Broadview
www.broadview.com.cn

Nginx

高性能Web服务器详解

苗泽 编著

- 内容全面：知识点覆盖广泛，内容由浅入深，适合学习和查询。
- 实例丰富：每个知识点配有实例，重点知识配有经典案例，是入门和实战的必备参考。
- 深入浅出：阐述基本的概念和知识，深入源码级别的设计开发，总结经典的实战技巧。
- 实战技巧：依托实例讲解Web开发实战技巧，全面提升开发者实力。
- 实战讲解：使用专门章节展示和分析经典应用实例，巩固理论知识，增强实战技能。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview
www.broadview.com.cn
博文视点 · IT出版旗舰品牌
技术凝聚实力 · 专业创新出版

Nginx

高性能Web服务器详解

本书读者对象

- Web服务器使用者
- Nginx学习开发者
- Linux研发工作者
- 网络编程爱好者

重点内容

- Nginx的安装和基础配置、优化配置
- Rewrite功能和Gzip压缩
- 服务器的代理服务
- Nginx服务器的高级配置
- Nginx源码的目录结构
- 源码的模块化结构
- 基本数据结构
- 时间管理
- 内存管理
- 相关源码分析
- Nginx服务器进程间通信
- 模块编程
- Nginx在JSP网站建设中的使用
- Nginx在PHP网站建设中的使用
- Nginx+Perl脚本在网站建设中的使用
- Nginx经典应用——LNAMP

上架建议：计算机 > 网络

ISBN 978-7-121-21518-6

定价：59.00元

新浪微博 weibo.com @博文视点Broadview

责任编辑：徐津平
封面设计：侯士卿

Nginx
高性能Web服务器详解

苗泽 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书全面介绍了当前 Internet 上流行的一款开放源代码的 Web 服务器——Nginx。全书一共分为四大部分，分别从入门、功能、实现和应用等四个方面对 Nginx 服务器的知识进行完整阐述，从而满足广大读者在应用 Nginx 服务器时的普遍性需求。同时也深入剖析了 Nginx 服务器的工作原理和实现技术，对其中使用到的数据结构和方法进行了详细阐述，并且结合实际的应用情况给出了多个基于 Nginx 服务器，同时还部署有其他典型服务器的分布式网站架构部署配置。

本书特别适合于希望了解和掌握 Nginx 服务器应用技术和实现技术的广大教师、学生和电脑爱好者阅读，对使用 Nginx 服务器搭建 Web 服务器架构或进行网络服务器应用开发的技术人员尤其具有重要的阅读和参考价值。

图书在版编目（CIP）数据

Nginx 高性能 Web 服务器详解 / 苗泽编著. —北京：电子工业出版社，2013.10
ISBN 978-7-121-21518-6

Ⅰ. ①N… Ⅱ. ①苗… Ⅲ. ①互联网络—网络服务器 Ⅳ. ①TP368.5

中国版本图书馆 CIP 数据核字（2013）第 223454 号

责任编辑：徐津平
特约编辑：梁卫红
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：787 × 1092　　1 /16　　印张：19.5　　字数：499千字
印　　次：2013年10月第1次印刷
印　　数：3500册　　定价：59.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

随着电子技术的日益繁荣，信息网络的急速发展，世界的每一个角落和人类的点滴生活都产生了日新月异的变化。技术的发展可以推动社会的进步，而社会的进步又能支持新技术的产生和应用，两者相互促进，共同发展，推动了人类历史前进的脚步。

“信息高速公路”这一概念的提出，实现了全球范围内声、像、图、文等多媒体信息的高速传输和共享。World Wide Web 技术突破性的发展，解决了远程信息服务中的文字显示、数据连接以及图像传递的问题，其成为了 Internet 上最为流行的信息传播方式。Web 服务器作为提供网络信息浏览服务的终端，它已成为 Internet 上最大的计算机群，并为 Internet 的普及迈出了开创性的一步，是 Internet 上取得的最激动人心的成就之一。

为什么要写本书

Web 服务器技术发展到现在，已经不能仅仅考虑单一的网络信息浏览查询功能，随着信息数据的不断增长和用户体验需求的不断提高，其涉及的技术也趋于多元化。这些技术主要关心三个方面的问题：

一是服务器自身的处理性能。信息数据的急速增长、云计算和大数据理论的相继提出，使得对 Web 服务器的性能要求越来越高，虽然数据处理不是 Web 服务器的主要工作，但是如何能够更快地处理和响应客户端请求是 Web 服务器面临的关键问题之一。

二是服务器的稳定性。Internet 遍布全球，每时每刻都有大量的请求需要处理，一台优秀的 Web 服务器应该能提供持续的不间断服务，这对 Web 服务器程序来说是一个考验。如何保证 Web 服务器在大量并发请求出现的时候仍然稳定运行，在长时间运转过程中降低产生问题的概率，在产生问题之前能够有效预防，产生问题时能够有效避免数据丢失，在运行过程中减小对系统平台的资源压力，等等，这些也是 Web 服务器致力于解决的关键问题之一。

三是 Web 服务器的定制性。Web 服务器通常面向的用户是管理人员，能够在其中快速添加、删除和配置功能，并且实施步骤简单，部署迅速，这也是 Web 服务器应该具备的优点。

目前市场上比较流行的 Web 服务器主要有 Apache、Microsoft IIS、Sun、Nginx、Tomcat、Lighttpd 等。这些服务器各具特色但也各有缺点。Nginx 服务器可以说是 Web 服务器市场的一匹黑马，从 2002 年第一个版本发布，到现在进入高速发展的时期，其已经占据了一席之地，受到全球广大 Web 服务器使用者的青睐。由于 Nginx 服务器发展迅速，因此目前能够完整系统介绍 Nginx 服务器相关知识的书籍不多。本书以 Nginx 服务器为对象，从 Nginx 服务器的功能配置、源码、部署实践等三个方面，较为完整地阐述了 Nginx 服务器的应用实践和技术实现。

本书有什么内容

本书一共分为 4 大部分 18 个章节，分别从入门篇、功能篇、实现篇和应用篇等 4 个方面对 Nginx 服务器进行了阐述。

入门篇简单介绍了目前的 Web 服务器市场和 Nginx 服务器的诞生历史，主要对 Nginx 服务器的安装和基础配置、优化配置进行了详细的分析和说明，在这一部分也简单介绍了 Nginx 服务器的模块化结构，这一方面为我们学习后面功能篇中的内容进行知识准备，另一方面也为我们学习实现篇中的内容做一个铺垫。

功能篇从 Nginx 服务器常用的功能入手，分别介绍了如何配置 Nginx 服务器的 gzip 功能、rewrite 功能、proxy 功能、cache 功能和邮件服务功能，以帮助大家能够根据不同的实际需求对 Nginx 服务器功能模块进行定制。

实现篇深入到 Nginx 服务器的源码实现，对 Nginx 服务器的初始化启动、主进程和工作进程的功能实现源码进行了详细的分析，帮助大家更深一步理解 Nginx 服务器的运行过程和实现技术，在这一部分还简单地介绍了 Nginx 服务器模块编程的相关内容，这为大家进行 Nginx 服务器模块设计提供了思路。

应用篇主要列举了 Nginx 服务器在动态网站建设中的应用，同时也介绍了 Nginx 服务器应用中比较经典的一个架构——LNAMP 的配置部署，为大家展示了在实际应用中如何使用 Nginx 服务器提供 Web 服务的配置实例。

需要学习的基础知识

在学习本书之前，大家应该对以下的知识有一个简单的初步了解，这样更有利于对本书知识的学习和理解：

- Web 服务器的基本使用经验
- 网络编程的基础知识
- HTTP 协议的基础知识
- C 语言编程基础知识

本书的优势

- 轻松入门。本书以 Nginx 服务器的发展历史作为切入点，详细介绍了 Nginx 服务器的优势、基本概念、进阶技术等，内容由浅入深，是广大 Web 服务器工作者了解 Nginx 服务器的首选。
- 上手容易。本书集合了丰富的实例，尽可能网罗所有生产中使用的经验技巧，让读者能够快速上手。

- 深入浅出。本书从多个角度完整地讲述了 Nginx 服务器的各个方面，不仅包括了基本概念相关知识，更包括了 Nginx 服务器的使用技巧、深层次的源码架构等内容，让读者能够全面了解 Nginx 服务器。
- 实战讲解。本书在结束了理论讲解之后，会特意安排一部分内容向读者完整展示常见 Nginx 服务器应用实例，便于读者巩固前面各章节学习的理论知识。
- 问题集锦。在本书各章节的讲解过程中收录了笔者在实际生产过程中遇到或者收集到的大量 Nginx 服务器使用的问题及可能的解答，这有助于读者在实际操作中对问题的处理。

在撰写本书的过程中，我们一直努力为读者呈现一个较为完整的知识结构，尽力将关于 Nginx 服务器的使用、实践、实现等多方面的知识传输给大家，希望它们能够起到抛砖引玉的作用，为大家进一步理解和学习 Nginx 服务器的使用方法和设计精髓提供基本知识和思路。该书是我们在使用 Nginx 服务器的过程中的思考和学习记录，由于知识水平和应用水平有限，疏漏和错误之处在所难免，欢迎大家积极批评和指正。

作者

2013 年 7 月

目　录

第 1 章

Nginx 初探

Nginx 服务器是轻量级 Web 服务器中广受好评的一款产品。从本章我们开始 Nignx 服务器的学习和实践。

在本章中，我们主要探究 Nginx 服务器是什么，它在相关的行业领域内地位如何，它有哪些用途等问题。我们将追随 Nginx 服务器由诞生到快速发展的历史轨迹，了解 Nginx 服务器提供了哪些令人兴奋的功能和特性。

在这一章中我们主要学习以下几个方面的内容：

- 常见的 Web 服务器产品。
- Nginx 服务器的诞生和发展。
- Nginx 服务器的功能和特性。

1.1 Nginx 的历史

近几年来，Nginx 逐步进入高速发展的时期，从各类主流的 IT 媒体到各大著名的 IT 论坛，我们不时能够看到它的身影。

Netcraft 公司，1994 年在英国成立，官方网址为 http://uptime.netcraft.com。Netcraft 公司为互联网市场以及在线安全方面提供咨询服务，同时针对网站服务器、域名解析/主机提供商以及 SSL 市场进行客观严谨的分析研究。公司官方网站每月定期公布的 Web Server Survey 已成为了解全球网站及服务器市场份额情况的主要参考依据。

根据 Netcraft 公司在 2012 年 8 月收到的对 628 170 204 个网站的调查数据显示，使用 Nginx 服务器的网站的比例在不断攀升，其市场份额已由 7 月份的 11.45%进一步上升至 12.31%，并成为此次调

查中唯一份额增长的服务器产品。

Nginx 的成功要归功于它在设计之初就已经形成的不同于其他同类产品的设计理念和架构体系。那么，在 Nginx 服务器名不见经传的时候，Web 服务器市场的情况是怎样的呢？

