件，而不会中断客户端服务;
- 支持 kqueue(FreeBSD 4.1+)，epoll(Linux 2.6+)，/ dev / poll(Solaris 7 11/99 +)，事件端口(Solaris 10)，select 和 poll;
- 支持各种 kqueue 功能，包括 EV_CLEAR，EV_DISABLE(临时禁用事件)，NOTE_LOWAT，EV_EOF，可用数据数，错误代码;
- 支持各种 epoll 功能，包括 EPOLLRDHUP(Linux 2.6.17+，glibc 2.8+)和 EPOLLEXCLUSIVE(Linux 4.5+，glibc 2.24+);

- 支持 sendfile(FreeBSD 3.1+， Linux 2.2+， macOS 10.5+)， sendfile64(Linux 2.4.21+)和 sendfilev(Solaris 8 7/01 +);
- 文件 AIO(FreeBSD 4.3+，Linux 2.6.22+);
- DIRECTIO(FreeBSD 4.4+，Linux 2.4+，Solaris 2.6+，macOS);
- 接受过滤器(FreeBSD 4.1+，NetBSD 5.0+)和 TCP_DEFER_ACCEPT(Linux 2.4+)支持;
- 10,000 个不活动的 HTTP 保持连接大约需要 2.5M 内存;
- 数据复制操作保持最小。

经测试的操作系统和平台

- FreeBSD 3 — 11 / i386; FreeBSD 5 — 11 / amd64;
- Linux 2.2 — 4 / i386; Linux 2.6 — 4 / amd64; Linux 3 — 4 / armv6l, armv7l, aarch64, ppc64le;
- Solaris 9 / i386, sun4u; Solaris 10 / i386, amd64, sun4v;
- AIX 7.1 / powerpc;
- HP-UX 11.31 / ia64;
- macOS / ppc, i386;
- Windows XP, Windows Server 2003.

Nginx 特性

NGINX 有什么不同？ NGINX 使用可扩展的事件驱动架构，而不是更传统的过程驱动架构。这需要更低的内存占用，并且当并发连接扩大时，使内存使用更可预测。
在传统的 Web 服务器体系结构中，每个客户端连接作为一个单独的进程或线程处理，随着网站的流行度增加，并发连接数量的增加，Web 服务器减慢，延迟了对用户的响应。
从技术的角度来看，产生一个单独的进程/线程需要将 CPU 切换到新的任务并创建一个新的运行时上下文，消耗额外的内存和 CPU 时间，从而对性能产生负面影响。
NGINX 开发的目标是实现 10 倍以上的性能，优化服务器资源的使用，同时也能够扩展和支持网站的动态增长。 因此，NGINX 成为最知名的模块化，事件驱动，异步，单线程 Web 服务器和 Web 代理之一。

Nginx 是一个高性能的 Web 和反向代理服务器, 它具有很多非常优越的特性:

作为 Web 服务器

相比 Apache，Nginx 使用更少的资源，支持更多的并发连接，体现更高的效率，这点使 Nginx 尤其受到虚拟主机提供商的欢迎。能够支持高达 50,000 个并发连接数的响应，感谢 Nginx 为我们选择了 epoll and kqueue 作为开发模型.

作为负载均衡服务器

Nginx 既可以在内部直接支持 Rails 和 PHP，也可以支持作为 HTTP 代理服务器 对外进行服务。Nginx 用 C 编写, 不论是系统资源开销还是 CPU 使用效率都比 Perlbal 要好的多。

作为邮件代理服务器

Nginx 同时也是一个非常优秀的邮件代理服务器(最早开发这个产品的目的之一也是作为邮件代理服务器)，Last.fm 描述了成功并且美妙的使用经验。
Nginx 安装非常的简单，配置文件 非常简洁(还能够支持 perl 语法)，Bugs 非常少的服务器: Nginx 启动特别容易，并且几乎可以做到 7*24 不间断运行，即使运行数个月也不需要重新启动。你还能够在 不间断服务的情况下进行软件版本的升级。

Nginx 架构

处理并发连接的传统的基于进程或线程的模型涉及使用单独的进程或线程处理每个连接，并阻止网络或输入/输出操作。 根据应用，在内存和 CPU 消耗方面可能非常低效。 产生一个单独的进程或线程需要准备一个新的运行时环境，包括分配堆和堆栈内存，以及创建新的执行上下文。 额外的 CPU 时间也用于创建这些项目，这可能会导致由于线程在过多的上下文切换上的转机而导致性能下降。 所有这些并发症都表现在较老的 Web 服务器架构(如 Apache)中。 这是提供丰富的一般应用功能和优化的服务器资源使用之间的一个折衷。

从一开始 nginx 就是一个专门的工具，可以实现更高性能，更密集和经济地使用服务器资源，同时实现网站的动态发展，所以它采用了不同的模式。 它实际上受到各种操作系统中高级事件机制的不断发展的启发。发展结果变成是一个模块化的，事件驱动的，异步的，单线程的非阻塞架构的 nginx 代码基础。

nginx 大量使用复用和事件通知，并专门用于分离进程的特定任务。 连接在有限数量的单线程进程称为工作(worker)的高效运行循环中处理。 在每个工作(worker)中，nginx 可以处理每秒数千个并发连接和请求。

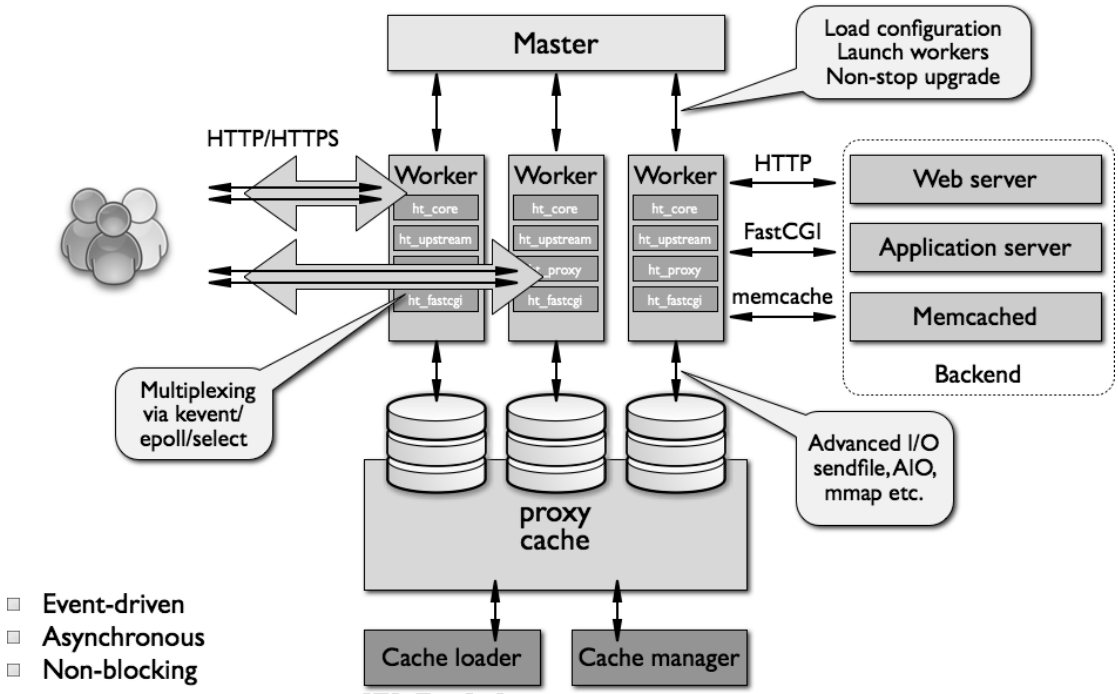
代码结构

nginx 工作(worker)码包括核心和功能模块。 nginx 的核心是负责维护严格的运行循环，并在请求处理的每个阶段执行模块代码的适当部分。 模块构成了大部分的演示和应用层功能。 模块读取和写入网络和存储，转换内容，执行出站过滤，应用服务器端包含操作，并在代理启动时将请求传递给上游服务器。

nginx 的模块化架构通常允许开发人员扩展一组 Web 服务器功能，而无需修改 nginx 内核。 nginx 模块略有不同，即核心模块，事件模块，阶段处理程序，协议，可变处理程序，

过滤器，上游和负载平衡器。nginx 不支持动态加载的模块; 即在构建阶段将模块与核心一起编译。

在处理与接受，处理和管理网络连接和内容检索相关的各种操作时，nginx 在基于 Linux，Solaris 和 BSD 的操作系统中使用事件通知机制和一些磁盘 I/O 性能增强，如：kqueue，epoll，和事件端口。 目标是为操作系统提供尽可能多的提示，以便及时获取入站和出站流量，磁盘操作，读取或写入套接字，超时等异步反馈。 对于每个基于 Unix 的 nginx 运行的操作系统，大量优化了复用和高级 I/O 操作的不同方法的使用。



工作模式

如前所述，nginx 不会为每个连接生成一个进程或线程。 相反，工作(worker)进程接受来自共享“listen”套接字的新请求，并在每个工作(worker)内执行高效的运行循环，以处理每个工作(worker)中的数千个连接。 没有专门的仲裁或分配与 nginx 工作(worker)的联系; 这个工作(worker)是由操作系统内核机制完成的。 启动后，将创建一组初始侦听套接字。 然后，工作(worker)在处理 HTTP 请求和响应时不断接受，读取和写入套接字。

运行循环是 nginx 工作(worker)代码中最复杂的部分。 它包括全面的内部调用，并且在很大程度上依赖异步任务处理的想法。 异步操作通过模块化，事件通知，广泛使用回调函数和微调定时器来实现。 总体而言，关键原则是尽可能不阻塞。 nginx 仍然可以阻塞的唯一情况是工作(worker)进程没有足够的磁盘存储。

由于 nginx 不会连接一个进程或线程，所以在绝大多数情况下，内存使用非常保守，非常有效。 nginx 也节省 CPU 周期，因为进程或线程没有持续的创建 - 销毁模式。 nginx 的作用是检查网络和存储的状态，初始化新连接，将其添加到运行循环中，并异步处理直到完成，此时连接被重新分配并从运行循环中删除。 结合仔细使用系统调用(syscall)和精确实现支持接口(如 pool 和 slab 内存分配器)，nginx 通常可以在极端工作负载下实现中到低的 CPU 使用。

在一些磁盘使用和 CPU 负载模式，应调整 nginx 工作(worker)的数量。 在这里说一点基础规则：系统管理员应该为其工作负载尝试几个配置。 一般建议可能如下：如果负载模