1.2 常见服务器产品介绍

我们仍然以 Netcraft 公布的数据为基础，对常见主流服务器产品进行介绍。图 1.1 摘自 Netcraft 官方网站，其展示了 2012 年全球主流 Web 服务器的市场份额情况，其中有 Apache、Microsoft IIS、Sun、Nginx、Google 以及 NCSA 等。在接下来的各小节中，我们主要针对 Apache、Microsoft IIS、NCSA 等 Web 服务器产品进行介绍，同时还将补充介绍图中未提及的 Tomcat、Lighttpd 等 Web 服务器产品。

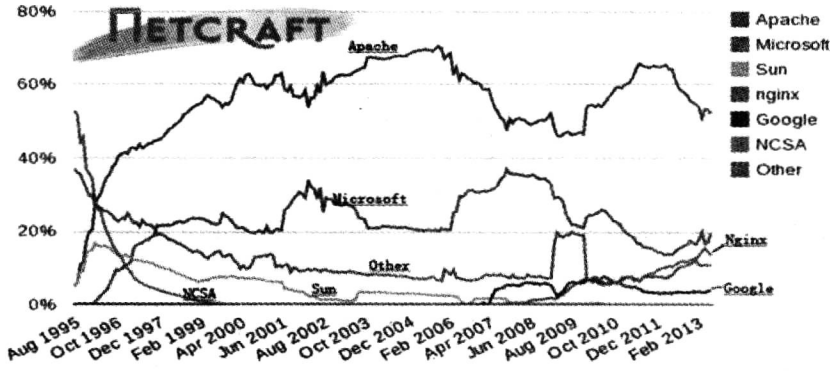

图 1.1 全球主流 Web 服务器市场份额（2012，来源：Netcraft）

1.2.1 Apache 服务器

相信大多数使用过 Web 服务器的人对 Apache 服务器都不会陌生。Apache，取自“a patchy server”的读音，意思是充满补丁的服务器。这一称呼隐含了 Apache 极富戏剧性的诞生历史。Apache HTTP Server 的官方网站为 http://httpd.apache.org。根据 Netcraft 公司公布的数据，截至目前，全球仍旧有超半数的活跃网站使用 Apache Web 服务器，其市场份额高达 55.46%。

Apache Web 服务器最初是由 NCSA httpd 1.3 服务器发展而来的，目前的最新版本为 Apache httpd 2.4.2。作为开源软件，不断有人为它开发新的功能、新的特性并修改原来的缺陷，使其逐渐形成跨平台能力强、安全性能高并且被业界广泛接受和使用的最流行的 Web 服务器软件。Apache Web 服务器在各种开源的 WWW 服务提供工具中特性最全，并且运行速度快，性能相对稳定，而且扩展功能丰富。

不可否认，Apache Web 服务器是当今 Web 服务器市场中的领军产品，但在具体的使用过程中，它仍然在某些方面表现不足。比如性能方面，Apache 在设计时使用了以“进程”为基础的结构。大家都知道，进程要比线程消耗更多的系统开支，这导致 Apache 在多处理器环境中性能有所下降。因此，在对一个 Apache Web 站点进行扩容时，通常是增加服务器或扩充群集节点而不是增加处理器。

1.2.2　Microsoft IIS

IIS 是 Microsoft 公司的 Web 服务器产品，其在全球 Web 服务器市场中占有相当大的市场份额，是被广泛采用的 Web 服务器之一。IIS，是 Internet Information Services 的缩写。最初 IIS 是 Windows NT 版本的可选包，随后内置在 Windows 2000、Windows XP Professional、Windows Server 2003 和 Windows Server 2008 中一起发行。最新的 IIS 7.5 已经和 Windows Server 2012 一起发行了。IIS 的官方网址为 http://www.iis.net，可以在此找到更多关于 IIS 功能特性的详细介绍。

IIS 具备 Web 服务器特性的同时还包含了 Gopher Server（Web 上一种信息查找系统）和 FTP Server，并且可以用于 HTTP Server、FTP Server、NNTP Server 或者 SMTP Server。我们通过 IIS 不仅仅可以发布网页，它还可以提供新闻服务、文件和应用程序服务、邮件服务等。同时 IIS 还包含一些有趣的扩展，比如环境编辑界面 FrontPage、具备全文检索功能的 Index Server 及具备多媒体功能的 Net Show 等。

IIS 和 Windows Server 相结合，使其在网络应用服务器的管理、可用性、可靠性、安全性、性能与可扩展性等方面都大为增强了。

为了进一步提高可靠性、安全性及可用性，从 6.0 版本以后，IIS 提供了全新的 IIS 架构，该架构采用新的内核监听模式和新的“应用程序隔离模式—工作进程隔离模式”，这增强了应用程序池、工作进程以及 Web 管理服务等方面的功能，同时增强了网络应用的开发与国际性支持。而从 7.0 版本开始，IIS 进一步强调了其模块特性，增加了 SQL 数据库管理器、日志报表可视化支持、对 IIS 设置提供完全控制的配置编辑器、请求过滤器、.NET 授权规则编辑器以及 FastCGI 机制等强有力的功能，这进一步增强了用户的可操作性和服务器的性能。

作为同一家公司的产品，IIS 和 Windows Server 组合可以提供可靠、高效、完整的网络服务器解决方案。当然，作为付费软件，IIS 部署成本高的缺点也是显而易见的。

1.2.3　Tomcat 服务器

Tomcat，在英文中是“公猫或其他雄性猫科动物”的意思。Tomcat 服务器最初是由 Sun 公司的软件构架师詹姆斯 • 邓肯 • 戴维森开发的，后来变为开源项目，并由 Sun 公司将其贡献给了 Apache 软件基金会。据说戴维森在最初接手 Tomcat 的开发项目时，希望将此项目以一个动物的名字命名。他希望这种动物能够自己照顾自己。最终，他将其命名为 Tomcat。

Tomcat 是 Sun 公司官方推荐的 Servlet 和 JSP 容器，在中小型系统和并发访问用户不是很多的场合下，其作为轻量级应用服务器，被广泛地使用。它是开发和调试 JSP 程序的首选。目前在 Apache Tomcat 官方网站上提供下载的最新版本为 7.0.29。

在一般的应用中，Tomcat 常作为 Apache 的扩展部分，为运行 JSP 页面和 Servlet 提供服务，独立的 Servlet 容器是 Tomcat 的默认模式。事实上，Tomcat 和 IIS、Apache 等 Web 服务器一样，具有处理 HTML 页面的功能，只是它处理静态 HTML 的能力不如 Apache 服务器。

Tomcat 服务器作为轻量级的服务器软件，无法满足复杂业务场景的要求，也没有复杂和丰富的功能；但 Tomcat 是免费开源的，且体积小，安装和部署都很方便，系统资源占用率低，是主要的 Servlet 和 JSP 容器，在这方面它比绝大多数的商业应用软件服务器要好。

1.2.4 Lighttpd 服务器

Lighttpd 服务器来自德国的一个开源轻量级 Web 服务器软件，它在 2004 年左右开始取得了高速发展。Lighttpd，是 Light footprint 和 httpd 的结合，读音同 Lighty。在 Lighttpd 的官方网站 http://www.lighttpd.net 中有这样一段介绍：

With a small memory footprint compared to other web-servers, effective management of the cpu-load, and advanced feature set lighttpd is the perfect solution for every server that is suffering load problems.

由此可见 Lighttpd 的名字就是此服务器设计理念的完美体现。

根据 Netcraft 曾经发布的数据调查显示，2007 年 1 月，全球使用 Lighttpd 的网站为 170 000 家，2 月这个数字就达到了 7 000 000，在短短的一个月内惊人地增长了 400%！当时，包括一些著名的网站，如 YouTube、Wikipedia、Meebo、Yahoo! Messenger、Windows Live Messenger、ICQ、AIM 等，以及国内的网易新闻、六间房、56.COM、豆瓣、新浪博客、迅雷在线、花瓣网等都使用它作为服务器软件。

Lighttpd 的急速发展得益于它专门针对高性能网站，提供了一套安全、快速、兼容性良好并且灵活的 Web Server 环境。同时，它具有非常低的内存开销、CPU 占用率低以及模块丰富等特点，支持 FastCGI、Output Compress（输出压缩）、URL 重写等绝大多数 Apache 具有的重要功能，是 Apache 的绝好替代者。

作为轻量级服务器，Lighttpd 与 Apache 等大型 Web 服务器软件相比，其在功能上存在不足和部分缺陷，比如 Proxy 功能不完善、对编码支持不完善等缺点。

1.2.5 Nginx 诞生记

Nginx（engine-x）是由 1994 年毕业于俄罗斯国立莫斯科鲍曼技术大学的 Igor Sysoev 为俄罗斯访问量居首的 Rambler.ru 站点（www.rambler.ru）设计开发的。开发工作从 2002 年开始，第一次正式公开发布是在 2004 年 10 月 4 日，版本号为 0.1.0。

Nginx 是一款免费开源的高性能 HTTP 服务器及反向代理服务器（Reverse Proxy）产品，同时，它还可以提供 IMAP/POP3 代理服务等功能。在实际的使用中，Nginx 还可以提供更多更丰富的功能，我们将在下一节介绍它的功能。

Nginx 的官方网站为 http://www.nginx.org，同时 Wiki 为 Nginx 开设了专门的介绍页面，链接为 http://wiki.nginx.org/Main。本文引述的部分背景信息来源于这两个网站。大家可以由此获取更多相关背景知识。

到目前为止，Nginx 已经在俄罗斯第一大网站 Rambler.ru 上运行了近 9 个年头。在这段时间中，Nginx 不断成长和发展，以其稳定的性能、丰富的功能集、低系统资源的消耗而逐渐被全球 Web 服务器使用者认可，在全球的市场份额节节攀升。根据 Wiki 的资料显示，目前全球活跃的网站中，有 12.18%（大约为 22 200 000 个）的网站是由 Nginx 提供服务。而根据上一节中 Netcraft 公布的全球主流 Web 服务器最新数据显示，Nginx 的发展势头仍然良好。

为什么 Nginx 会成为众多 Web 服务器产品中的后起之秀呢？Nginx 到底能给我们带来怎样不同寻常的服务呢？在回答这些问题之前，我们先来简单梳理一下 Nginx 的历史版本和更新，从中感受 Nginx

快速成长的历程。

1.2.6　版本变更大事记

Nginx 从 2004 年 10 月发布到如今，已经趋于成熟和完善。它之所以能够如此快速地发展成为全世界广大 Web 服务器使用者青睐的对象，很重要的一个原因是，它依靠软件开源优势，集合全球技术人员的智慧，快速修复缺陷，更新功能，优化设计。所有人员都可以在 Nginx 的官方网站 http://trac.nginx.org/nginx/browser 浏览版本库并获取源码，进行进一步的开发和修改。

根据官方的版本变更说明（http://nginx.org/en/CHANGES）可以看到，自诞生到现在，Nginx 一共经历了十多次较大更新，以及近 340 次版本变更，更新频率非常高。

引起 Nginx 版本变更的主要原因包括软件缺陷修正、功能优化以及新功能加入等。

需要指出的是，Nginx 发布伊始主要针对 Linux 平台。从 Nginx 7.0.69 开始，Nginx 官方开始提供 Windows 版本，版本号与对应 Linxu 版本的版本号相同。由于 Windows 平台的 Nginx 在性能和使用广泛程度上不如 Linux 平台的 Nginx，因此在本文的以后篇章中，如无特别说明，讲述内容主要针对 Linux 平台的 Nginx。