式是 CPU 密集型的，例如，处理大量 TCP/IP，执行 SSL 或压缩，则 nginx 工作(worker)的数量应与 CPU 内核数量相匹配；如果负载大多是磁盘 I/O 绑定，例如，从存储或重代理服务不同的内容集合 - 工作(worker)的数量可能是核心数量的一到两倍。有些工程师会根据个人存储单元的数量选择工作(worker)的数量,但这种方法的效率取决于磁盘存储的类型和配置。

nginx 的开发人员将在即将推出的版本中解决的一个主要问题是如何避免磁盘 I/O 上的大多数阻塞。 目前，如果没有足够的存储性能来提供特定工作(worker)生成的磁盘操作，该工作(worker)可能仍然阻止从磁盘读取/写入。存在许多机制和配置文件指令来减轻此类磁盘 I/O 阻塞情况。要注意的是，诸如：sendfile 和 AIO 之类的选项的组合通常会为磁盘性能带来很大的余量。 应该根据数据集，可用于 nginx 的内存量和底层存储架构来规划安装一个 nginx 服务器。

现有工作(worker)模式的另一个问题是与嵌入式脚本的有限支持有关。 一个使用标准的 nginx 分发，只支持嵌入 Perl 脚本。一个简单的解释：关键问题是嵌入式脚本阻止任何操作或意外退出的可能性。 这两种类型的行为将立即导致工作(worker)挂起的情况，同时影响到数千个连接。 更多的工作(worker)计划是使 nginx 的嵌入式脚本更简单，更可靠，适用于更广泛的应用。

nginx 进程角色

nginx 在内存中运行多个进程； 有一个主进程和几个工作(worker)进程。 还有一些特殊用途的过程,特别是缓存加载器和缓存管理器。所有进程都是单线程版本为 1.x 的 nginx。所有进程主要使用共享内存机制进行进程间通信。主进程作为 root 用户运行。 缓存加载器，缓存管理器和工作(worker)则以无权限用户运行。

主程序负责以下任务：

- 读取和验证配置
- 创建，绑定和关闭套接字
- 启动，终止和维护配置的工作(worker)进程数
- 重新配置，无需中断服务
- 控制不间断的二进制升级(如果需要，启动新的二进制并回滚)
- 重新打开日志文件
- 编译嵌入式 Perl 脚本

工作(worker)进程接受，处理和处理来自客户端的连接，提供反向代理和过滤功能，并执行几乎所有其他的 nginx 能力。 关于监视 nginx 实例的行为，系统管理员应该关注工作(worker)进程，因为它们是反映 Web 服务器实际日常操作的过程。

缓存加载器进程负责检查磁盘缓存项目，并使用缓存元数据填充 nginx 的内存数据库。本质上，缓存加载器准备 nginx 实例来处理已经存储在磁盘上的特定分配的目录结构中的文件。 它遍历目录，检查缓存内容元数据，更新共享内存中的相关条目，然后在所有内容清洁并准备使用时退出。

缓存管理器主要负责缓存到期和无效。 在正常的 nginx 操作期间它保持在内存中，并且在失败的情况下由主进程重新启动。

nginx 缓存简介

在 nginx 中的缓存以文件系统上的分层数据存储的形式实现。 缓存密钥是可配置的，并且可以使用不同的请求特定参数来控制进入缓存的内容。 缓存密钥和缓存元数据存储在共享存储器段中，高速缓存加载器，缓存管理器和工作(worker)可以访问它们。 目前，除了操作系统的虚拟文件系统机制暗示的优化之外，没有任何内存中的文件缓存。 每个缓存的响应都放在文件系统上的不同文件中。 层次结构(级别和命名细节)通过 nginx 配置指令进行控制。当响应写入缓存目录结构时，文件的路径和名称将从代理 URL 的 MD5 哈希导出。

将内容放置在缓存中的过程如下：当 nginx 从上游服务器读取响应时，内容首先写入缓存目录结构之外的临时文件。 当 nginx 完成处理请求时，它重命名临时文件并将其移动到缓存目录。 如果用于代理的临时文件目录位于另一个文件系统上，则该文件将被复制，因此建议将临时文件目录和缓存目录保存在同一文件系统上。 当需要显式清除缓存目录结构时，从文件中删除文件也是非常安全的。 nginx 有第三方扩展，可以远程控制缓存的内容，还有更多的工作计划将此功能集成到主分发中。

Nginx 安装

准备第三方支持库源码：

- nginx-1.13.7.tar.gz
- openssl-1.1.0g.tar.gz
- pcre-8.41.tar.gz
- zlib-1.2.11.tar.gz

```
wangbojing@ubuntu:~/share/nginx$ ls
nginx-1.13.7         openssl-1.1.0g         pcre-8.41         zlib-1.2.11
nginx-1.13.7.tar.gz  openssl-1.1.0g.tar.gz  pcre-8.41.tar.gz  zlib-1.2.11.tar.gz
wangbojing@ubuntu:~/share/nginx$ cd openssl-1.1.0g/
```

解压每个包

```
$ tar xzvf nginx-1.13.7.tar.gz
$ tar xzvf openssl-1.1.0g.tar.gz
$ tar xzvf pcre-8.41.tar.gz
$ tar xzvf zlib-1.2.11.tar.gz

$ cd nginx-1.13.7
$  ./configure  --prefix=/usr/local/nginx  --with-http_realip_module  --with-http_addition_module  --with-http_gzip_static_module  --with-http_secure_link_module --with-http_stub_status_module --with-stream --with-pcre=/home/wangbojing/share/nginx/pcre-8.41  --with-zlib=/home/wangbojing/share/nginx/zlib-1.2.11  --with-openssl=/home/wangbojing/share/nginx/openssl-1.1.0g

$ make
```

```
$ sudo make install
```

在/usr/local/目录下面，产生了 nginx 的目录

```
wangbojing@ubuntu:/usr/local/nginx$ ls
conf  html  logs  sbin
wangbojing@ubuntu:/usr/local/nginx$
```

```
$ ./sbin/nginx -c ./conf/nginx.conf
```

192.168.199.133

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

Nginx 快速入门

主要介绍 nginx 的基本配置和操作，并介绍了一些可以完成的简单任务。假设您已经学习过并已经安装好了 nginx 服务器。 如果没有，请参阅安装 nginx 页面。 本指南介绍如何启动和停止 nginx，并重新加载其配置，解释配置文件的结构，并介绍如何设置 nginx 以提供静态内容，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 有一个主进程和几个工作进程。 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实际处理。 nginx 采用基于事件的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

在配置文件中确定 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。

在前面安装文章配置中，使用的安装配置目录是：/usr/local/nginx/conf 。所以可以在这个目录下找到这个配置文件。

1. 启动，停止和重新加载 Nginx 配置

要启动 nginx，请运行可执行文件。 当 nginx 启动后，可以通过使用-s 参数调用可执行文件来控制它。 使用以下语法：

```
$ nginx -s signal
```

信号(signal)的值可能是以下之一：

- stop - 快速关闭服务
- quit - 正常关闭服务
- reload - 重新加载配置文件
- reopen - 重新打开日志文件

例如，要通过等待工作进程完成服务当前请求来停止 nginx 进程，可以执行以下命令：

```
$ nginx -s quit
```

注：该命令应该在启动 nginx 的同一用户下执行。

在将重新配置命令的命令发送到 nginx 或重新启动之前，配置文件中的更改将不会被应用。

要重新加载配置文件，请执行：

```
$ nginx -s reload
```

当主进程收到要重新加载配置的信号，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。 如果这是成功的，主进程将启动新的工作进程，并向旧的工作进程发送消息，请求它们关闭。 否则，主进程回滚更改，并继续使用旧配置。 老工作进程，接收关闭命令，停止接受新连接，并继续维护当前请求，直到所有这些请求得到维护。 之后，旧的工作进程退出。

还可以借助 Unix 工具(如 kill utility)将信号发送到 nginx 进程。 在这种情况下，信号直接发送到具有给定进程 ID 的进程。 默认情况下，nginx 主进程的进程 ID 写入目录/usr/local/nginx/logs 或/var/run 中的 nginx.pid。 例如，如果主进程 ID 为 1628，则发送 QUIT 信号导致 nginx 的正常关闭，请执行：

```
$ kill -s QUIT 1628
```

要获取所有运行的 nginx 进程的列表，可以使用 ps 命令，例如，以下列方式：

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

2. 配置文件的结构

Nginx 由配置文件中指定的指令控制的模块组成。 指令分为简单指令和块指令。 一个简单的指令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({和})包围的一组附加指令结束。 如果块指令可以在大括号内部有其他指令，则称为上下文(例如：events，http，server 和 location)。

配置文件中放置在任何上下文之外的伪指令都被认为是**主上下文**。 events 和 http 指令驻留在**主上下文**中，server 在 http 中的，而 location 在 http 块中。

#号之后的一行的部分被视为注释。

3. 提供静态内容服务(静态网站)

一个重要的 Web 服务器任务是提供文件(如图像或静态 HTML 页面)。这里我们来学习如何实现一个示例，根据请求，文件将从不同的本地目录提供： /usr/local/nginx/html/www(包含 HTML 文件)和/usr/local/nginx/html/images(包含图像)。这将需要编辑配置文件，并使用两个位置块在 http 块内设置服务器块。

首先，创建/usr/local/nginx/html/www 目录，并将一个包含任何文本内容的 exmaple.html 文件放入其中，并创建/usr/local/nginx/html/images 目录并在其中放置一些图像。

```
root@ubuntu:/usr/local/nginx# mkdir -p /data/images
root@ubuntu:/usr/local/nginx#
```

分别在上面创建的两个目录中放入两个文件：/usr/local/nginx/html/www/exmaple.html 和 /usr/local/nginx/html/1.png, 2.png, 3.png，/usr/local/nginx/html/ww 文件的内容就一行，如下

```
<h2> 0voice Demo.</h2>
```

接下来，打开配置文件(/usr/local/nginx/conf/nginx.conf)。 默认的配置文件已经包含了服务

器块的几个示例，大部分是注释掉的。 现在注释掉所有这样的块，并启动一个新的服务器块：

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

通常，配置文件可以包括服务器监听的端口和服务器名称区分的几个 server 块。当 nginx 决定哪个服务器处理请求后，它会根据服务器块内部定义的 location 指令的参数测试请求头中指定的 URI。

将以下 location 块添加到服务器(server)块：

```
http {
    server {
        location / {
            root /usr/local/nginx/html/www;
        }
    }
}
```

该 location 块指定与请求中的 URI 相比较的“/”前缀。 对于匹配请求，URI 将被添加到 root 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。 如果有几个匹配的 location 块，nginx 将选择具有最长前缀来匹配 location 块。 上面的 location 块提供最短的前缀长度为 1，因此只有当所有其他 location 块不能提供匹配时，才会使用该块。

接下来，添加第二个 location 块：

```
http {
    server {
        location / {
            root /usr/local/nginx/html/www;
        }
        location /images/ {
            root /usr/local/nginx/html;
        }
    }
}
```

它将是以/images/(位置/也匹配这样的请求，但具有较短前缀，也就是“/images/”比“/”长)的请求来匹配。

server 块的最终配置应如下所示：

```
server {
    location / {
          root /usr/local/nginx/html/www;
    }
    location /images/ {
          root /usr/local/nginx/html;
    }
}
```