目前，官方将 Nginx 版本分为开发版本（Development version，最新版本为 nginx-1.3.4）、稳定版本（Stable version，最新版本为 nginx-1.2.3）以及过期版本三种。其中，开发版本主要用于 Nginx 软件项目的研发，稳定版本即可作为 Web 服务器投入商业应用。

注意

在本书以下的章节中，如无特别说明，“Nginx X.X.X”的说法都是指“Nginx 稳定版本 X.X.X”。如果是其他类型版本，笔者会注明。

经过逐步的改进，Nginx 已成为一款高性能、功能完善、性能稳定的服务器产品。下一节，将对 Nginx 的功能进行介绍，为大家展现其丰富的功能特性。

1.3　Nginx 的功能特性

Nginx 服务器以其功能丰富著称于世。它既可以作为 HTTP 服务器，也可以作为反向代理服务器或者邮件服务器；能够快速响应静态页面（HTML）的请求；支持 FastCGI、SSL、Virtual Host、URL Rewrite、HTTP Basic Auth、Gzip 等大量使用功能；并且支持更多的第三方功能模块的扩展。

本节我们来梳理 Nginx 服务器提供的基本功能和服务，了解它较为完善的功能体系。在本书的以后章节中，会对本节提到的重要技术从理论和应用两个层次进行更为详细的说明。

我们将 Nginx 提供的基本功能服务从大体上归纳为基本 HTTP 服务、高级 HTTP 服务和邮件服务等三大类。

- Nginx 提供基本 HTTP 服务，可以作为 HTTP 代理服务器和反向代理服务器，支持通过缓存加速访问，可以完成简单的负载均衡和容错，支持包过滤功能，支持 SSL 等。

- Nginx 提供高级 HTTP 服务，可以进行自定义配置，支持虚拟主机，支持 URL 重定向，支持网络监控，支持流媒体传输等。
- Nginx 作为邮件代理服务器是最早开发这个产品的目的之一，它支持 IMAP/POP3 代理服务功能，支持内部 SMTP 代理服务功能。

下面进一步对 Nginx 的这三类基础功能进行说明。

1.3.1 基本 HTTP 服务

在 Nginx 提供的基本 HTTP 服务中，主要包含以下功能特性：

- 处理静态文件（如 HTML 静态网页及请求）；处理索引文件以及支持自动索引。
- 打开并自行管理文件描述符缓存。
- 提供反向代理服务，并且可以使用缓存加速反向代理，同时完成简单负载均衡及容错。
- 提供远程 FastCGI 服务的缓存机制，加速访问，同时完成简单的负载均衡以及容错。
- 使用 Nginx 的模块化特性提供过滤器功能。Nginx 基本过滤器包括 gzip 压缩、ranges 支持、chunked 响应、XSLT、SSI 以及图像缩放等。其中，针对包含多个 SSI 的页面，经由 FastCGI 或反向代理，SSI 过滤器可以并行处理。
- 支持 HTTP 下的安全套接层安全协议 SSL。

这些功能特性的应用将在后续章节中详细阐述。

1.3.2 高级 HTTP 服务

在 Nginx 提供的高级 HTTP 服务中，主要包含以下功能特性：

- 支持基于名字和 IP 的虚拟主机设置，在以后的章节中大家将看到具体应用。
- 支持 HTTP/1.0 中的 KEEP-Alive 模式和管线（PipeLined）模型连接。
- 支持重新加载配置以及在线升级时，无须中断正在处理的请求。
- 自定义访问日志格式、带缓存的日志写操作以及快速日志轮转。后面章节会讨论其具体应用。
- 提供 3xx ~ 5xx 错误代码重定向功能。后面章节会讨论其具体应用。
- 支持重写（Rewrite）模块扩展。后面章节会讨论其具体应用。
- 支持 HTTP DAV 模块，从而为 Http WebDAV 提供 PUT、DELETE、MKCOL、COPY 以及 MOVE 方法。
- 支持 FLV 流和 MP4 流传输。
- 支持网络监控，包括基于客户端 IP 地址和 HTTP 基本认证机制的访问控制、速度限制、来自同一地址的同时连接数或请求数限制等。
- 支持嵌入 Perl 语言。

1.3.3　邮件代理服务

Nginx 提供邮件代理服务也是其基本开发需求之一，主要包含以下功能特性：

- 支持使用外部 HTTP 认证服务器重定向用户到 IMAP/POP3 后端，并支持 IMAP 认证方式（LOGIN、AUTH LOGIN/PLAIN/CRAM-MD5）和 POP3 认证方式（USER/PASS、APOP、AUTH LOGIN/PLAIN/CRAM-MD5）。
- 支持使用外部 HTTP 认证服务器认证用户后重定向连接到内部 SMTP 后端，并支持 SMTP 认证方式（AUTH LOGIN/PLAIN/CRAM-MD5）。
- 支持邮件代理服务下的安全套接层安全协议 SSL。
- 支持纯文本通信协议的扩展协议 STARTTLS。

由以上的功能列表可以看到，Nginx 作为邮件代理服务器也是具备完善功能的，本书在后面介绍 Nginx 的 Mail 模块时将进一步阐述它。

1.4　常用功能介绍

这部分在内容上和其他部分没有明显的连贯性，主要是根据笔者的实践经验，选择 Nginx 服务器最常涉及的几个功能对其进行进一步阐述，其目的是阐明相关概念，为我们以后的学习奠定良好的知识基础。

1.4.1　HTTP 代理和反向代理

代理服务和反向代理服务是Nginx服务器作为Web服务器的主要功能之一，尤其是反向代理服务，是应用十分广泛的功能。

在提供反向代理服务方面，Nginx 服务器转发前端请求性能稳定，并且后端转发与业务配置相互分离，配置相当灵活。在后面的章节我们可以看到，在进行 Nginx 服务器配置时，配置后端转发请求完全不用关心网络环境如何，可以指定任意的 IP 地址和端口号，或其他类型的链接、请求等。

Nginx 服务器的反向代理服务功能并不只有这些，它提供的配套功能相当丰富。首先，它支持判断表达式。通过使用正则表达式进行相关配置，可以实现根据不同的表达式，采取不同的转发策略，相关内容将在下一章节中详细阐述。其次，它对后端返回情况进行了异常判断，如果返回结果不正常，则重新请求另一台主机（即将前端请求转向另一后端 IP），并自动剔除返回异常的主机。它还支持错误页面跳转功能。

谈到 Nginx 服务器的反向代理功能的应用，就不能不提到它在解决网络负载、性能方面的一个出色功能，这也是 Nginx 服务器的另一个重要的能力——负载均衡。

1.4.2　负载均衡

负载均衡，一般包含两方面的含义。一方面是，将单一的重负载分担到多个网络节点上做并行处理，每个节点处理结束后将结果汇总返回给用户，这样可以大幅提高网络系统的处理能力；第二个方面的含义是，将大量的前端并发访问或数据流量分担到多个后端网络节点上分别处理，这样可以有效

减少前端用户等待响应的时间。Web 服务器、FTP 服务器、企业关键应用服务器等网络应用方面谈到的负载均衡问题，基本隶属于后一方面的含义。因此，Nginx 服务器的负载均衡主要是对大量前端访问和流量进行分流，以保证前端用户访问效率。可以说，在绝大多数的 Nginx 应用中，都会或多或少涉及它的负载均衡服务。

Nginx 服务器的负载均衡策略可以划分为两大类：即内置策略和扩展策略。内置策略主要包含轮询、加权轮询和 IP hash 三种；扩展策略主要通过第三方模块实现，种类比较丰富，常见的有 url hash、fair 等。

在默认情况下，内置策略会被编译进 Nginx 内核，使用时只需要在 Nginx 服务器配置中设置相关参数即可，我们在专门的章节中会详细阐述；扩展策略不会编译进 Nginx 内核，需要手动将第三方模块编译到 Nginx 内核。第三方模块的编译技术也留在后文专门讲解。下面简单介绍一下几种负载均衡策略的实现原理。

轮询策略比较简单，就是将每个前端请求按顺序（时间顺序或者排列次序）逐一分配到不同的后端节点上，对于出现问题的后端节点自动排除。加权轮询策略，顾名思义，就是在基本的轮询策略上考虑各后端节点接受请求的权重，指定各后端节点被轮询到的几率。加权轮询策略主要用于后端节点性能不均的情况。根据后端节点性能的实际情况，我们可以在 Nginx 服务器的配置文件中调整权值，使得整个网络对前端请求达到最佳的响应能力。

IP hash 策略，是将前端的访问 IP 进行 hash 操作，然后根据 hash 结果将请求分配给不同的后端节点。事实上，这种策略可以看作是一种特殊的轮询策略。通过 Nginx 的实现，每个前端访问 IP 会固定访问一个后端节点。这样做的好处是避免考虑前端用户的 session 在后端多个节点上共享的问题。

扩展策略中的 url hash 在形式上和 IP hash 相近，不同之处在于，IP hash 策略是对前端访问 IP 进行了 hash 操作，而 url hash 策略是对前端请求的 url 进行了 hash 操作。url hash 策略的优点在于，如果后端有缓存服务器，它能够提高缓存效率，同时也解决了 session 的问题；但其缺点是，如果后端节点出现异常，它不能自动排除该节点。在实际使用过程中笔者发现，后端节点出现异常会导致 Nginx 服务器返回 503 错误。

扩展的第三方模块 fair 则是从另一个角度来实现 Nginx 服务器负载均衡策略的。该模块将前端请求转发到一个最近负载最小的后台节点。那么，负载最小怎么判断呢？Nginx 通过后端节点对请求的响应时间来判断负载情况。响应时间短的节点负载相对就轻。得出判断结果后，Nginx 就将前端请求转发到选中的负载最轻的节点。

1.4.3 Web 缓存

相信不少读者对 Squid 有所了解。它在 Web 服务器领域中是一款相当流行的开源代理服务器和 Web 缓存服务器。作为网页服务器的前置缓存服务器，在很多优秀的站点中，它被用以缓存前端请求，从而提高 Web 服务器的性能；而且，它还可以缓存万维网、域名系统或者其他网络搜索等，为一个集体提供网络资源共享服务。

Nginx 服务器从 0.7.48 版本开始，也支持了和 Squid 类似的缓存功能。在本节中，我们主要介绍几个将在后文中涉及的重要知识点，具体的技术细节将放在专门的章节中详细讨论。

Nginx 服务器的 Web 缓存服务主要由 Proxy_Cache 相关指令集和 FastCGI_Cache 相关指令集构成。

其中，Proxy_Cache 主要用于在 Nginx 服务器提供反向代理服务时，对后端源服务器的返回内容进行 URL 缓存；FastCGI_Cache 主要用于对 FastCGI 的动态程序进行缓存。另外还有一款常用的第三方模块 ngx_cache_purge 也是 Nginx 服务器 Web 缓存功能中经常用到的。它主要用于清除 Nginx 服务器上指定的 URL 缓存。