这已经是一个在标准端口 80 上侦听并且可以在本地机器上访问的服务器(http://localhost/)的工作配置。 响应以/images/开头的 URI 的请求，服务器将从/data/images 目录发送文件。例如，响应 http://192.168.199.133:8080/images/logo.jpg 请求，nginx 将发送服务上的/usr/local/nginx/html/images/logo.jpg 文件。 如果文件不存在，nginx 将发送一个指示 404 错误的响应。 不以/images/开头的 URI 的请求将映射到/usr/local/nginx/html/www 目录。 例如，响应 http://192.168.199.133:8080/example.html 请求时，nginx 将发送/usr/local/nginx/html/www /example.html 文件。

要应用新配置，如果尚未启动 nginx 或者通过执行以下命令将重载信号发送到 nginx 的主进程：

```
root@ubuntu:/usr/local/nginx# /usr/local/nginx/sbin/nginx -t
nginx: the configuration file /usr/local/nginx/conf/nginx.conf syntax is ok
nginx: configuration file /usr/local/nginx/conf/nginx.conf test is successful
root@ubuntu:/usr/local/nginx# /usr/local/nginx/sbin/nginx -s reload
```

如果错误或异常导致无法正常工作，可以尝试查看目录/usr/local/nginx/logs 或/var/log/nginx 中的 access.log 和 error.log 文件中查找原因。

打开浏览器或使用 CURL 访问 Nginx 服务器如下所示:

完整的 nginx.conf 文件配置内容如下：

```
events {
    worker_connections  1024;
}
```

```
http {
       server {
              listen       8080;
              server_name  localhost;
              location / {
                     root /usr/local/nginx/html/www;
              }
              location /images/ {
                     root /usr/local/nginx/html;
              }
       }
}
```

4. 设置简单的代理服务器

nginx 的一个常见用途是将其设置为代理服务器，这意味着它可作为一个接收请求的服务器，将其传递给代理服务器，从代理服务器中检索响应，并将其发送给客户端。

我们将配置一个基本的代理服务器，它为来自本地目录的文件提供图像请求，并将所有其他请求发送到代理的服务器。 在此示例中，两个服务器将在单个 nginx 实例上定义。

首先，通过向 nginx 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 / {
    }
}
```

这将是一个监听端口 8080 的简单服务器(以前，由于使用了标准端口 80，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 块上下文中。 当选择用于服务请求的 location 块不包含自己的 root 指令时，将使用此 root 指令。

在上面创建的目录/data/up1 中放入一个文件：/data/www/demo.html，文件的内容就一行，如下 -

<h2>About proxy_pass Page at port 8080</h2>

接下来，使用上一节中的服务器配置进行修改，使其成为代理服务器配置。 在第一个位置块中，将 proxy_pass 指令与参数中指定的代理服务器的协议，名称和端口(在本例中为 http://localhost:8080)：

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

我们将修改当前使用/images/prefix 将请求映射到/data/images 目录下的文件的第二个 location 块，使其与典型文件扩展名的图像请求相匹配。 修改后的位置块如下所示：

```
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~字符。 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，否则选择之前记住的那一个。

代理服务器的最终配置将如下所示：

```
server {
    location / {
        proxy_pass http://192.168.199.128:8080/;
    }
    location ~ \.(gif|jpg|png)$ {
        root /data/images;
    }
}
```

此服务器将过滤以.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)，并将所有其他请求传递到上面配置的代理服务器。

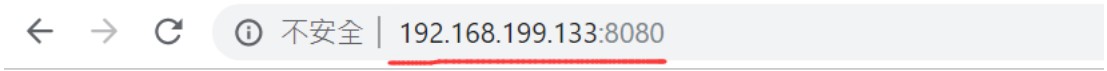
要应用新配置，如果尚未启动 nginx 或者通过执行以下命令将重载信号发送到 nginx 的主进程：

```
root@ubuntu:/usr/local/nginx# ./sbin/nginx -c ./conf/demo.conf
```

首先测试上面配置的 8080 端口的服务，访问服务的 8080 端口，得到以下结果：

0voice Demo.

再次访问 80 端口(这里只是一个代理，它会把请求转发给 8080 的服务，由 8080 端口这这个服务处理并返回结果到客户端)，得到以下结果 -



html in host:192.168.199.128

完整的配置 nginx.conf 文件内容如下：

```
events {
    worker_connections  1024;
}

http {
```

```
        server {
                listen       8080;
                server_name  localhost;
                location / {
                        root /usr/local/nginx/html/www;
#                       proxy_pass http://192.168.199.128;
                }
                location /images/ {
                        root /usr/local/nginx/html;
                }
        }
}
```

5. 设置 FastCGI 代理

nginx 可用于将请求路由到运行使用各种框架和 PHP 等编程语言构建的应用程序的 FastCGI 服务器。使用 FastCGI 服务器的最基本 nginx 配置包括使用 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;
    }
}
```

这将设置一个服务器，将除静态图像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。

Nginx 进程和运行时控制

介绍 NGINX 在运行时启动的过程以及如何控制它们。

在这个部分中，主要涉及两个部分的内容：

- 主进程和工作进程
- 控制 NGINX

1. 主进程和工作进程

NGINX 有一个主进程和一个或多个工作进程。 如果启用缓存，缓存加载程序和缓存管理器进程也将在启动时运行。

主程序的主要目的是读取和评估配置文件以及维护工作进程。 工作进程执行请求的实际处理。 NGINX 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 nginx.conf 配置文件中定义，可以针对给定的配置进行修复，或者自动调整为可用 CPU 内核数。

2. 控制 Nginx

要重新加载配置文件，可以停止或重新启动 NGINX，或者发送信号到主进程。 可以使用-s 参数运行 nginx 命令(调用 NGINX 可执行文件)来发送信号。

```
$ nginx -s signal
```

信号的值可以是以下之一：

- quit – 正常地关闭
- reload – 重新加载配置文件
- reopen – 重新打开日志文件
- stop – 立即关闭(快速关闭)

杀死实用程序也可以使用，将信号直接发送到主进程。 默认情况下，主进程的进程 ID 被写入位于/usr/local/nginx/logs 或/var/run 目录中的 nginx.pid 文件。

nginx 可以用信号控制。 默认情况下，主进程的进程 ID 将写入文件/usr/local/nginx/logs/nginx.pid。 该名称可能在配置时更改，或使用 pid 指令在 nginx.conf 文件中进行更改。主程序支持以下信号：

- TERM, INT - 快速关闭
- QUIT - 正常关闭
- HUP - 改变配置，跟上改变的时区(仅适用于 FreeBSD 和 Linux)，使用新配置启动新的工作进程，正常关闭旧的工作进程
- USR1 - 重新打开日志文件
- USR2 - 升级可执行文件
- WINCH - 正常关闭工作进程

个别工作进程可以用信号来控制，尽管这不是必需的。 支持的信号有：

- 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(需要启用 debug_points)

更改配置

为了使 nginx 重新读取配置文件，应将 HUP 信号发送到主进程。 主进程首先检查语法有效性，然后尝试应用新配置，即打开日志文件和新的监听套接字。 如果失败，它会回滚更改，并继续使用旧配置。 如果此操作成功，它将启动新的工作进程，并向旧的工作进程发送消息，请求它们正常关闭。 旧工作进程密切监听套接字，并继续为旧客户端服务。 在所有客户端被服务之后，旧的工作进程被关闭。
我们来举例说明一下。 想象一下，nginx 是在 ubuntu14.04 上运行，执行以下命令：

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

产生以下输出：

```
  PID  PPID USER    %CPU   VSZ WCHAN  COMMAND
33126     1 root     0.0  1148 pause  nginx: master process
/usr/local/nginx/sbin/nginx
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)
```

如果将 HUP 发送到主进程，则输出变为：

```
  PID  PPID USER    %CPU   VSZ WCHAN  COMMAND
33126     1 root     0.0  1164 pause  nginx: master process
/usr/local/nginx/sbin/nginx
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)
```

PID 33129 的老工作流程仍然继续运行。 一段时间后，它退出：

```
 PID  PPID USER    %CPU   VSZ WCHAN  COMMAND
33126     1 root     0.0  1164 pause  nginx: master process
/usr/local/nginx/sbin/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)
```

循环日志文件

要循环日志文件，需要首先重命名。 之后，USR1 信号应发送到主进程。 然后，主进程将重新打开所有当前打开的日志文件，并将其分配给正在运行的工作进程的非特权用户作为所有者。 成功重新打开后，主程序关闭所有打开的文件，并将消息发送到工作进程，要求他们重新打开文件。 工作进程也会打开新文件并立即关闭旧文件。 因此，旧文件几乎立

即可用于后处理，如压缩。

Nginx 配置文件

NGINX 与其他服务类似，因为它具有以特定格式编写的基于文本的配置文件。 默认情况下，文件名为 nginx.conf 并放在/etc/nginx 目录中(对于开源 NGINX 产品，位置取决于用于安装 NGINX 和操作系统的软件包系统，它通常位于/usr/local/nginx/conf/etc/nginx 或 /usr/local/etc/nginx。)

配置文件由指令及其参数组成。 简单(单行)指令各自以分号结尾。 其他指令作为“容器”，将相关指令组合在一起，将其包围在花括号({})中。 以下是简单指令的一些示例。

```
user             nobody;
error_log        logs/error.log notice;
worker_processes 1;
```

为了使配置更易于维护，建议您将其拆分为存储在/etc/nginx/conf.d 目录中的一组功能特定文件，并在主 nginx.conf 文件中使用 include 指令引用(包函)指定文件的内容。如下所示 -

```
include conf.d/http;
include conf.d/stream;
include conf.d/exchange-enhanced;
```

几个顶级指令(称为上下文)将适用于不同流量类型的指令组合在一起：

- events – 一般连接处理
- http – HTTP 协议流量
- mail – Mail 协议流量
- stream – TCP 协议流量

指定在这些上下文之外的指令是在主上下文中。

在每个流量处理上下文中，可包括一个或多个服务器上下文来定义控制请求处理的虚拟服务器。 您可以在服务器环境中包含的指令根据流量类型而有所不同。

对于 HTTP 流量(http 上下文)，每个服务器指令控制对特定域或 IP 地址上的资源请求的处理。服务器上下文中的一个或多个位置上下文定义了如何处理特定的 URI 集合。

对于邮件和 TCP 流量(mail 和 stream 上下文)，服务器指令各自控制到达特定 TCP 端口或 UNIX 套接字的流量处理。

以下配置说明了上下文的使用情况。

```
user nobody; # a directive in the 'main' context

events {
    # configuration of connection processing
}
http {
    # Configuration specific to HTTP and affecting all virtual servers
    server {
        # configuration of HTTP virtual server 1
        location /one {
            # configuration for processing URIs with '/one'
        }
```

```
        location /two {
            # configuration for processing URIs with '/two'
        }
    }
    server {
        # configuration of HTTP virtual server 2
    }
}
stream {
    # Configuration specific to TCP and affecting all virtual servers
    server {
        # configuration of TCP virtual server 1
    }
}
```

对于大多数指令，在另一个上下文(子上下文)中定义的上下文将继承父级中包含的伪指令的值。 要覆盖从父进程继承的值，请在子上下文中包含该指令。
要更改配置文件才能生效，NGINX 必须重新加载该文件。可以重新启动 nginx 进程或发送 reload 信号来升级配置，而不会中断当前请求的处理。

Nginx 配置 Web 服务器

介绍如何将 NGINX 配置作为 Web 服务器，并包括以下部分：

- 设置虚拟服务器
- 配置位置
- 使用变量
- 返回特定状态码
- 重写请求中的 URI
- 重写 HTTP 响应
- 处理错误

在高层次上，将 NGINX 配置作为 Web 服务器有一些问题需要了解，定义它处理哪些 URL 以及如何处理这些 URL 上的资源的 HTTP 请求。 在较低层次上，配置定义了一组控制对特定域或 IP 地址的请求的处理的虚拟服务器。

用于 HTTP 流量的每个虚拟服务器定义了称为位置的特殊配置实例，它们控制特定 URI 集合的处理。 每个位置定义了自己的映射到此位置的请求发生的情况。 NGINX 可以完全控制这个过程。 每个位置都可以代理请求或返回一个文件。 此外，可以修改 URI，以便将请求重定向到另一个位置或虚拟服务器。 此外，可以返回特定的错误代码，也可以配置特定的页面以对应于每个错误代码。

1. 虚拟服务器

NGINX 配置文件必须至少包含一个服务器指令来定义虚拟服务器。 当 NGINX 处理请求时，它首先选择提供请求的虚拟服务器。

虚拟服务器由 http 上下文中的服务器指令定义，例如：

```
http {
    server {
        # Server configuration
    }
}
```

可以将多个 server 指令添加到 http 上下文中以定义多个虚拟服务器。

server 配置块通常包括一个 listen 指令，用于指定服务器侦听请求的 IP 地址和端口(或 Unix 域套接字和路径)。IPv4 和 IPv6 地址均被接受; 将方括号(。

下面的示例显示了监听 IP 地址 127.0.0.1 和端口 8080 的服务器的配置：

```
server {
    listen 127.0.0.1:8080;
    # The rest of server configuration
}
```

如果省略端口，则使用标准端口。 同样地，如果省略一个地址，服务器将侦听所有地址。

如果没有包含 listen 指令，则“标准”端口为 80/tcp，“default”端口为 8000/tcp，具体取决于超级用户权限。

如果有多个服务器与请求的 IP 地址和端口相匹配，则 NGINX 将根据服务器块中的 server_name 指令测试请求的主机头域。 server_name 的参数可以是完整(精确)名称，通配符或正则表达式。 通配符是一个字符串，其开头，结尾或两者都包含星号(*); 星号匹配任何字符序列。 NGINX 将 Perl 语法用于正则表达式; 在它们之前使用波浪号(～)。 此示例说明了一个确切的名称。

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

如果匹配主机头几个名称，则 NGINX 通过按以下顺序搜索名称并使用其找到的第一个匹配来选择一个：

- 确切的名字(完整准确的名称)
- 以星号开头的最长通配符，例如:*.example.org
- 以星号结尾的最长通配符，如：mail.*
- 第一个匹配正则表达式(按照出现在配置文件中的顺序)

如果主机头字段与服务器名称不匹配，则 NGINX 会将请求路由到请求到达端口的默认服务器。 默认服务器是 nginx.conf 文件中列出的第一个服务器，除非您将 listen_server 参数包含在 listen 指令中以明确指定服务器为默认值。

```
server {
    listen    80    default_server;
    ...
}
```

一个完整的 Nginx 虚拟机配置示例，这里我们演示配置两个虚拟机，对应域名分别为：vhost1.com 和 vhost2.com，vhost1.com 网站的主目录在 html/www/vhost1，vhost2.com 网站的主目录在/usr/local/nginx/html/www/vhost1：

```
events {
```

```
    worker_connections  1024;
}

http {
   server {
      listen       8888;
         server_name  vhost1.com www.vhost1.com;
      index example.html;

      root /usr/local/nginx/html/www/vhost1/;
      access_log /usr/local/nginx/logs/vhost1.com.log;
   }
   server {
               listen       8889;
               server_name  vhost2.com www.vhost2.com;
               index example.html;

               root /usr/local/nginx/html/www/vhost2/;
               access_log /usr/local/nginx/logs/vhost2.com.log;
        }
```

2. 配置位置

NGINX 可以根据请求 URI 向不同的代理发送流量或提供不同的文件。 这些块是使用放置在 server 指令中的 location 指令来定义的。

例如，您可以定义三个 location 块，以指示虚拟服务器向一个代理服务器发送一些请求，将其他请求发送到不同的代理服务器，并通过从本地文件系统传递文件来提供其余请求。

NGINX 测试根据所有 location 指令的参数请求 URI，并应用匹配 location 中定义的指令。 在每个 location 块内，通常可能(除了一些例外)放置更多的 location 指令以进一步细化特定组请求的处理。

注意：在本教程文章中，单词 location 是指单个 location 上下文。

location 指令有两种类型的参数：前缀字符串(路径名)和正则表达式。 对于要匹配前缀字符串的请求 URI，必须以前缀字符串开头。

具有 pathname 参数的以下示例位置匹配以/some/path/开头的请求 URI，例如/some/path/document.html，它不匹配/my-site/some/path，因为/some/path 不在该 URI 的开头出现。

```
location /some/path/ {
    ...
}
```

正则表达式之前是区分大小写匹配的波形符号(~)，或者不区分大小写匹配的波形符号(~*)。以下示例将包含字符串.html 或.html 的 URI 与任何位置相匹配。

```
location ~ \.html? {
    ...
```

```
}
```

要找到最符合 URI 的位置，NGINX 首先将 URI 与前缀字符串的位置进行比较。然后用正则表达式搜索位置。

除非使用^~修饰符对正则表达式给予更高的优先级。在前缀字符串中，NGINX 选择最具体的字符串(也就是最长和最完整的字符串)。 下面给出了选择处理请求的位置的确切逻辑：

- 测试所有 URI 的前缀字符串。
- =(等号)修饰符定义了 URI 和前缀字符串完全匹配。如果找到完全匹配，则搜索停止。
- 如果^~(插入符号)修饰符预先添加最长匹配前缀字符串，则不会检查正则表达式。
- 存储最长匹配的前缀字符串。
- 根据正则表达式测试 URI。
- 断开第一个匹配的正则表达式并使用相应的位置。
- 如果没有正则表达式匹配，则使用与存储的前缀字符串相对应的位置。

=修饰符的典型用例是/(正斜杠)的请求。 如果请求/是频繁的，则指定=/作为 location 指令的参数加速处理，因为搜索匹配在第一次比较之后停止。

```
location = / {
    ...
}
```

location 上下文可以包含定义如何解析请求的指令 - 提供静态文件或将请求传递给代理的服务器。 在以下示例中，匹配第一个 location 上下文的请求将从/data/images 目录中提供文件，并将匹配第二个位置的请求传递给承载 www.example.com 域内容的代理服务器。

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

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

root 指令指定要在其中搜索要提供的静态文件的文件系统路径。 与该位置相关联的请求 URI 将附加到路径，以获取要提供的静态文件的全名。 在上面的示例中，要响应/images/logo.png 的请求，NGINX 提供服务器本地实际对应文件是：/data/images/logo.png。

proxy_pass 指令将请求传递给使用配置的 URL 访问代理服务器。然后将代理服务器的响应传回客户端。在上面的示例中，所有不以/images/开头的 URI 的请求都将被传递给代理的服务器(也就是：www.example.com)。

3. 使用变量

可以使用配置文件中的变量，使 NGINX 进程的请求根据定义的情况而有所不同。 变量是在运行时计算的命名值，用作指令的参数。 一个变量由它的名字开头的$(美元)符号表示。 变量根据 NGINX 的状态定义信息，例如正在处理的请求的属性。

有许多预定义的变量，如核心 HTTP 变量，您可以使用 set，map 和 geo 指令定义自定义变

量。 大多数变量在运行时计算的，并包含与特定请求相关的信息。 例如，$remote_addr 包含客户端 IP 地址，$uri 保存当前的 URI 值。

4. 返回特定状态码

一些网站 URI 需要立即返回具有特定错误或重定向代码的响应，例如当页面被暂时移动或永久移动时。 最简单的方法是使用 return 指令。 例如返回未找到的 404 状态码：

```
location /wrong/url {
    return 404;
}
```

返回的第一个参数是响应代码。可选的第二个参数可以是重定向的 URL(代码 301,302,303 和 307)或在响应体中返回文本。 例如：

```
location /permanently/moved/url {
    return 301 http://www.example.com/moved/here;
}
```

返回指令可以包含在 location 和 server 上下文中。

5. 重写 URI 请求

可以通过使用 rewrite 指令在请求处理期间多次修改请求 URI，该指令具有一个可选参数和两个必需参数。 第一个(必需)参数是请求 URI 必须匹配的正则表达式。 第二个参数是用于替换匹配 URI 的 URI。 可选的第三个参数是可以停止进一步重写指令的处理或发送重定向(代码 301 或 302)的标志。例如：

```
location /users/ {
    rewrite ^/users/(.*)$ /show?user=$1 break;
}
```

如该示例所示，用户通过匹配正则表达式捕获第二个参数。

您可以在 location 和 server 上下文中包含多个 rewrite 指令。 NGINX 按照它们发生的顺序逐个执行指令。 当选择该上下文时，server 上下文中的 rewrite 指令将被执行一次。

在 NGINX 处理一组 rewrite 指令之后，它根据新的 URI 选择一个 location 上下文。 如果所选 location 块包含 rewrite 指令，则依次执行它们。 如果 URI 与其中任何一个匹配，则在处理所有定义的 rewrite 指令之后，将搜索新 location 块。

以下示例显示了与返回指令相结合的 rewrite 指令。

```
server {
    ...
    rewrite ^(/download/.*)/media/(.*)\..*$ $1/mp3/$2.mp3 last;
    rewrite ^(/download/.*)/audio/(.*)\..*$ $1/mp3/$2.ra  last;
    return  403;
    ...
}
```

此示例配置区分两组 URI。 诸如/download/some/media/file 之类的 URI 更改为/download/some/mp3/file.mp3。由于最后一个标志，所以跳过后续指令(第二次

rewrite 和 return 指令)，但 NGINX 继续处理该请求，该请求现在具有不同的 URI。类似地,诸如/download/some/audio/file 的 URI 被替换为/download/some/mp3/file.ra。如果 URI 与 rewrite 指令不匹配，则 NGINX 将 403 错误代码返回给客户端。
有两个中断处理重写指令的参数：

- last - 停止执行当前服务器或位置上下文中的重写指令，但是 NGINX 会搜索与重写的 URI 匹配的位置,并且应用新位置中的任何重写指令(URI 可以再次更改,往下继续匹配)。
- break- 像 break 指令一样，在当前上下文中停止处理重写指令，并取消搜索与新 URI 匹配的位置。新位置(location)块中的 rewrite 指令不执行。

6. 重写 HTTP 响应

有时您需要重写或更改 HTTP 响应中的内容，将一个字符串替换为另一个字符串。 您可以使用 sub_filter 指令来定义要应用的重写。 该指令支持变量和替代链，使更复杂的更改成为可能。
例如，您可以更改引用除代理服务器之外的绝对链接：