到 Nginx 0.8.32 版本，Proxy_Cache 和 FastCGI_Cache 两部分的功能已经比较完善，再配合第三方的 ngx_cache_purge 模块，Nginx 服务器已经具备了 Squid 所拥有的 Web 缓存加速功能和清除指定 URL 缓存的功能；同时，Nginx 服务器对多核 CPU 的调度比 Squid 更胜一筹，性能高于 Squid，而在反向代理、负载均衡等其他方面，Nginx 也不逊于 Squid。这使得 Nginx 服务器可以同时作为负载均衡服务器和 Web 缓存服务器来使用，基本可以取代 Squid。

1.5　本章小结

本章是全书的第一章，首先介绍了几种市场占有率较高的服务器产品，其中有大型的 Web 服务器，如 Apache、IIS；也有轻量级的 Web 服务器，如 Lighttpd 等；由此引出后来居上的 Nginx 服务器。接着，我们回顾了 Nginx 服务器的发展历程，对它能够在短时间内异军突起的市场表现充满了好奇。究竟是什么因素使得 Nginx 服务器能够在激烈的 Web 服务器市场竞争中占领一席之地呢？为了找到原因，我们对 Nginx 服务器的功能特性进行了简单梳理，并且选取反向代理功能、负载均衡功能、Web 缓存服务三个功能点进行了进一步的阐述。

第 2 章

Nginx 服务器的安装部署

在开始 Nginx 学习旅程之前，需要说明的是，本书着眼于 Nginx 服务器的实际应用，我们希望通过对 Nginx 服务器的系统介绍，能够在最短时间内尽可能向大家全面展示 Nginx 服务器常见的应用场景；也希望为在实际开展和 Nginx 服务器相关工作和学习的过程中遇到问题的人员提供指导和参考。建议你在学习本书的同时，能够在我们的指导下亲自动手实践，达到事半功倍的效果。接下来，我们正式开始 Nginx 的学习！

本章首先为大家介绍 Nginx 服务器的基本安装部署。我们将在本章学习到以下知识：

- 获取 Nginx 服务器安装文件的途径。
- Nginx 服务器安装部署之前的准备工作。
- Windows 平台下 Nginx 服务器的安装部署。
- Linux 平台下 Nginx 服务器的编译和安装。
- 认识 Nginx 服务器的配置文件，以及如何进行基本配置。
- 初步学习通过优化 Nginx 配置，提高 Nginx 服务器的性能。
- 展示一个 Nginx 配置的完整实例。

2.1 如何获取 Nginx 服务器安装文件

Nginx 服务器的软件版本包括 Windows 版和 Linux 版两种，在官方网站上可以找到对应版本的下载链接。下面详细介绍如何获取所需版本的 Nginx 服务器软件。

2.1.1　获取新版本的 Nginx 服务器

Nginx 的官方下载网站为 http://nginx.org/en/download.html。打开网站，下载部分的内容如图 2.1 所示。可以看到，网页上提供了 Nginx 服务器三种版本的下载，分别是开发版本（Development version）、稳定版本（Stable version）和过期版本（Legacy versions）。三种版本的差异和选择在第 1 章中已经介绍过，这里不再赘述。本页提供下载的各类版本均为 Nginx 的较新版本。其中，开发版本同时也是 Nginx 全部版本中最新的版本。

下面分别介绍页面上下载部分各链接的具体含义：

“CHANGES-x.x”链接，记录的是对应版本的功能变更日志。包括新增功能、功能的优化和功能缺陷的修复等。

紧接着“CHANGES-x.x”链接后面的“nginx-x.x.x”链接，是 Nginx 服务器的 Linux 版本下载链接。右击链接，选择“另存为”命令或者选择下载专用工具就可以获取到 Nginx 服务器的 Linux 版本。得到的文件的后缀名为.tar.gz。

“pgp”链接，记录的是提供下载的版本使用 PGP 加密自由软件 GnuPG 计算后的签名。PGP 可以解释为 Pretty Good Privacy，是 PGP 公司的加密或签名工具套件。点击链接进入相关页面，可以查看 GnuPG 针对本下载版本的签名，以及执行本次计算的 GnuPG 软件版本号。这些数据可以用于下载文件的验证。

“nginx/Windows-x.x.x”链接，是 Nginx 服务器的 Windows 版本下载链接，下载方法和 Linux 版本相同。得到的文件的后缀名为.zip。

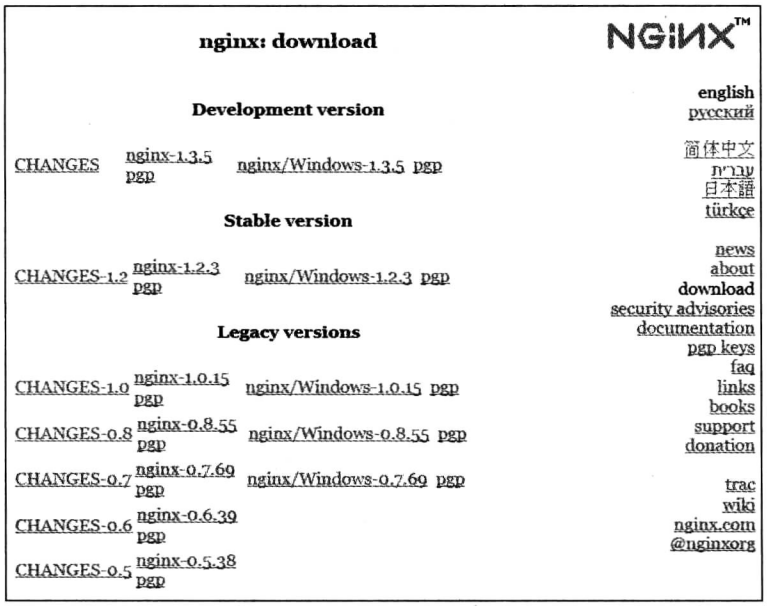

图 2.1　Nginx 的获取

2.1.2　获取 Nginx 服务器的历史版本

在第 1 章中，我们了解到 Nginx 服务器包含众多的历史版本。Nginx 官方网站没有显式提供下载

历史版本的链接，但是如果由于特殊的需求需要获取 Nginx 服务器的历史版本，该怎么做呢?

笔者为大家介绍一个可以下载 Nginx 服务器全部历史版本的链接，即 http://nginx.org/download。打开此网页，可以看到 Nginx 全部历史版本的列表，从 nginx-0.1.0 到 nginxnginx-1.3.5 一应俱全，下载方式和上面介绍的新版本 Nginx 服务器软件下载方式相同。

仔细查看版本列表，可以发现以 nginx-0.7.52 为分界线，之前的低版本只提供后缀名为.tar.gz 的 Linux 版本，从 nginx-0.7.52 开始同时提供后缀名为.zip 的 Windows 版本，出现这种现象的原因我们在第 1 章回顾 Nginx 服务器发展历史时已经说明。从 nginx-0.7.64 开始，还提供了后缀名为.asc 的文件，该文件记录了 Linux 版本 PGP 加密自由软件签名数据。

2.2　安装 Nginx 服务器和基本配置

上一节我们获取了自己需要的 Nginx 服务器版本。笔者使用的版本是 2012 年 8 月 7 日发布的稳定版本 nginx-1.2.3，包括 Windows 版本和 Linux 版本。本书后文中涉及的 Nginx 服务器使用、Nginx 服务器软件源代码分析均以此版本为基础。

本节主要指导大家在 Windows 平台和 Linux 平台上安装 Nginx 服务器软件。

2.2.1　Windows 版本的安装

Windows 版本的 Nginx 服务器安装方法与一般的 Windows 安装程序有所不同。

我们获取的 Windows 版本 Nginx 服务器安装文件为 nginx-1.2.3.zip 压缩文件。安装 Windows 版本的 Nginx 服务器不需要进行特殊的安装操作，使用解压工具解压此压缩文件后，得到如图 2.2 所示的文件资源，这就是 nginx 服务器运行的全部资源。

图 2.2　Windows 版本 Nginx 的安装文件资源

Windows 版本的 Nginx 服务器在效率上比 Linux 版本要差一些，并且 Nginx 在实际使用中一般用在 Linux/Unix 系统中，本文以 Nginx 1.2.3 版为学习重点进行说明。但为了方便后续的学习，我们还是有必要对解压出来的部分文件和目录做简单的介绍。

- conf 目录中存放的是 Nginx 服务器的配置文件，包含 Nginx 服务器的基本配置文件和对部分特性的配置文件。在本节的后文中，我们将重点介绍名为 nginx.conf 的配置文件如何使用。正确配置此文件可以保证 Nginx 服务器的正常运行。其他配置文件将在后续相关章节中陆续提及。

- docs 目录中存放了 Nginx 服务器的文档资料，包含 Nginx 服务器的 LICENSE、OpenSSL 的 LICENSE、PCRE 的 LICENCE 以及 zlib 的 LICENSE，还包括本版本 Nginx 服务器升级的版本变更说明，以及 README 文档。如果想了解 Nginx 服务器的具体细节，可以访问 Nginx 的官方网站 http://www.nginx.org。另外，http://wiki.nginx.org/Main 也是了解 Nginx 服务器相关信息的不错网站。
- html 目录中存放了两个后缀名为.html 的静态网页文件。这两个文件与 Nginx 服务器的运行相关，在后面介绍 Nginx 启动时将介绍这两个文件，在此不再赘述。
- logs 目录中存放了 Nginx 服务器的运行日志文件。我们将在讲述 Nginx 服务器配置时介绍它的日志功能，那里会涉及本目录的使用，在此也不再赘述。
- nginx.exe 即为启动 Nginx 服务器的运行程序。如果 conf 目录下的 nginx.conf 文件配置正确，通过它即可完成 nginx 服务器的启动操作。

2.2.2　Linux 版本的编译和安装：准备工作

Linux 版本 Nginx 服务器的安装比 Windows 版要麻烦一些，需要先对 Nginx 源代码进行编译。在正式开始操作之前，我们先检查 Nginx 编译和安装需要的条件是否满足。

在安装 Linux 版本的 Nginx 服务器之前，首先需要安装一款 Linux/UNIX 操作系统发行版，常见的有 Redhat、SUSE、Fedora、CentOS、Ubuntu、FreeBSD、Solaris 以及 Debian 等。在一些 Linux 发行版和 BSD 的衍生版本中自带了 Nginx 软件的二进制文件，但由于 Nginx 软件升级频繁，这些编译好的二进制文件大都比较陈旧，建议使用者直接从较新的源代码编译安装。

本文以 Fedora 16 为例，介绍 Nginx 的安装与使用。Fedora Linux 是比较具有知名度的 Linux 发行包之一，它基于 Red Hat Linux。在 Red Hat Linux 终止发行后，由 Fedora Project 社区开发、Redhat 公司赞助的 Fedora Linux，用来取代 Red Hat Linux 在个人领域的应用。

Nginx 服务器软件包（3.91 MB）和安装文件（4 MB 左右）一共需要不到 10 MB 的磁盘空间，实际情况可能因为编译设置和第三方模块的加入有所不同。目前，在不加入第三方模块的条件下，应该保证 10 MB 以上的磁盘空间。

为了编译 Nginx 源代码，我们需要标准的 GCC 编译器。GCC 的全称为 GNU Compiler Collection，其由 GNU 开发，并以 GPL 及 LGPL 许可证发行，是自由的类 UNIX 及苹果电脑 Mac OS X 操作系统的标准编译器。因为 GCC 原本只能处理 C 语言，所以原名为 GNU C 语言编译器，后来得到快速扩展，可处理 C++、Fortran、Pascal、Objective-C、Java 以及 Ada 等其他语言。

除此之外，我们还需要 Automake 工具，以完成自动创建 Makefile 的工作。

由于 Nginx 的一些模块需要依赖其他第三方库，通常有 pcre 库（支持 rewrite 模块）、zlib 库（支持 gzip 模块）和 openssl 库（支持 ssl 模块）等。

Fedora 的安装光盘中包含了以上提到的软件和第三方库，如果在安装 Fedora 时选择了安装以上软件，则可以略过；如果没有，可以进行在线安装：

```
yum -y install gcc gcc-c++ automake pcre pcre-devel zlib zlib-devel open openssl-devel
```

这里需要注意的是，我们不需要安装 Autoconf 工具。Nginx 软件的自动脚本不是用 Autoconf 工具

生成的，而是作者手工编写的。

到此，我们就完成了编译和安装 Nginx 服务器软件的环境准备工作。

2.2.3　Linux 版本的编译和安装：Nginx 软件的自动脚本

为了方便管理和使用，我们在文件系统的根目录“/”下新建 Nginx_123 目录，最后会把编译好的 Nginx 安装到此目录中。同时，在此目录中新建 Nginx_123_ Compile，用来编译 Nginx 软件：

```
#mkdir /Nginx_123/
```

将 2.1 节中获取的 Linux 版本的 nginx-1.2.3.tar.gz 复制到对应目录：

```
#cp nginx-1.2.3.tar.gz /Nginx_123/
```

解压 Nginx 归档，得到 Nginx 软件安装包的所有资源：

```
#tar xf nginx-1.2.3.tar.gz                                    //解压归档文件
#cd /Nginx_123/nginx-1.2.3
# ls -l
总用量 572
drwxr-xr-x. 6 1001 1001   4096 1月   7 10:45 auto
-rw-r--r--. 1 1001 1001 212690 11月 13 21:36 CHANGES
-rw-r--r--. 1 1001 1001 324135 11月 13 21:36 CHANGES.ru
drwxr-xr-x. 2 1001 1001   4096 1月   7 10:45 conf
-rwxr-xr-x. 1 1001 1001   2369 8月   7 2012 configure
drwxr-xr-x. 3 1001 1001   4096 1月   7 10:45 contrib
drwxr-xr-x. 2 1001 1001   4096 1月   7 10:45 html
-rw-r--r--. 1 1001 1001   1397 8月   6 2012 LICENSE
-rw-r--r--. 1 root root    374 1月   7 14:48 Makefile
drwxr-xr-x. 2 1001 1001   4096 1月   7 10:45 man
drwxr-xr-x. 3 root root   4096 1月   7 14:48 objs
-rw-r--r--. 1 1001 1001     49 10月 31 2011 README
drwxr-xr-x. 8 1001 1001   4096 1月   7 10:45 src
```

同样，为了方便后续的学习，我们有必要对解压出来的部分文件和目录做个简单的介绍。

- src 目录中存放了 Nginx 软件的所有源代码。本书的第二篇“Nginx 进阶”会对其做详细的介绍。
- man 目录中存放了 Nginx 软件的帮助文档，Nginx 安装完成后，在 Fedora 的命令行中使用 man 命令可以查看：

  ```
  #man nginx
  ```

- html 目录和 conf 目录中存放的内容和 Windows 版本的同名目录相同。
- auto 目录中存放了大量脚本文件，和 configure 脚本程序有关。
- configure 文件是 Nginx 软件的自动脚本程序。有过 Linux 软件编译经验的朋友应该对 configure 自动脚本程序有所了解。运行 configure 自动脚本一般会完成两项工作：一是检查环境，根据环境检查结果生成 C 代码；二是生成编译代码需要的 Makefile 文件。

进入 auto 目录，我们可以看到各种脚本资源。这些脚本职能划分清晰，有的检查环境（如 os 目录下的脚本），有的检查模块（如 modules 脚本），有的处理脚本参数（如 options 脚本），有的是用来

输出信息到生成文件的（如 have、nohave、make 及 install 等），还有的是为自动脚本本身服务（如 feature 脚本）的。前面已经提到，Nginx 软件的自动脚本是作者手工编写的，如果你在工作中需要编写自动脚本或者希望学习相关的内容，这个目录下的文件具有很高的参考价值。这些脚本文件没有涉及生僻的用法，可以仔细研究一下，在这里就不再赘述了。

CHANGES 文件、LICENSE 文件、README 文件和 Windows 版本 docs 目录下存放的同名文件相同。

Nginx 源代码的编译需要现使用 configure 脚本自动生成 Makefile 文件。在介绍生成 Makefile 文件操作之前，我们先介绍一下 configure 脚本支持的常用选项（见表 2.1）。

表 2.1　configure 脚本支持的常用选项

选项	说明
--prefix=<path>	指定 Nginx 软件的安装路径。此项如果未指定，默认为/usr/local/nginx/目录
--sbin-path=<path>	指定 Nginx 可执行文件安装路径。此项只能在安装时指定，如果未指定，默认为<prefix>/sbin/nginx/目录
--conf-path=<path>	在未给定-c 选项下，指定默认的 nginx.conf 路径。如果未指定，默认为<prefix>/conf/
--pid-path=<path>	在 nginx.conf 中未指定 pid 指令的情况下，指定默认的 nginx.pid 路径。如果未指定，默认为 <prefix>/logs/nginx.pid。nginx.pid 保存了当前运行的 Nginx 服务的进程号
--lock-path=<path>	指定 nginx.lock 文件的路径。nginx.lock 是 Nginx 服务器的锁文件，如果未指定，默认为/var/lock/目录
--error-log-path=<path>	在 nginx.conf 中未指定 error_log 指令的情况下，指定默认的错误日志的路径。如果未指定，默认为 <prefix>/logs/error.log
--http-log-path=<path>	在 nginx.conf 中未指定 access_log 指令的情况下，指定默认的访问日志的路径。如果未指定，默认为 <prefix>/logs/access.log
--user=<user>	在 nginx.conf 中未指定 user 指令的情况下，指定默认的 Nginx 服务器进程的属主用户，即 Nginx 进程运行的用户。可以理解为指定哪个用户启动 Nginx。如果未指定，默认为 nobody，表示不限制
--group=<group>	在 nginx.conf 中未指定 user 指令的情况下，指定默认的 Nginx 服务器进程的属主用户组，即 Nginx 进程运行的用户组。可以理解为指定哪个用户组的用户启动 Nginx。如果未指定，默认为 nobody，表示不限制
--builddir=<dir>	指定编译时的目录
--with-debug	声明启用 Nginx 的调试日志
--add-module=<path>	指定第三方模块的路径，用以编译到 Nginx 服务器中
--with-poll_module	声明启用 poll 模块。poll 模块是信号处理的一种方法，和下面提到的 select 模式类似，都是采用轮询方法处理信号
--without-poll_module	声明禁止 poll 模块

续表

选项	说明
--with-select_module	声明启用 select 信号处理模式。若 configure 未找到指定其他的信号处理模式（上面提到的 SUN 系统中的 kqueue、Linux2.6+内核的 epoll、实时信号 rtsig 以及和 select 类似的/dev/poll 等），则默认使用 select 模式
--without-select_module	声明禁止 select 信号处理模式
--with-http_ssl_module	声明启用 HTTP 的 ssl 模块，这样 Nginx 服务器就可以支持 HTTPS 请求了。这个模块的正常运行需要安装 openssl 库（在 DEBIAN 上为 libssl）
--with-http_realip_module	声明启用 HTTP 的 realip 模块。默认不启用
--with-http_addition_module	声明启用 HTTP 的 addition 模块。默认不启用
--with-http_sub_module	声明启用 HTTP 的 sub 模块。默认不启用
--with-http_dav_module	声明启用 HTTP 的 dav 模块。默认不启用
--with-http_flv_module	声明启用 HTTP 的 flv 模块。默认不启用。flv 模块使得 Nginx 服务器支持 flv 媒体流的传输
--with-http_stub_status_module	声明启用 Server Status 页。默认不启用
--with-http_perl_module	声明启用 HTTP 的 perl 模块。默认不启用。perl 模块使得 Nginx 服务器支持 perl 脚本的运行
--with-perl_modules_path=<path>	指定 perl 模块的路径
--with-perl=<path>	指定 perl 执行文件的路径
--without-http_charset_module	声明禁用 HTTP 的 charset 模块。默认启用
--without-http_gzip_module	声明禁用 HTTP 的 gzip 模块。默认启用。使用 gzip 模块，需要安装 zlib 库
--without-http_ssi_module	声明禁用 HTTP 的 ssi 模块。默认启用
--without-http_userid_module	声明禁用 HTTP 的 userid 模块。默认启用
--without-http_access_module	声明禁用 HTTP 的 access 模块。默认启用
--without-http_auth_basic_module	声明禁用 HTTP 的 auth basic 模块。默认启用
--without-http_autoindex_module	声明禁用 HTTP 的 autoindex 模块。默认启用
--without-http_geo_module	声明禁用 HTTP 的 geo 模块。默认启用
--without-http_map_module	声明禁用 HTTP 的 map 模块。默认启用
--without-http_referer_module	声明禁用 HTTP 的 referer 模块。默认启用
--without-http_rewrite_module	声明禁用 HTTP 的 geo 模块。默认启用。使用 rewrite 模块，需要安装 pcre 库
--without-http_proxy_module	声明禁用 HTTP 的 proxy 模块。默认启用
--without-http_fastcgi_module	声明禁用 HTTP 的 fastcgi 模块。默认启用
--without-http_memcached_module	声明禁用 HTTP 的 memcached 模块。默认启用