```
location / {
    sub_filter      /blog/ /blog-staging/;
    sub_filter_once off;
}
```

另一个示例将方法从 http://更改为 http://,并从请求头域替换本地主机地址到主机名。sub_filter_once 指令告诉 NGINX 在一个位置(location)内连续应用 sub_filter 伪指令：

```
location / {
    sub_filter                  'href="http://127.0.0.1:8080/'
'href="http://$host/';
    sub_filter          'img  src="http://127.0.0.1:8080/'  'img
src="http://$host/';
    sub_filter_once on;
}
```

请注意，如果发生另一个 sub_filter 匹配，则使用 sub_filter 修改的响应部分将不再被替换。

7. 处理错误

使用 error_page 指令，您可以配置 NGINX 返回自定义页面以及错误代码，替换响应中的其他错误代码，或将浏览器重定向到其他 URI。 在以下示例中，error_page 指令指定要返回 404 页面错误代码的页面(/404.html)。

```
error_page  404  /404.html;
```

请注意，此伪指令并不立即返回该错误(返回指令执行该操作)，而仅仅是指定发生时如何处理错误。 错误代码可以来自代理服务器，或者在 NGINX 处理期间发生(例如，当 NGINX 找不到客户端请求的文件时，显示 404 对应的结果)。
在以下示例中，当 NGINX 找不到页面时，它会将代码 301 替换为代码 404，并将客户端重定向到http:/example.com/new/path.html。当客户端仍尝试访问其旧 URI 的页面时，

此配置非常有用。 301 代码通知浏览器页面已经永久移动，并且需要在返回时自动替换旧地址。

```
location /old/path.html {
    error_page 404 =301 http:/example.com/new/path.html;
}
```

以下配置是在未找到文件时将请求传递给后端的示例。 因为在 error_page 指令的等号之后没有指定状态代码,所以对客户机的响应具有代理服务器返回的状态代码(不一定是 404)。

```
server {
    ...
    location /images/ {
        # Set the root directory to search for the file
        root /data/www;

        # Disable logging of errors related to file existence
        open_file_cache_errors off;

        # Make an internal redirect if the file is not found
        error_page 404 = /fetch$uri;
    }

    location /fetch/ {
        proxy_pass http://backend/;
    }
}
```

当没有找到文件时，error_page 指令指示 NGINX 进行内部重定向。 error_page 指令的最终参数中的$uri 变量保存当前请求的 URI，该 URI 在重定向中被传递。

例如，如果没有找到/images/some/file，它将被替换为/fetch/images/some/file，并且新的搜索位置(location)开始。最后请求最终在第二个 location 上下文中，并被代理到 http://backend/。

如果没有找到文件，则 open_file_cache_errors 指令可防止写入错误消息。 因为丢失的文件可被正确地处理，但这不是必需的。

Nginx 配置静态内容服务器

介绍如何使用 NGINX 来提供静态内容服务，定义搜索路径以查找请求的文件的方法，以及如何设置索引文件。

在这个部分，我们主要涉及以下几个方面的内容：

- 根目录和索引文件
- 尝试几个选项
- 优化 NGINX 服务内容的速度

1. 根目录和索引文件

root 指令指定将用于搜索文件的根目录。 要获取请求文件的路径，NGINX 将请求 URI 附加到 root 指令指定的路径。 该指令可以放置在 http，server 或 location 上下文中的任何级别上。 在下面的示例中，为虚拟服务器定义了 root 指令。 它适用于不包括 root 指令的所有 location 块以显式重新定义根：

```
server {
    root /www/data;
    location / {
    }
    location /images/ {
    }
    location ~ \.(mp3|mp4) {
        root /www/media;
    }
}
```

这里，NGINX 在文件系统的/www/data/images/目录中搜索以/images/开头的 URI。 但是，如果 URI 以.mp3 或.mp4 扩展名结尾，则 NGINX 会在/www/media/目录中搜索.mp3 或.mp4 文件，因为它在匹配的 location 块中定义。

如果请求以斜杠结尾，则 NGINX 将其视为对目录的请求，并尝试在目录中找到索引文件。 index 指令定义索引文件的名称(默认值为 index.html)。

如果请求 URI 为/images/some/path/，则 NGINX 会传递文件/www/data/images/ /index.html(如果存在)。

如果不存在文件，NGINX 默认返回 HTTP 代码 404(未找到)。 要配置 NGINX 以返回自动生成的目录列表，请将 on 参数添加到 autoindex 指令中：

```
location /images/ {
    autoindex on;
}
```

可以在索引指令中列出多个文件名。 NGINX 以指定的顺序搜索文件，并返回它找到的第一个文件。

```
location / {
    index index.$geo.html index.html index.html;
}
```

这里使用的$geo 变量是通过 geo 指令设置的自定义变量。 变量的值取决于客户端的 IP 地址。

要返回索引文件，NGINX 检查其是否存在，然后通过将索引文件的名称附加到基本 URI 来对通过 URI 获取的内部重定向。内部重定向会导致对某个位置(location)的新搜索，并且可能会在另一个位置(location)中结束，如以下示例所示：

```
location / {
    root /data;
    index index.html index.php;
}
```

```
location ~ \.php {
    fastcgi_pass localhost:8000;
    ...
}
```

在这里，如果请求中的 URI 是/path/，并且/data/path/index.html 不存在，但是/data/path/index.php 存在，则将/path/index.php 内部重定向映射到第二个位置(location)。 因此，请求被代理。

2. 尝试几个选项

try_files 指令可用于检查指定的文件或目录是否存在并进行内部重定向，如果没有指定的文件或目录，则返回特定的状态代码。 例如，要检查与请求 URI 相对应的文件的存在，请使用 try_files 指令和$uri 变量，如下所示：

```
server {
    root /www/data;

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

该文件以 URI 的形式指定，它使用在当前位置或虚拟服务器的上下文中设置的 root 或 alias 伪指令进行处理。 在这种情况下，如果与原始 URI 相对应的文件不存在，则 NGINX 将内部重定向到最后一个参数中指定的 URI，也就是返回 default.gif。
最后一个参数也可以是一个状态代码(直接在前面的等号)或位置的名称。 在以下示例中，如果 try_files 指令的任何参数都不会解析为现有文件或目录，则会返回 404 错误。

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

在下一个示例中，如果原始 URI 和带有附加尾部斜线的 URI 都不能解析为现有文件或目录，则将请求重定向到将其传递给代理服务器的命名位置(location)。

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

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

3. 优化 NGINX 服务内容的速度

加载速度是服务任何内容的关键因素。 对您的 NGINX 配置进行小幅优化可能会提高生产力并帮助实现最佳性能。

启用 sendfile

默认情况下，NGINX 会自动处理文件传输，并在发送文件之前将其复制到缓冲区中。 启用 sendfile 指令将消除将数据复制到缓冲区中的步骤，并允许将数据从一个文件描述符直接复制到另一个文件描述符。 或者，为了防止一个快速连接完全占用工作进程，您可以通过定义 sendfile_max_chunk 指令来限制在单个 sendfile()调用中传输的数据量：

```
location /mp3 {
    sendfile           on;
    sendfile_max_chunk 1m;
    ...
}
```

启用 tcp_nopush

将 tcp_nopush 选项与 sendfile 一起使用。 该选项将使 NGINX 能够通过 sendfile 获取数据块之后，在一个数据包中发送 HTTP 响应头

```
location /mp3 {
    sendfile   on;
    tcp_nopush on;
    ...
}
```

启用 tcp_nodelay

tcp_nodelay 选项可以覆盖 Nagle 的算法，最初是为了解决慢网络中的小数据包问题而设计的。 该算法将大量小数据包整合到较大的数据包中，并以 200 ms 的延迟发送数据包。 如今，当服务大型静态文件时，无论数据包大小如何，都可以立即发送数据。 延迟也会影响在线应用程序(ssh，在线游戏，网上交易)。 默认情况下，tcp_nodelay 指令设置为 on，表示 Nagle 的算法被禁用。 该选项仅用于 Keepalive 连接：

```
location /mp3  {
    tcp_nodelay       on;
    keepalive_timeout 65;
    ...
}
```

优化积压队列

其中一个重要因素是 NGINX 可以处理传入连接的速度。 一般规则是建立连接时，将其放入监听套接字的“侦听”队列中。 在正常负载下，有一个低队列，或根本没有队列。 但是在高负载下，队列可能会急剧增长，这可能会导致性能不均衡，连接丢失和延迟。

测量侦听队列

让我们来查看当前的侦听队列。 运行命令：

```
netstat -Lan
```

命令输出可能如下所示：

```
Current listen queue sizes (qlen/incqlen/maxqlen)
Listen         Local Address
0/0/128        *.12345
10/0/128        *.80
0/0/128        *.8080
```

命令输出显示端口 80 的监听队列中有 10 个不接受的连接，而连接限制为 128 个连接，这种情况是正常的。

但是，命令输出可能如下所示：

```
Current listen queue sizes (qlen/incqlen/maxqlen)
Listen         Local Address
0/0/128        *.12345
192/0/128        *.80
0/0/128        *.8080
```

命令输出显示超过 128 个连接限制的 192 个不可接受的连接。 当网站的流量很大时，这是很常见的。 为了达到最佳性能，您需要增加 NGINX 在操作系统和 NGINX 配置中排队等待接收的最大连接数。

调整操作系统

将 net.core.somaxconn 键的值从其默认值(128)增加到足够高的值以能够处理高突发流量：

打开文件：/etc/sysctl.conf，将下面一行添加到文件并保存文件：

```
net.core.somaxconn = 4096
```

调整 NGINX

如果将 somaxconn 键设置为大于 512 的值，请更改 NGINX listen 指令的 backlog 参数以匹配：

```
server {
    listen 80 backlog 4096;
    # The rest of server configuration
}
```

Nginx 反向代理

介绍代理服务器的基本配置。 您将学习如何通过不同协议将 NGINX 请求传递给代理的服务器，修改发送到代理服务器的客户端请求标头，以及配置来自代理服务器的响应缓冲。

代理服务器的基本配置目录

- 代理服务器介绍
- 将请求传递给代理的服务器
- 传递请求标头
- 配置缓冲区
- 选择传出 IP 地址

1. 代理服务器介绍

代理通常用于在多个服务器之间分配负载，无缝地显示来自不同网站的内容，或者通过 HTTP 以外的协议将请求传递给应用服务器。

2. 将请求传递给代理的服务器

当 NGINX 代理请求时，它将请求发送到指定的代理服务器，获取响应，并将其发送回客户端。 可以使用指定的协议将请求代理到 HTTP 服务器(另一个 NGINX 服务器或任何其他服务器)或非 HTTP 服务器(可以运行使用特定框架开发的应用程序，如 PHP 或 Python)。 支持的协议包括 FastCGI，uwsgi，SCGI 和 memcached。