续表

选项	说明
--without-http_limit_zone_module	声明禁用 HTTP 的 limit zone 模块。默认启用。Limit zone 模块主要负责 Nginx 服务器共享内存的管理
--without-http_empty_gif_module	声明禁用 HTTP 的 empty gif 模块。默认启用
--without-http_browser_module	声明禁用 HTTP 的 browser 模块。默认启用
--without-http_upstream_ip_hash_module	声明禁用 HTTP 的 upstream ip hash 模块。默认启用
--http-client-body-temp-path=<path>	指定存放 HTTP 访问客户端请求报文的临时文件的路径
--http-proxy-temp-path=<path>	启用 HTTP 的 proxy 模块后，指定存放 HTTP 代理临时文件的路径
--http-fastcgi-temp-path=<path>	启用 HTTP 的 fastcgi 模块后，指定存放 fastcgi 模块临时文件的路径
--without-http	声明禁用 HTTP Server
--with-mail	声明启用 IMAP4/POP3/SMTP 代理模块。该模块负责 Mail 代理服务的处理
--with-mail_ssl_module	声明启用 ngx_mail_ssl_module
--with-cc=<path>	指定 C 编译器的路径
--with-cpp=<path>	指定 C 预处理器的路径
--with-cc-opt=<options>	为 CFLGS 变量添加额外的参数，保证 Nginx 源代码及其模块能够正确编译。比如在 FreeBSD 系统中，为了在编译 Nginx 源代码的同时正确编译 pcre 库，必须使用--with-cc-opt="-I /usr/local/include"声明；再比如，为了使用 select 模块，必须增加系统允许打开的文件描述符的数量，其中一种方法就是使用--with-cc-opt="-D FD_SETSIZE=2048"声明
--with-ld-opt=<path>	为部分 Nginx 软件的模块编译指定链接库目录。比如在 FreeBSD 系统中，为了在编译 Nginx 源代码的同时编译 pcre 库，必须使用--with-ld-opt="-L /usr/local/lib"声明
--with-cpu-opt=<cpu>	为特定的 CPU 编译 Nginx 源代码。官方文档中指定的有效值包括：pentium、pentium pro、pentium 3、pentium 4、athlon、opteron、amd 64、sparc 32、sparc 64、ppc 64 等
--with-pcre=<dir>	指定 pcre 库源代码的路径。这样可以在编译 Nginx 源代码的同时编译 pcre 库，而不需要提前安装 pcre 库
--without-pcre	禁止 Nginx 服务器使用 pcre 库。此设置同时也会禁止 HTTP rewrite 模块，因为 rewrite 模块的正常运行必须由 pcre 库支持。在解析配置文件的 location 指令中的正则表达式时，也需要使用 pcre 库。因此，强烈建议用户不要使用此选项
--with-pcre-opt=<options>	为 pcre 库的 building 指定额外的指令
--with-md5=<dir>	指定 md5 库源代码的路径。这样可以在编译 Nginx 源代码的同时编译 md5 库，而不需要提前安装 md5 库
--with-md5-opt=<options>	为 md5 库的 building 指定额外的指令
--with-md5-asm	声明使用 md5 库的汇编源代码

续表

选项	说明
--with-sha1=<dir>	指定 sha1 库源代码的路径。这样可以在编译 Nginx 源代码的同时编译 sha1 库，而不需要提前安装 sha1 库
--with-sha1-opt==<options>	为 sha1 库的 building 指定额外的指令
--with-sha1-asm	声明使用 sha1 库的汇编源代码
--with-zlib=<dir>	指定 zlib 库源代码的路径。这样可以在编译 Nginx 源代码的同时编译 zlib 库，可以不提前安装 zlib 库。zlib 库支持 gzip 模块
--with-zlib-opt=<options>	为 zlib 库的 building 指定额外的指令
--with-zlib-asm=<cpu>	针对特殊的 CPU 声明使用汇编源代码。官方文档中指定的有效值包括：pentium、pentium pro 等
--with-openssl=<dir>	指定 OpenSSL 库源代码的路径。这样可以在编译 Nginx 源代码的同时编译 OpenSSL 库，而不需要提前安装 OpenSSL 库
--with-openssl-opt=<options>	为 OpenSSL 库的 building 指定额外的指令

了解了 configure 支持的常用选项后，就可以根据自己的实际情况使用 configure 脚本自动生成 Makefile 文件了。

将当前工作路径定位到/Nginx_123/Nginx_123_Compile/nginx-1.2.3/目录之后，笔者在自己的 Fedora 系统中，使用以下命令配置并生成 Makefile 文件：

```
#./configure --prefix=/Nginx                                    //编译配置
```

--prefix 指定了 Nginx 软件的安装路径为之前新建的 Nginx 目录。

其他的设置使用默认设置。

按 Enter 键运行命令，可以在屏幕上看到 configure 自动脚本运行的全过程。在运行过程中，configure 脚本调用 auto 目录中的各种脚本对系统环境以及相关的配置和设置进行了检查。

注意

生成的 Nginx 软件的 Makefile 文件就保存在当前的工作目录，即/Nginx_123/Nginx_123_Compile/nginx-1.2.3/中，可以使用 ls 指令查询。这里使用了最基本的 Nginx 配置。我们将在后文中逐渐增加对 Nginx 服务器的功能需求，指定的 configure 选项也将逐渐增加。

2.2.4 Linux 版本的编译和安装：Nginx 源代码的编译和安装

得到了 Nginx 软件的 Makefile 文件后，我们就可以编译源代码了。保持当前工作路径仍为/Nginx_123/Nginx_123_ Compile/nginx-1.2.3/目录，使用 make 命令进行编译：

```
#make                                                           //编译
```

同样，我们可以在屏幕上看到 Nginx 源代码的编译全过程。

编译顺利完成以后，使用 make 的 install 命令安装 Nginx 软件：

```
#make install                                                   //安装
```

命令运行完成后，将当前工作目录定位到/nginx 下，可以对 Nginx 服务器安装后的全部资源进行查看：

```
#cd /Nginx;
#ls -l
总用量 36
drwx------. 2 nobody root 4096 1月  7 14:52 client_body_temp
drwxr-xr-x. 2 root   root 4096 1月  9 17:11 conf
drwx------. 2 nobody root 4096 1月  7 14:52 fastcgi_temp
drwxr-xr-x. 2 root   root 4096 1月  7 14:48 html
drwxr-xr-x. 2 root   root 4096 1月 18 11:46 logs
drwx------. 12 nobody root 4096 1月  7 16:06 proxy_temp
drwxr-xr-x. 2 root   root 4096 1月  7 14:48 sbin
drwx------. 2 nobody root 4096 1月  7 14:52 scgi_temp
drwx------. 2 nobody root 4096 1月  7 14:52 uwsgi_temp
```

注意

如果在编译过程中出现了源代码的编译错误，屏幕上会出现错误信息，你可以对这些信息认真查阅，并结合表 2.1 的相关内容解决编译错误。根据笔者的经验，编译错误多是因缺少一些模块的支持库引起的，比如笔者曾经遇到过缺少 pcre 库等问题引起的编译错误。如果遇到此错误，则可以下载 pcre 的源代码包，并复制到某目录下，然后为 configure 指定--with-pcre=<dir>后生成 Makefile 文件。

另外，在没有改动源代码的情况下如果需要重新编译和安装 Nginx 软件，就不必再使用 configure 脚本自动生成 Makefile 了。可以先使用以下命令删除上次安装的 Nginx 软件：

```
#rm -rf /Nginx/*
```

然后定位到 nginx-1.2.3.tar.gz 解压目录，清除上次编译的遗留文件：

```
#cd /Nginx_123/Nginx_123_ Compile/nginx-1.2.3/; make clean
```

之后再使用以下命令进行编译和安装：

```
#make; make install
```

到此为止，我们就安装好了一个最基本的 Nginx 服务器，其安装路径为/Nginx 目录。我们来看一下 Nginx 服务器的安装目录中包含了哪些内容：

```
#ls *
conf:
fastcgi.conf    fastcgi_params    koi-utf  mime.types      nginx.conf
scgi_params   uwsgi_params    win-utf    fastcgi.conf.default
fastcgi_params.default
koi-win   mime.types.default   nginx.conf.default   scgi_params.default
uwsgi_params.default
html:
50x.html  index.html
logs:
access.log  error.log
sbin:
nginx
```

Nginx 服务器的安装目录中主要包括了 conf、html、logs 和 sbin 等 4 个目录。

其中，conf 目录中存放了 Nginx 的所有配置文件。其中，nginx.conf 文件是 Nginx 服务器的主配置文件，其他配置文件是用来配置 Nginx 的相关功能的，比如，配置 fastcgi 使用的 fastcgi.conf 和

fastcgi_params 两个文件，这些将在专门的章节中详细讲解。在此目录下，所有的配置文件都提供了以.default 结尾的默认配置文件，方便我们将配置过的.conf 文件恢复到初始状态。

html 目录中存放了 Nginx 服务器在运行过程中调用的一些 html 网页文件。我们来简单看一下其中的内容。

首先是 index.html 文件。

```
#cat /Nginx/html/index.html                                    //打印文件内容
<html>
<head>
<title>Welcome to nginx!</title>
</head>
<body bgcolor="white" text="black">
<center><h1>Welcome to nginx!</h1></center>
</body>
</html>
```

从内容来看，index.html 是在浏览器页面上打印出“Welcome to nginx!”这样一句话。可以推测一下，这个文件应该是 Nginx 服务器运行成功后，默认调用的网页，提示用户 Nginx 服务器运行成功了吧？我们到学习 Nginx 服务器启动的相关内容时再来验证推测是否正确。

接下来再看 50x.html 文件的内容：

```
#cat /Nginx/html/50x.html
    <html>
        <head>
            <title>The page is temporarily unavailable</title>
                <style>
            body { font-family: Tahoma, Verdana, Arial, sans-serif; }
                </style>
        </head>
        <body bgcolor="white" text="black">
            <table width="100%" height="100%">
                <tr>
                    <td align="center" valign="middle">
                The page you are looking for is temporarily unavailable.<br/>
                Please try again later.
                    </td>
                </tr>
            </table>
        </body>
</html>
```

50x.html 是在浏览器页面上打印出“The page you are looking for is temporarily unavailable. Please try again later.”这样一句话。很显然，Nginx 在出现某些问题时将会调用这个页面。

事实上，我们还可以在 html 目录下自定义一些网页文件，并在配置文件中配置发生什么情况时转到相应的文件。这些技巧我们在以后的章节中会逐步涉及。

logs 目录，顾名思义，是用来存放 Nginx 服务器的日志的。目前 Nginx 服务器没有启用，因此目

录是空的。Nginx 的日志功能比较强大，有不同的种类，并且可以自定义输出格式内容等，我们在相关的章节详细阐述。

最后是 sbin 目录，目前其中只有 nginx 一个文件，这就是 Nginx 服务器的主程序了。

2.3　Nginx 服务的启停控制

完成 Nginx 服务器的安装以后，我们来学习如何启动、重启和停止 Nginx 服务。在 Linux 平台下，控制 Nginx 服务的启停有不止一种方法。

2.3.1　Nginx 服务的信号控制

在 Nginx 服务的启停办法中，有一类是通过信号机制来实现的，因此，我们先来介绍一下 Nginx 服务器的信号控制。

Nginx 服务在运行时，会保持一个主进程和一个或多个 worker process 工作进程。我们通过给 Nginx 服务的主进程发送信号就可以控制服务的启停了。那么，如何给主进程发送信号呢？首先要知道主进程的进程号 PID。

获取 PID 有两个途径。一个是，在 Nginx 服务启动以后，默认在 Nginx 服务器安装目录下的 logs 目录中会产生文件名为 nginx.pid 的文件，此文件中保持的就是 Nginx 服务主进程的 PID。这个文件的存放路径和文件名都可以在 Nginx 服务器的配置文件中进行配置，我们在下一节中可以看到详细过程。

当前笔者的 Nginx 服务主进程 PID 为 4136：

```
# cat nginx.pid
4136
```

第二个获取 Nginx 服务主进程 PID 的办法是使用 Linux 平台下查看进程的工具 ps，使用方法是：

```
# ps -ef | grep nginx
root         4136    1   0 01:05 ?        00:00:00 nginx: master process ./nginx
nobody       4137  4136   0 01:05 ?        00:00:00 nginx: worker process
nobody       4138  4136   0 01:05 ?        00:00:00 nginx: worker process
nobody       4139  4136   0 01:05 ?        00:00:00 nginx: worker process
root         4360  4062   0 02:44 pts/0    00:00:00 grep Nginx
```