要将请求传递给 HTTP 代理服务器，则在一个 location 块内指定 proxy_pass 指令。 例如：

```
location /some/path/ {
    proxy_pass http://www.example.com/link/;
}
```

此示例配置将在此 location 处理的所有请求传递到指定地址(www.example.com/link/)处的代理服务器。该地址可以指定为域名或 IP 地址。 该地址还可能包括一个端口：

```
location ~ \.php {
    proxy_pass http://127.0.0.1:8000;
}
```

请注意，在上述第一个示例中，代理服务器的地址后面是 URI 为 /link/。 如果 URI 与地址一起指定，它将替换与 location 参数匹配请求 URI 的部分。 例如，这里使用 /some/path/page.html 的 URI 请求将被代理到 www.example.com/link/page.html。 如果地址被指定为没有 URI，或者不可能确定要替换的 URI 部分，则会传递完整的请求 URI(可能是修改)。

要将请求传递给非 HTTP 代理服务器，应使用适当的**_ pass 指令：

- fastcgi_pass 将请求传递给 FastCGI 服务器
- uwsgi_pass 将请求传递给 uwsgi 服务器
- scgi_pass 将请求传递给 SCGI 服务器
- memcached_pass 将请求传递给 memcached 服务器

请注意，在这些情况下，指定地址的规则可能不同。 您可能还需要向服务器传递其他参数。

proxy_pass 指令也可以指向一组命名的服务器。 在这种情况下，根据指定的方法在组中的服务器之间分配请求。

3. 传递请求标头

默认情况下，NGINX 在代理请求“Host” 和 “Connection”中重新定义了两个头字段，并消除了其值为空字符串的头字段。 “Host”设置为$proxy_host 变量，“Connection”设置为关闭(close)。
要更改这些设置，以及修改其他 header 字段，请使用 proxy_set_header 指令。 该指令可以在一个或多个位置(location)指定。 它也可以在特定的 server 上下文或 http 块中指定。 例如：

```
location /some/path/ {
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_pass http://localhost:8000;
}
```

在此配置中，“Host”字段设置为 $host 变量。
为了防止头域被传递给代理服务器，请将其设置为空字符串，如下所示：

```
location /some/path/ {
    proxy_set_header Accept-Encoding "";
    proxy_pass http://localhost:8000;
}
```

4. 配置缓冲区

默认情况下，NGINX 缓存来自代理服务器的响应。 响应存储在内部缓冲区中，并且不会发送到客户端，直到收到整个响应。 缓冲有助于通过慢客户端优化性能，如果响应从 NGINX 同步传递到客户端，这可能会浪费代理服务器时间。 然而，当启用缓冲时，NGINX 允许代理服务器快速处理响应，而 NGINX 存储响应时间与客户端需要下载的时间一样长。

负责启用和禁用缓冲的指令是 proxy_buffering。 默认情况下，它被设置为开启且缓冲已启用。

proxy_buffers 指令控制分配给请求的缓冲区的大小和数量。 来自代理服务器的响应的第一部分存储在单独的缓冲区中，其大小由 proxy_buffer_size 指令设置。 这部分通常包含一个比较小的响应头，并且可以比其余的响应的缓冲区小。

在以下示例中，缓冲区的默认数量增加，并且响应的第一部分的缓冲区的大小小于默认值。

```
location /some/path/ {
    proxy_buffers 16 4k;
    proxy_buffer_size 2k;
    proxy_pass http://localhost:8000;
}
```

如果缓存被禁用，则在从代理服务器接收缓冲时，响应将同步发送到客户端。 对于需要尽快开始接收响应的快速交互式客户端，此行为可能是可取的。
要禁用特定位置的缓冲，请在 location 块中将 proxy_buffering 伪指令设置为 off，如下所示：

```
location /some/path/ {
    proxy_buffering off;
    proxy_pass http://localhost:8000;
}
```

在这种情况下，NGINX 只使用由 proxy_buffer_size 配置的缓冲区来存储响应的当前部分。

5. 选择传出 IP 地址

如果您的代理服务器有多个网络接口，有时您可能需要选择特定的源 IP 地址才能连接到代理服务器或上游。 如果 NGINX 后端的代理服务器只配置为接受来自特定 IP 网络或 IP 地址范围的连接，在这种情况下，这个配置选项就很有用。

指定 proxy_bind 指令和必要网络接口的 IP 地址：

```
location /app1/ {
    proxy_bind 127.0.0.1;
    proxy_pass http://example.com/app1/;
}

location /app2/ {
    proxy_bind 127.0.0.2;
    proxy_pass http://example.com/app2/;
}
```

IP 地址也可以用变量指定。 例如，$server_addr 变量传递接受请求的网络接口的 IP 地址：

```
location /app3/ {
    proxy_bind $server_addr;
    proxy_pass http://example.com/app3/;
}
```

Nginx 压缩和解压

介绍如何配置响应的压缩或解压缩以及发送压缩文件。涉及内容如下：

- 压缩和解压缩介绍
- 启用压缩
- 启用解压缩
- 发送压缩文件

1. 压缩和解压缩介绍

压缩响应通常会显着减少传输数据的大小。 然而，由于压缩在运行时发生，它还可以增加相当大的处理开销，这会对性能产生负面影响 在向客户端发送响应之前，NGINX 会执行压缩，但不会“压缩”已压缩的响应(例如，由代理的服务器)。

2. 启用压缩

要启用压缩，请使用包含 gzip 指令并指定 on 值。

```
gzip on;
```

默认情况下，NGINX 仅使用 MIME 类型 text/html 压缩响应。要使用其他 MIME 类型压缩响应，请包含 gzip_types 指令并列出其他类型。

```
gzip_types text/plain application/xml;
```

要指定要压缩的响应的最小长度，请使用 gzip_min_length 指令。 默认值为 20 字节(可将此处调整为 1000)：

```
gzip_min_length 1000;
```

默认情况下，NGINX 不会压缩对代理请求的响应(来自代理服务器的请求)。 请求来自代理服务器的事实由请求中 Via 头字段的存在确定。 要配置这些响应的压缩，请使用 gzip_proxied 指令。 该指令具有多个参数，指定 NGINX 应压缩哪种代理请求。例如，仅对不会在代理服务器上缓存的请求压缩响应是合理的。 为此，gzip_proxied 指令具有指示 NGINX 在响应中检查 Cache-Control 头字段的参数，如果值为 no-cache,no-store 或 private，则压缩响应。 另外，您必须包括 Expires 参数以用来检查 Expires 头域的值。 这些参数在以下示例中与 auth 参数一起设置，该参数检查 Authorization 头字段的存在(授权响应特定于最终用户，并且通常不被缓存)：

```
gzip_proxied no-cache no-store private expired auth;
```

与大多数其他指令一样,配置压缩的指令可以包含在 http 上下文中,也可以包含在 server 或 location 配置块中。

gzip 压缩的整体配置可能如下所示。

```
server {
    gzip on;
    gzip_types      text/plain application/xml;
    gzip_proxied    no-cache no-store private expired auth;
    gzip_min_length 1000;
    ...
}
```

3. 启用解压缩

某些客户端不支持使用 gzip 编码方法的响应。 同时，可能需要存储压缩数据，或者即时压缩响应并将它们存储在缓存中。 为了成功地服务于不接受压缩数据的客户端，NGINX 可以在将数据发送到后一种类型的客户端时即时解压缩数据。

要启用运行时解压缩，请使用 gunzip 指令。

```
location /storage/ {
    gunzip on;
    ...
}
```

gunzip 指令可以在与 gzip 指令相同的上下文中指定：

```
server {
```

```
    gzip on;
    gzip_min_length 1000;
    gunzip on;
    ...
}
```

请注意，此指令在单独的模块中定义，默认情况下可能不包含在开源 NGINX 构建中。

4. 发送压缩文件

要将文件的压缩版本发送到客户端而不是常规文件，请在适当的上下文中将 gzip_static 指令设置为 on。

```
location / {
    gzip_static on;
}
```

在这种情况下，为了服务/path/to/file 的请求，NGINX 尝试查找并发送文件/path/to/file.gz。 如果文件不存在，或客户端不支持 gzip，则 NGINX 将发送未压缩版本的文件。

请注意，gzip_static 指令不启用即时压缩。它只是使用压缩工具预先压缩的文件。要在运行时即时压缩内容(而不仅仅是静态内容)，请使用 gzip 指令。

该指令在单独的模块中定义，默认情况下可能不包含在开源 NGINX 构建中。

Nginx 内容缓存

介绍如何启用和配置从代理服务器接收的响应的缓存。主要涉及以下内容:

- 缓存介绍
- 启用响应缓存
- 涉及缓存的 NGINX 进程
- 指定要缓存的请求
- 限制或绕过缓存
  - 从缓存中清除内容
  - 配置缓存清除
  - 发送清除命令
  - 限制访问清除命令
  - 从缓存中完全删除文件
- 缓存清除配置示例
- 字节缓存
- 组合配置示例

1. 介绍

当启用缓存时，NGINX 将响应保存在磁盘缓存中，并使用它们来响应客户端，而不必每次都为同一内容代理请求。

2. 启用响应缓存

要启用缓存，请在顶层的 http 上下文中包含 proxy_cache_path 指令。 强制的第一个参数是缓存内容的本地文件系统路径，强制 keys_zone 参数定义用于存储有关缓存项目的元数据的共享内存区域的名称和大小：

```
http {
    ...
    proxy_cache_path /data/nginx/cache keys_zone=one:10m;
}
```

然后在要缓存服务器响应的上下文(协议类型，虚拟服务器或位置)中包含 proxy_cache 指令，将由 keys_zone 参数定义的区域名称指定为 proxy_cache_path 指令(在本例中为一)：

```
http {
    ...
    proxy_cache_path /data/nginx/cache keys_zone=one:10m;

    server {
        proxy_cache one;
        location / {
            proxy_pass http://localhost:8000;
        }
    }
}
```

请注意，由 keys_zone 参数定义的大小不会限制缓存的响应数据的总量。 缓存响应本身存储在文件系统上的特定文件中的元数据副本。要限制缓存的响应数据量，请将 max_size 参数包含到 proxy_cache_path 指令中(但请注意，缓存数据的数量可能会临时超出此限制，如以下部分所述。)

3. 涉及缓存的 NGINX 进程

缓存中还有两个额外的 NGINX 进程：

缓存管理器周期性地被激活以检查缓存的状态。 如果缓存大小超过了由 max_cize_path 指令设置的 max_size 参数，缓存管理器将删除最近访问的数据。如前所述，高速缓存管理器激活之间的缓存数据量可以临时超过限制。

NGINX 启动后，缓存加载程序只运行一次。 它将有关以前缓存的数据的元数据加载到共享

内存区域。一次加载整个缓存可能会在启动后的最初几分钟内消耗足够的资源来减慢 NGINX 的性能。 为了避免这种情况，请通过将以下参数包含到 proxy_cache_path 伪指令来配置缓存的迭代加载：

- loader_threshold - 迭代的持续时间，以毫秒为单位(默认为 200)
- loader_files - 在一次迭代期间加载的最大项目数(默认为 100)
- loader_sleeps - 迭代之间的延迟(以毫秒为单位)(默认为 50)

在以下示例中，迭代持续 300 毫秒或直到加载了 200 个项目：

```
proxy_cache_path /data/nginx/cache keys_zone=one:10m
loader_threshold=300 loader_files=200;
```

4. 指定要缓存的请求

默认情况下，NGINX 首次从代理服务器接收到这样的响应后，缓存对 HTTP GET 和 HEAD 方法的请求的所有响应。 作为请求的密钥(标识符)，NGINX 使用请求字符串。 如果请求具有与缓存响应相同的密钥，则 NGINX 将缓存的响应发送给客户端。 您可以在 http, server, 或 location 上下文中包含各种指令，以控制哪些响应被缓存。

要更改在计算密钥时使用的请求字符，请包括 proxy_cache_key 伪指令：

```
proxy_cache_key "$host$request_uri$cookie_user";
```

要定义在缓存响应之前必须进行具有相同密钥的请求的最小次数，请包括 proxy_cache_min_uses 指令：

```
proxy_cache_min_uses 5;
```

要使用除 GET 和 HEAD 之外的方法来缓存对请求的响应，请将它们与 GET 和 HEAD 一起列为 proxy_cache_methods 伪指令的参数：

```
proxy_cache_methods GET HEAD POST;
```

5. 限制或绕过缓存

默认情况下，响应将无限期地保留在缓存中。 只有缓存超过最大配置大小，然后按照最后一次请求的时间长度，它们才被删除。 您可以通过在 http, server, 或 location 上下文中包含指令来设置缓存响应被认为有效的时间长度，甚至是否使用它们。

要限制缓存响应与特定状态代码被认为有效的时间，请包括 proxy_cache_valid 指令：

```
proxy_cache_valid 200 302 10m;
proxy_cache_valid 404      1m;
```

在此示例中，使用代码 200 或 302 的响应有效时间为 10 分钟，并且代码 404 的响应有效 1 分钟。 要定义具有所有状态代码的响应的有效时间，请指定 any 作为第一个参数：

```
proxy_cache_valid any 5m;
```

要定义 NGINX 不会向客户端发送缓存响应的条件，请包括 proxy_cache_bypass 指令。每个参数定义一个条件并由多个变量组成。如果至少有一个参数不为空，并且不等于“0”(零)，则 NGINX 不会在缓存中查找响应，而是将请求立即转发到后端服务器。

```
proxy_cache_bypass $cookie_nocache $arg_nocache$arg_comment;
```

要定义 NGINX 根本不缓存响应的条件，请包括 proxy_no_cache 指令，以与 proxy_cache_bypass 伪指令相同的方式定义参数。

```
proxy_no_cache $http_pragma $http_authorization;
```

6. 从缓存中清除内容

NGINX 可以从缓存中删除过期的缓存文件。删除过期的缓存内容以防止同时提供旧版本和新版本的网页。 在接收到包含自定义 HTTP 头或“PURGE” HTTP 方法的特殊“purge”请求时，缓存被清除。

6.1 配置缓存清除

我们设置一个配置来标识使用“PURGE” HTTP 方法的请求并删除匹配的 URL。
在 http 块层级上，创建一个新变量，例如：$purge_method，这将取决于$request_method 变量：

```
http {
    ...
    map $request_method $purge_method {
        PURGE 1;
        default 0;
    }
}
```

在 location 中配置高速缓存，包括指定缓存清除请求的条件的 proxy_cache_purge 指令。
在我们的例子中，它是在上一步配置的$purge_method：

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

 location / {
     proxy_pass  http://localhost:8002;
     proxy_cache mycache;

     proxy_cache_purge $purge_method;
 }
}
```

6.3 发送清除命令

配置 proxy_cache_purge 指令后，您需要发送一个特殊的缓存清除请求来清除缓存。 您可以使用一系列工具发出清除请求，例如 curl 命令：

```
$ curl -X PURGE -D - "http://www.example.com/*"
HTTP/1.1 204 No Content
Server: nginx/1.5.7
Date: Sat, 01 Dec 2015 16:33:04 GMT
Connection: keep-alive
```

在该示例中，具有公共 URL 部分(由星号通配符指定)的资源将被删除。 但是，这些高速缓

存条目将不会从缓存中完全删除：它们将保留在磁盘上，直到它们被删除为非活动状态(proxy_cache_path 的非活动参数)，或由缓存清除程序进程处理，或客户端尝试访问它们 。

6.4 限制访问清除命令

建议您配置允许发送缓存清除请求的有限数量的 IP 地址：

```
geo $purge_allowed {
   default         0;  # deny from other
   10.0.0.1        1;  # allow from localhost
   192.168.0.0/24  1;  # allow from 10.0.0.0/24
}

map $request_method $purge_method {
   PURGE   $purge_allowed;
   default 0;
}
```

在这个例子中，NGINX 检查请求中是否使用“PURGE”方法，如果是，分析客户端 IP 地址。 如果 IP 地址被列入白名单，那么$purge_method 设置为$purge_allowed：“1”允许清除，“0”表示清除。

6.5 从缓存中完全删除文件

要完全删除与星号相匹配的缓存文件，您将需要激活一个特殊的缓存清除程序，该过程将永久地遍历所有缓存条目，并删除与通配符相匹配的条目。 在 http 块级别上，将 purger 参数添加到 proxy_cache_path 指令中：

```
proxy_cache_path          /data/nginx/cache          levels=1:2
keys_zone=mycache:10m purger=on;
```

6.6 缓存清除配置示例

```
http {
    ...
    proxy_cache_path        /data/nginx/cache        levels=1:2
keys_zone=mycache:10m purger=on;

    map $request_method $purge_method {
        PURGE 1;
        default 0;
    }

    server {
```

```
        listen      80;
        server_name www.example.com;

        location / {
            proxy_pass        http://localhost:8002;
            proxy_cache       mycache;
            proxy_cache_purge $purge_method;
        }
    }

    geo $purge_allowed {
       default         0;
       10.0.0.1        1;
       192.168.0.0/24  1;
    }

    map $request_method $purge_method {
       PURGE   $purge_allowed;
       default 0;
    }
}
```

7. 字节缓存

有时，初始缓存填充操作可能需要一些时间，特别是对于大文件。 当第一个请求开始下载视频文件的一部分时，下一个请求将不得不等待整个文件被下载并放入高速缓存。

NGINX 使缓存这样的范围请求成为可能，并逐渐用缓存片模块填充高速缓存。 该文件分为较小的“切片”。 每个范围请求选择将覆盖所请求范围的特定切片，并且如果此范围仍未缓存，请将其放入缓存中。 对这些切片的所有其他请求将从缓存中获取响应。

要启用字节范围缓存：

确保您的 NGINX 是使用 slice 模块编译的。

使用 slice 指令指定切片的大小：

```
location / {
 slice  1m;
}
```

slice 的大小应适当调整，使切片快速下载。 在处理请求时，太小的大小可能会导致内存使用量过多和大量打开的文件描述符，太大的值可能会导致延迟。

将$slice_range 变量包含到缓存键中：

```
proxy_cache_key $uri$is_args$args$slice_range;
```

启用使用 206 状态代码缓存响应：

```
proxy_cache_valid 200 206 1h;
```

通过在 Range 头字段中传递$slice_range 变量来将传递范围请求设置为代理服务器：

```
proxy_set_header  Range $slice_range;
```

字节范围缓存示例：

```
location / {
    slice            1m;
    proxy_cache       cache;
    proxy_cache_key   $uri$is_args$args$slice_range;
    proxy_set_header  Range $slice_range;
    proxy_cache_valid 200 206 1h;
    proxy_pass        http://localhost:8000;
}
```

请注意，如果切片(slice)缓存打开，则不应更改初始文件。

8. 组合配置示例

以下示例配置组合了上述某些缓存选项。

```
http {
    ...
    proxy_cache_path         /data/nginx/cache        keys_zone=one:10m
loader_threshold=300
                     loader_files=200 max_size=200m;

    server {
        listen 8080;
        proxy_cache one;

        location / {
            proxy_pass http://backend1;
        }

        location /some/path {
            proxy_pass http://backend2;
            proxy_cache_valid any 1m;
            proxy_cache_min_uses 3;
            proxy_cache_bypass $cookie_nocache $arg_nocache$arg_comment;
        }
    }
}
```

在这个例子中，两个位置使用相同的缓存，但是以不同的方式。

由于 backend1 服务器的响应很少更改，因此不包括缓存控制指令。 首次请求响应缓存，并无限期保持有效。

相比之下，对 backend2 服务的请求的响应频繁变化，因此它们被认为只有 1 分钟有效，并且在相同请求 3 次之前不被缓存。 此外，如果请求符合 proxy_cache_bypass 指令定义的条件，则 NGINX 会立即将请求传递给 backend2，而不在缓存中查找。

Nginx 配置日志

如何在 NGINX 中配置日志记录错误和处理的请求。将涉及以下内容

- 设置错误日志
- 设置访问日志
- 启用条件日志记录
- 日志记录到 Syslog

1. 设置错误日志

NGINX 将遇到的不同严重性级别问题的信息写入错误日志。 error_log 指令将日志记录设置为特定文件，stderr 或 syslog，并指定要记录的消息的最低级别。 默认情况下，错误日志位于{NGING_INSTALL_PATH}/logs/error.log(绝对路径取决于操作系统和安装)，并记录来自所指定的所有严重级别的消息。

以下配置将错误消息的最小严重性级别更改为从错误记录到警告：

```
error_log logs/error.log warn;
## 或者可写为： error_log /var/logs/nginx/error.log warn;
```

在这种情况下，将记录 warn,error crit,alert 和 emerg 级别的消息。

错误日志的默认设置全局工作。 要覆盖它，将 error_log 指令放在 main(顶级)配置上下文中。main 上下文中的设置始终由其他配置级别继承。 还可以在 http,stream,server 和 location 级别指定 error_log 指令，并覆盖从较高级别继承的设置。如果发生错误，则该消息只写入一个错误日志，最接近发生错误的级别的错误日志。 但是，如果在同一级别指定了多个 error_log 伪指令，则会将消息写入所有指定的日志。

注意：在开源 NGINX 版本 1.5.2 中添加了在同一配置级别指定多个 error_log 伪指令的功能。

2. 设置访问日志

在处理请求之后，NGINX 在访问日志中写入有关客户端请求的信息。 默认情况下，访问日志位于{NGING_INSTALL_PATH}logs/access.log 中，{NGING_INSTALL_PATH}为安装 nginx 的目录，信息以预定义的组合格式写入日志。要覆盖这个默认设置，请使用 log_format 指令更改记录消息的格式，以及 access_log 指令，以指定日志的位置及其格式。日志格式使用变量定义。

以下示例定义扩展预定义组合格式的日志格式，其值指示响应 gzip 的压缩比。 然后将格式应用于启用压缩的虚拟服务器。