从运行命令的结果来看，系统中包含一个 Nginx 服务的主进程 master process 和三个工作进程 worker process，其中主进程对应的 PID 为第二列中的 4136，这和 nginx.pid 文件中的一致。

Nginx 服务主进程能够接收的信号如表 2.2 所列。

表 2.2　Nginx 服务可接收的信号

信号	作用
TERM 或 INT	快速停止 Nginx 服务
QUIT	平缓停止 Nginx 服务
HUP	使用新的配置文件启动进程，之后平缓停止原有进程，也就是所谓的“平滑重启”

续表

信号	作用
USR1	重新打开日志文件，常用于日志切割，在相关章节中会对此进一步说明
USR2	使用新版本的 Nginx 文件启动服务，之后平缓停止原有 Nginx 进程，也就是所谓的“平滑升级”
WINCH	平缓停止 worker process，用于 Nginx 服务器平滑升级

向 Nginx 服务主进程发送信号也有两种方法。一种是使用 nginx 二进制文件，在下一小节中说明；另一种方法是使用 kill 命令发送信号，其用法是：

```
kill SIGNAL PID
```

SIGNAL，用于指定信号，即指定表 2.2 中的某一个。

PID，指定 Nginx 服务主进程的 PID，也可以使用 nginx.pid 动态获取 PID 号：

```
kill SIGNAL `filepath`
```

其中，filepath 为 nginx.pid 的路径。

2.3.2 Nginx 服务的启动

在 Linux 平台下，启动 Nginx 服务器直接运行安装目录下 sbin 目录中的二进制文件即可。这里简要介绍一下二进制文件 nginx 的相关用法。将当前工作目录定位到 Nginx 安装目录下，运行以下命令：

```
# ./sbin/nginx -h
nginx version: nginx/1.2.4
Usage: nginx [-?hvVtq] [-s signal] [-c filename] [-p prefix] [-g directives]
Options:
  -?,-h         : this help      显示该帮助信息
  -v            : show version and exit  打印版本号并退出
  -V            : show version and configure options then exit 打印版本号和配置并退出
  -t            : test configuration and exit    测试配置正确性并退出
  -q            : suppress non-error messages during configuration testing   测试
配置时只显示错误
  -s signal     : send signal to a master process: stop, quit, reopen, reload向主
进程发送信号
  -p prefix     : set prefix path (default: /Nginx/bin/)     指定 Nginx 服务器路径前缀
  -c filename   : set configuration file (default: conf/nginx.conf)  指定 Nginx 配
置文件路径
  -g directives : set global directives out of configuration file    指定 Nginx 附
加配置文件路径
```

接下来对帮助信息中允许使用的参数逐条进行分析。

在 nginx 的帮助信息中可以看到，nginx 可以使用一些参数。“-h”或者“-？”用来打印二进制文件 nginx 的用法，也就是当前显示的内容；“-v”用来显示 Nginx 服务器的版本号；“-V”除了显示版本号，还显示 Nginx 服务器编译情况，笔者测试输出为：

```
# ./nginx -V
nginx version: nginx/1.2.3
```

```
built by gcc 4.1.2 20070115 (prerelease) (Fedora Linux)
configure arguments: --prefix=/Nginx
```

“-t”检查 Nginx 服务器配置文件是否有语法错误，可以与“-c”联用，使输出内容更详细，这对查找配置文件中的语法错误很有帮助，如果检查通过，将显示类似下面的信息：

```
# ./nginx -t
nginx: the configuration file /Nginx/conf/nginx.conf syntax is ok
nginx: configuration file /Nginx/conf/nginx.conf test is successful
```

“-q”与“-t”联用，如果配置文件无错误，将不输出上面的内容；“-s *signal*”用来向 Nginx 服务的主进程发送信号，关于信号控制，查看上一小节的内容；“-p *prefix*”用来改变 Nginx 的安装路径，常用在平滑升级 Nginx 服务器的场合；“-c *filename*”用来指定启动 Nginx 服务使用的配置文件；“-g *directives*”用来补充 Nginx 配置文件，向 Nginx 服务指定启动时应用于全局的配置。

介绍了以上二进制文件 nginx 的用法后，相信读者已经知道如何启动 Nginx 服务了。我们使用默认的配置文件，因此直接运行：

```
# ./sbin/Nginx
```

如果没有任何错误信息输出，Nginx 服务就启动了。可以使用 ps -ef | grep nginx 命令查看 Nginx 服务的进程状态。

2.3.3　Nginx 服务的停止

停止 Nginx 服务有两种方法：一种是快速停止；一种是平缓停止。快速停止是指立即停止当前 Nginx 服务正在处理的所有网络请求，马上丢弃连接，停止工作；平缓停止是指允许 Nginx 服务将当前正在处理的网络请求处理完成，但不再接收新的请求，之后关闭连接，停止工作。

停止 Nginx 服务的操作比较多。可以发送信号：

```
# ./sbin/Nginx -g TERM | INT | QUIT
```

其中，TERM 和 INT 信号用于快速停止，QUIT 用于平缓停止。

或者：

```
# kill TERM | INT | QUIT `/Nginx/logs/nginx.pid`
```

当然也可以使用 kill 命令向 Nginx 进程发送-9 或者 SIGKILL 信号强制关闭 Nginx 服务：

```
# kill -9 | SIGKILL `/Nginx/logs/nginx.pid`
```

但不建议这样使用。

2.3.4　Nginx 服务的重启

更改 Nginx 服务器的配置和加入新模块后，如果希望当前的 Nginx 服务应用新的配置或使新模块生效，就需要重启 Nginx 服务。当然我们可以先关闭 Nginx 服务，然后使用新的 Nginx 配置文件重启服务。这里主要介绍 Nginx 服务的平滑重启。

平滑重启是这样一个过程，Nginx 服务进程接收到信号后，首先读取新的 Nginx 配置文件，如果配置语法正确，则启动新的 Nginx 服务，然后平缓关闭旧的服务进程；如果新的 Nginx 配置有问题，将显示错误，仍然使用旧的 Nginx 进程提供服务。

使用以下命令实现 Nginx 服务的平滑重启：

```
# ./sbin/nginx -g HUP [-c newConfFile]
```

HUP 信号用于发送平滑重启信号。

newConfFile，可选项，用于指定新配置文件的路径。

或者，使用新的配置文件代替了旧的配置文件后，使用：

```
# kill HUP `/Nginx/logs/nginx.pid`
```

也可以实现平滑重启。

2.3.5　Nginx 服务器的升级

如果要对当前的 Nginx 服务器进行版本升级，应用新模块，最简单的办法是停止当前 Nginx 服务，然后开启新的 Nginx 服务，但这样就会导致在一段时间内，用户无法访问服务器。为了解决这个问题，Nginx 服务器提供平滑升级的功能。

平滑升级的过程是这样的，Nginx 服务接收到 USR2 信号后，首先将旧的 nginx.pid 文件（如果在配置文件中更改过这个文件的名字，也是相同的过程）添加后缀.oldbin，变为 nginx.pid.oldbin 文件；然后执行新版本 Nginx 服务器的二进制文件启动服务。如果新的服务启动成功，系统中将有新旧两个 Nginx 服务共同提供 Web 服务。之后，需要向旧的 Nginx 服务进程发送 WINCH 信号，使旧的 Nginx 服务平滑停止，并删除 nginx.pid.oldbin 文件。在发送 WINCH 信号之前，可以随时停止新的 Nginx 服务。

注意

为了实现 Nginx 服务器的平滑升级，新的服务器安装路径应该和旧的保持一致。因此建议用户在安装新服务器之前先备份旧服务器。如果由于某种原因无法保持新旧服务器安装路径一致，则可以先使用以下命令将旧服务器的安装路径更改为新服务器的安装路径：

```
# ./Nginx/nginx -p newInstallPath
```

其中，newInstallPath 为新服务器的安装路径。之后，备份旧服务器，安装新服务器即可。

做好准备工作以后，使用以下命令实现 Nginx 服务的平滑升级：

```
# ./sbin/Nginx -g USR2
```

其中，USR2 信号用于发送平滑升级信号。或者，使用：

```
# kill USR2 `/Nginx/logs/nginx.pid`
```

通过 ps -ef | grep nginx 查看新的 Nginx 服务启动正常，再使用：

```
# ./sbin/Nginx -g WINCH
```

其中，WINCH 信号用于发送平滑停止旧服务信号。或者，使用：

```
# kill WINCH `/Nginx/logs/nginx.pid`
```

这样就在不停止提供 Web 服务的前提下完成了 Nginx 服务器的平滑升级。

2.4　Nginx 服务器基础配置指令

从上面的内容我们知道，默认的 Nginx 服务器配置文件都存放在安装目录 conf 中，主配置文件名为 nginx.conf。这一节，我们学习 nginx.conf 的内容和基本配置方法。

以下代码清单是默认的 nginx.conf 文件中的完整内容。

注意

在 nginx.conf 原始文件中，还包括注释内容，注释标志为“#”，由于篇幅所限，代码清单中将所有的注释去除。笔者添加的注释说明了各条语句的生效范围，让大家对指令的作用域有一个初步的了解，后面章节我们会进一步讲解。

```
worker_processes  1;                                    #全局生效

events {
    worker_connections  1024;                           #在 events 部分中生效
}

http {
    include       mime.types;                           #以下指令在 http 部分中生效
    default_type  application/octet-stream;
    sendfile        on;
    keepalive_timeout  65;
     server {                                           #以下指令在 http 的 server 部分中生效
        listen       80;
        server_name  localhost;
        location / {                                    #以下指令在 http/server 的 location 中生效
            root   html;
            index  index.html index.htm;
        }
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   html;
        }
    }
}
```

初始的 Nginx 服务器主配置文件比较长，不过结构和内容还是比较清晰的，以下我们分小节对上面的内容进行详细介绍。

2.4.1　nginx.conf 文件的结构

从上面的配置文件内容，我们可以归纳出 nginx.conf 文件的基本结构为：（“#”后边的内容是笔者添加的注释内容，它们的含义在后文中会给出）：

```
 ...                                                    #全局块
 events                                                 #events 块
{
    ...
 }
 http                                                   #http 块
 {
        ...                                             #http 全局块
```

```
    server                                                    #server 块
    {
        ...                                                   #server 全局块
        location [PATTERN]                                    #location 块
        {
            ...
        }
        location [PATTERN]                                    #location 块
        {
            ...
        }
    }
    server                                                    #server 块
    {
        ...
    }
    ...                                                       #http 全局块
}
```

最外层的花括号将内容整体分为两部分，再加上最开始的内容，即第一行省略号表示的，nginx.conf 一共由三部分组成，分别为全局块、events 块和 http 块。在 http 块中，又包含 http 全局块、多个 server 块。每个 server 块中，可以包含 server 全局块和多个 location 块。在同一配置块中嵌套的配置块，各个之间不存在次序关系。

配置文件支持大量可配置的指令，绝大多数指令不是特定属于某一个块的。同一个指令放在不同层级的块中，其作用域也不同，一般情况下，高一级块中的指令可以作用于自身所在的块和此块包含的所有低层级块。如果某个指令在两个不同层级的块中同时出现，则采用“就近原则”，即以较低层级块中的配置为准。比如，某指令同时出现在 http 全局块中和 server 块中，并且配置不同，则应该以 server 块中的配置为准。