```
http {
    log_format compression '$remote_addr - $remote_user [$time_local] '
                           '"$request" $status $body_bytes_sent '
                           '"$http_referer"             "$http_user_agent"
"$gzip_ratio"';

    server {
```

```
        gzip on;
        access_log /spool/logs/nginx-access.log compression;
        ...
    }
}
```

以下是一些如何读取生成时间值的规则：

- 当通过多个服务器处理请求时，变量包含由逗号分隔的多个值。
- 当从一个上游组到另一个上游组有内部重定向时，这些值以分号分隔。
- 当请求无法到达上游服务器或无法接收到完整报头时，变量包含“0”(零)。
- 在连接到上游或从缓存中获取回复时出现内部错误的情况下，该变量包含“-”(连字符)。

可以通过启用缓冲区的日志消息和名称包含变量的常用日志文件的描述符缓存来优化日志记录。 要启用缓冲，请使用 access_log 指令的缓冲区参数来指定缓冲区的大小。 当下一个日志消息不适合缓冲区以及其他情况时，缓冲的消息将被写入日志文件。

要启用日志文件描述符的缓存，请使用 open_log_file_cache 指令。与 error_log 指令类似，在特定配置级别定义的 access_log 伪指令将覆盖以前级别的设置。 当请求的处理完成时，消息将被写入到当前级别上配置的日志中，或者从先前的级别继承。 如果一个级别定义了多个访问日志，则会将消息写入所有的访问日志。

3. 启用条件日志记录

条件记录允许从访问日志中排除琐碎或不重要的日志条目。在 NGINX 中，条件日志记录由 access_log 伪指令的 if 参数启用。

此示例不包括使用 HTTP 状态代码 2xx(成功)和 3xx(重定向)的请求：

```
map $status $loggable {
    ~^[23]  0;
    default 1;
}

access_log /path/to/access.log combined if=$loggable;
```

4. 日志记录到 Syslog

syslog 实用程序是计算机消息记录的标准，并允许从单个 syslog 服务器上的不同设备收集日志消息。 在 NGINX 中，对 syslog 的日志记录使用 error_log 和 access_log 伪指令中的 syslog：前缀进行配置。

Syslog 消息可以发送到服务器=可以是域名，IP 地址或 UNIX 域的套接字路径。 可以使用端口指定域名或 IP 地址来覆盖默认端口 514. 可以在 unix：prefix 之后指定 UNIX 域套接字路径：

```
error_log server=unix:/var/log/nginx.sock debug;
access_log
syslog:server=[2001:db8::1]:1234,facility=local7,tag=nginx,sev
erity=info;
```

在该示例中，NGINX 错误日志消息将在调试日志记录级别写入 UNIX 域套接字，并将访

问日志写入具有 IPv6 地址和端口 1234 的 syslog 服务器。

facility =参数指定正在记录消息的程序类型。 默认值为 local7。 其他可能的值是：auth，authpriv，daemon，cron，ftp，lpr，kern，mail，news，syslog，user，uucp，local0 … local7。

tag =参数将自定义标签应用于 syslog 消息(在我们的示例中为 nginx)。

severity =参数设置访问日志的 syslog 消息的严重性级别。 严重性越来越高的可能值为：debug，info，notice，warn，error(default)，crit，alert 和 emerg。 消息记录在指定的级别和更严重的级别。在我们的示例中,严重性级别错误还使得可以 crit, alert 和 emerg 级别。

Nginx 主要应用场景

只针对 Nginx 在不加载第三方模块的情况能处理哪些事情，由于第三方模块太多所以也介绍不完，当然本文本身也可能介绍的不完整，这里是根据个人使用过和了解到过总结出来的。从以下四个方面：

- 反向代理
- 负载均衡
- HTTP 服务器(包含动静分离)
- 正向代理

以上就是我了解到的 Nginx 在不依赖第三方模块能处理的事情，下面详细说明每种功能怎么做

1. 反向代理

反向代理应该是 Nginx 做的最多的一件事了，什么是反向代理呢，反向代理(Reverse Proxy)方式是指以代理服务器来接受 internet 上的连接请求,然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给 internet 上请求连接的客户端，此时代理服务器对外就表现为一个反向代理服务器。简单来说就是真实的服务器不能直接被外部网络访问，所以需要一台代理服务器，而代理服务器能被外部网络访问的同时又跟真实服务器在同一个网络环境，当然也可能是同一台服务器，端口不同而已。

下面贴上一段简单的实现反向代理的代码：

```
server {
        listen       80;
        server_name  localhost;
        client_max_body_size 1024M;

        location / {
            proxy_pass http://localhost:8080;
            proxy_set_header Host $host:$server_port;
        }
```

```
}
```

保存配置文件后启动 Nginx，这样当我们访问 localhost 的时候，就相当于访问 localhost:8080 了。

2. 负载均衡

负载均衡也是 Nginx 常用的一个功能，负载均衡其意思就是分摊到多个操作单元上进行执行，例如 Web 服务器、FTP 服务器、企业关键应用服务器和其它关键任务服务器等，从而共同完成工作任务。简单而言就是当有 2 台或以上服务器时，根据规则随机的将请求分发到指定的服务器上处理，负载均衡配置一般都需要同时配置反向代理，通过反向代理跳转到负载均衡。而 Nginx 目前支持自带 3 种负载均衡策略，还有 2 种常用的第三方策略。

1、RR(默认)

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

```
    upstream test {
        server localhost:8080;
        server localhost:8081;
    }
    server {
        listen       81;
        server_name  localhost;
        client_max_body_size 1024M;

        location / {
            proxy_pass http://test;
            proxy_set_header Host $host:$server_port;
        }
    }
```

负载均衡的核心代码为

```
    upstream test {
        server localhost:8080;
        server localhost:8081;
    }
```

这里配置了 2 台服务器，当然实际上是一台，只是端口不一样而已，而 8081 的服务器是不存在的,也就是说访问不到，但是我们访问 http://localhost 的时候,也不会有问题，会默认跳转到 http://localhost:8080 具体是因为 Nginx 会自动判断服务器的状态，如果服务器处于不能访问(服务器挂了)，就不会跳转到这台服务器，所以也避免了一台服务器挂了影响使用的情况，由于 Nginx 默认是 RR 策略，所以我们不需要其他更多的设置。

2、权重

指定轮询几率，weight 和访问比率成正比，用于后端服务器性能不均的情况。 例如

```
    upstream test {
        server localhost:8080 weight=9;
        server localhost:8081 weight=1;
    }
```

那么 10 次一般只会有 1 次会访问到 8081，而有 9 次会访问到 8080

3、ip_hash

上面的 2 种方式都有一个问题，那就是下一个请求来的时候请求可能分发到另外一个服务器，当我们的程序不是无状态的时候(采用了 session 保存数据)，这时候就有一个很大的很问题了，比如把登录信息保存到了 session 中，那么跳转到另外一台服务器的时候就需要重新登录了，所以很多时候我们需要一个客户只访问一个服务器，那么就需要用 iphash 了，iphash 的每个请求按访问 ip 的 hash 结果分配，这样每个访客固定访问一个后端服务器，可以解决 session 的问题。

```
    upstream test {
        ip_hash;
        server localhost:8080;
        server localhost:8081;
    }
```

4、fair(第三方)

按后端服务器的响应时间来分配请求，响应时间短的优先分配。

```
 upstream backend {
        fair;
        server localhost:8080;
        server localhost:8081;
    }
```

5、url_hash(第三方)

按访问 url 的 hash 结果来分配请求，使每个 url 定向到同一个后端服务器，后端服务器为缓存时比较有效。 在 upstream 中加入 hash 语句，server 语句中不能写入 weight 等其他的参数，hash_method 是使用的 hash 算法

```
    upstream backend {
        hash $request_uri;
        hash_method crc32;
        server localhost:8080;
```

```
        server localhost:8081;
    }
```

以上 5 种负载均衡各自适用不同情况下使用，所以可以根据实际情况选择使用哪种策略模式,不过 fair 和 url_hash 需要安装第三方模块才能使用，由于本文主要介绍 Nginx 能做的事情，所以 Nginx 安装第三方模块不会再本文介绍

3. HTTP 服务器

Nginx 本身也是一个静态资源的服务器，当只有静态资源的时候，就可以使用 Nginx 来做服务器，同时现在也很流行动静分离，就可以通过 Nginx 来实现，首先看看 Nginx 做静态资源服务器

```
    server {
        listen       80;
        server_name  localhost;
        client_max_body_size 1024M;
        location / {
              root   /usr/local/nginx/html/www;
              index  index.html;
           }
    }
```

这样如果访问 http://localhost 就会默认访问到 html/www 目录下面的 index.html，如果一个网站只是静态页面的话，那么就可以通过这种方式来实现部署。

4. 动静分离

动静分离是让动态网站里的动态网页根据一定规则把不变的资源和经常变的资源区分开来，动静资源做好了拆分以后，我们就可以根据静态资源的特点将其做缓存操作，这就是网站静态化处理的核心思路。

```
upstream test{
    server localhost:8080;
    server localhost:8081;
    }

    server {
        listen       80;
        server_name  localhost;

        location / {
            root   root   /usr/local/nginx/html/www;
            index  index.html;
        }

        # 所有静态请求都由 nginx 处理，存放目录为 html
```

```
        location ~ .(gif|jpg|jpeg|png|bmp|swf|css|js)$ {
            root    /var/www;
        }

        # 所有动态请求都转发给 tomcat 处理
        location ~ .(jsp|do)$ {
            proxy_pass  http://test;
        }

        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   /var/www;
        }
    }
```

这样我们就可以吧 HTML 以及图片和 css 以及 js 放到 html/www 目录下，而 tomcat 只负责处理 jsp 和请求，例如当我们后缀为 gif 的时候，Nginx 默认会从 html/www 获取到当前请求的动态图文件返回，当然这里的静态文件跟 Nginx 是同一台服务器，我们也可以在另外一台服务器，然后通过反向代理和负载均衡配置过去就好了，只要搞清楚了最基本的流程，很多配置就很简单了，另外 localtion 后面其实是一个正则表达式，所以非常灵活

5. 正向代理

正向代理，意思是一个位于客户端和原始服务器(origin server)之间的服务器，为了从原始服务器取得内容，客户端向代理发送一个请求并指定目标(原始服务器)，然后代理向原始服务器转交请求并将获得的内容返回给客户端。客户端才能使用正向代理。

```
    resolver 114.114.114.114 8.8.8.8;
    server {
        resolver_timeout 5s;
        listen 81;
        access_log  /var/www/access.log;
        error_log   /var/www/error.log;

        location / {
            proxy_pass http://$host$request_uri;
        }
    }
```

resolver 是配置正向代理的 DNS 服务器，listen 是正向代理的端口，配置好了就可以在 ie 上面或者其他代理插件上面使用服务器 ip+端口号进行代理了。

Nginx 是支持热启动的，也就是说当我们修改配置文件后，不用关闭 Nginx，就可以实现让配置生效，当然我并不知道多少人知道这个，反正我一开始并不知道，导致经常杀死了 Nginx 线程再来启动。。。Nginx 从新读取配置的命令是 -

```
$ nginx -s reload
```

windows 下面就是

```
$ nginx.exe -s reload
```