在介绍可配置指令之前，我们先来看看各个块的作用。

1. 全局块

全局块是默认配置文件从开始到 events 块之间的一部分内容，主要设置一些影响 Nginx 服务器整体运行的配置指令，因此，这些指令的作用域是 Nginx 服务器全局。

通常包括配置运行 Nginx 服务器的用户（组）、允许生成的 worker process 数、Nginx 进程 PID 存放路径、日志的存放路径和类型以及配置文件引入等。

2. events 块

events 块涉及的指令主要影响 Nginx 服务器与用户的网络连接。常用到的设置包括是否开启对多 worker process 下的网络连接进行序列化，是否允许同时接收多个网络连接，选取哪种事件驱动模型处理连接请求，每个 worker process 可以同时支持的最大连接数等。

这一部分的指令对 Nginx 服务器的性能影响较大，在实际配置中应该根据实际情况灵活调整。相关的详细分析我们在本书后面的相关章节会陆续学习。

3. http 块

http 块是 Nginx 服务器配置中的重要部分，代理、缓存和日志定义等绝大多数的功能和第三方模块的配置都可以放在这个模块中。

前面已经提到，http 块中可以包含自己的全局块，也可以包含 server 块，server 块中又可以进一步包含 location 块，在本书中我们使用“http 全局块”来表示 http 中自己的全局块，即 http 块中不包含在 server 块中的部分。

可以在 http 全局块中配置的指令包括文件引入、MIME-Type 定义、日志自定义、是否使用 sendfile 传输文件、连接超时时间、单连接请求数上限等。

4. server 块

server 块和“虚拟主机”的概念有密切联系。为了加深对相关配置的理解，在介绍 server 块之前，我们简单了解一下虚拟主机的相关内容。

虚拟主机，又称虚拟服务器、主机空间或是网页空间，它是一种技术。该技术是为了节省互联网服务器硬件成本而出现的。这里的“主机”或“空间”是由实体的服务器延伸而来，硬件系统可以基于服务器群，或者单个服务器等。虚拟主机技术主要应用于 HTTP、FTP 及 EMAIL 等多项服务，将一台服务器的某项或者全部服务内容逻辑划分为多个服务单位，对外表现为多个服务器，从而充分利用服务器硬件资源。从用户角度来看，一台虚拟主机和一台独立的硬件主机是完全一样的。

在使用 Nginx 服务器提供 Web 服务时，利用虚拟主机的技术就可以避免为每一个要运行的网站提供单独的 Nginx 服务器，也无需为每个网站对应运行一组 Nginx 进程。虚拟主机技术使得 Nginx 服务器可以在同一台服务器上只运行一组 Nginx 进程，就可以运行多个网站。那么，如何对 Nginx 进行配置才能达到这种效果呢？本节介绍的 server 块就是用来完成这个功能的。

在前面提到过，每一个 http 块都可以包含多个 server 块，而每个 server 块就相当于一台虚拟主机，它内部可有多台主机联合提供服务，一起对外提供在逻辑上关系密切的一组服务（或网站）。我们先来学习 server 全局块中常见的指令及其配置。server 全局块指令的作用域为本 server 块，其不会影响到其他的 server 块。

注意

在 http 全局块中介绍过的部分指令可以在 server 块中和 location 块中使用，其作用域问题也已在前文中说明，后面就不再赘述。

和 http 块相同，server 块也可以包含自己的全局块，同时可以包含多个 location 块。在 server 全局块中，最常见的两个配置项是本虚拟主机的监听配置和本虚拟主机的名称或 IP 配置。

5. location 块

每个 server 块中可以包含多个 location 块。从严格意义上说，location 其实是 server 块的一个指令，只是由于其在整个 Nginx 配置文档中起着重要的作用，而且 Nginx 服务器在许多功能上的灵活性往往在 location 指令的配置中体现出来，因此笔者认为应该将其单独列为一个“块”，一方面引起读者的重视，另一方面通过专门的详细介绍突出其重要性，加深读者的理解。

这些 location 块的主要作用是，基于 Nginx 服务器接收到的请求字符串（例如，*server_name/uri-string*），对除虚拟主机名称（也可以是 IP 别名，后文有详细阐述）之外的字符串（前

例中“*/uri-string*”部分）进行匹配，对特定的请求进行处理。地址定向、数据缓存和应答控制等功能都是在这部分实现。许多第三方模块的配置也是在 location 块中提供功能。

2.4.2　配置运行 Nginx 服务器用户（组）

用于配置运行 Nginx 服务器用户（组）的指令是 user，其语法格式为：

```
user user [group];
```

- *user*，指定可以运行 Nginx 服务器的用户。
- *group*，可选项，指定可以运行 Nginx 服务器的用户组。

只有被设置的用户或者用户组成员才有权限启动 Nginx 进程，如果是其他用户（test_user）尝试启用 Nginx 进程，将会报错：

```
nginx: [emerg] getpwnam("test_user") failed (2: No such file or directory) in
/Nginx/conf/nginx.conf:2
```

可以从错误信息中看到，Nginx 无法运行的原因是查找 test_user 失败，引起错误的原因是 nginx.conf 的第二行内容，即配置 Nginx 服务器用户（组）的内容。

如果希望所有用户都可以启动 Nginx 进程，有两种办法：一是将此指令行注释掉：

```
#user [user] [group];
```

或者将用户（和用户组）设置为 nobody：

```
user nobody nobody;
```

这也是 user 指令的默认配置。user 指令只能在全局块中配置。

注意

在 Nginx 配置文件中，每一条指令配置都必须以分号结束，请不要忘记。

2.4.3　配置允许生成的 worker process 数

worker process 是 Nginx 服务器实现并发处理服务的关键所在。从理论上来说，worker process 的值越大，可以支持的并发处理量也越多，但实际上它还要受到来自软件本身、操作系统本身资源和能力、硬件设备（如 CPU 和磁盘驱动器）等的制约。后面会用专门的章节来讨论 Nginx 服务器的高级配置。

配置允许生成的 worker process 数的指令是 worker_processes，其语法格式为：

```
worker_processes number | auto;
```

- *number*，指定 Nginx 进程最多可以产生的 worker process 数。
- auto，设置此值，Nginx 进程将自动检测。

在默认的配置文件中，*number*=1。启动 Nginx 服务器以后，使用以下命令可以看到 Nginx 服务器除了主进程 master process ../sbin/nginx 之外，还生成了一个 worker process：

```
#ps ax | grep nginx
 8579 ?        Ss     0:00 nginx: master process ../sbin/nginx
 8580 ?        S      0:00 nginx: worker process
```

如果将 number 改为 3，重新运行 Nginx 进程，再次使用以上命令，则可以看到此时的 Nginx 服务

器除了主进程 master process ../sbin/nginx 之外，已经生成了三个 worker process：

```
#ps ax | grep nginx
 8626 ?        Ss     0:00 nginx: master process ../sbin/nginx
 8627 ?        S      0:00 nginx: worker process
 8628 ?        S      0:00 nginx: worker process
 8629 ?        S      0:00 nginx: worker process
```

此指令只能在全局块中设置。

2.4.4　配置 Nginx 进程 PID 存放路径

Nginx 进程作为系统的守护进程运行，我们需要在某文件中保存当前运行程序的主进程号。Nginx 支持对它的存放路径进行自定义配置，指令是 pid，其语法格式为：

```
pid file;
```

其中，*file* 指定存放路径和文件名称。

配置文件默认将此文件存放在 Nginx 安装目录 logs 下，名字为 nginx.pid。path 可以是绝对路径，也可以是以 Nginx 安装目录为根目录的相对路径。比如要把 nginx.pid 放置到 Nginx 安装目录 sbin 下，文件名为 web_nginx，则可以使用以下配置：

```
pid sbin/web_nginx
```

注意

在指定[path]的时候，一定要包括文件名，如果只设置了路径，没有设置文件名，则会报以下错误：

```
nginx: [emerg] open() "/Nginx/logs/" failed (21: Is a directory)
```

此指令只能在全局块中进行配置。

2.4.5　配置错误日志的存放路径

在全局块、http 块和 server 块中都可以对 Nginx 服务器的日志进行相关配置。这里首先介绍全局块下日志的存放配置，后两种情况的配置基本相同，只是作用域不同。使用的指令是 error_log，其语法结构是：

```
error_log file | stderr [debug | info | notice | warn | error | crit | alert | emerg];
```

从语法结构可以看到，Nginx 服务器的日志支持输出到某一固定的文件 file 或者输出到标准错误输出 stderr；日志的级别是可选项，由低到高分为 debug（需要在编译时使用--with-debug 开启 debug 开关，参见 2.2.2 节）、info、notice、warn、error、crit、alert、emerg 等。需要注意的是，设置某一级别后，比这一级别高的日志也会被记录。比如设置 warn 级别后，级别为 warn 以及 error、crit、alert 和 emerg 的日志都会被记录下来。

下面我们先看一个配置的实例，这也是 Nginx 默认的日志存放和级别设置：

```
error_log logs/error.log error;
```

注意

指定的文件对于运行 Nginx 进程的用户具有写权限，否则在启动 Nginx 进程的时候会出现以下报错信息：

```
nginx: [alert]: could not open error log file: open() "/Nginx/logs/error.log" failed
(13: Permission denied)
```

此指令可以在全局块、http 块、server 块以及 location 块中配置。

2.4.6 配置文件的引入

在一些情况下，我们可能需要将其他的 Nginx 配置或者第三方模块的配置引用到当前的主配置文件中。Nginx 提供了 include 指令来完成配置文件的引入，其语法结构为：

```
include file;
```

其中，*file* 是要引入的配置文件，它支持相对路径。

注意

新引用进来的文件同样要求运行 Nginx 进程的用户对其具有写权限，并且符合 Nginx 配置文件规定的相关语法和结构。

此指令可以放在配置文件的任意地方。

2.4.7 设置网络连接的序列化

在《UNIX 网络编程》第 1 卷里提到过一个叫“惊群”的问题（Thundering herd problem），大致意思是，当某一时刻只有一个网络连接到来时，多个睡眠进程会被同时叫醒，但只有一个进程可获得连接。如果每次唤醒的进程数目太多，会影响一部分系统性能。在 Nginx 服务器的多进程下，就有可能出现这样的问题。

为了解决这样的问题，Nginx 配置中包含了这样一条指令 accept_mutex，当其设置为开启的时候，将会对多个 Nginx 进程接收连接进行序列化，防止多个进程对连接的争抢。其语法结构为：

```
accept_mutex on | off;
```

此指令默认为开启（on）状态，其只能在 events 块中进行配置。

2.4.8 设置是否允许同时接收多个网络连接

每个 Nginx 服务器的 worker process 都有能力同时接收多个新到达的网络连接，但是这需要在配置文件中进行设置，其指令为 multi_accept，语法结构为：

```
multi_accept on | off;
```

此指令默认为关闭（off）状态，即每个 worker process 一次只能接收一个新到达的网络连接。此指令只能在 events 块中进行配置。

2.4.9 事件驱动模型的选择

Nginx 服务器提供了多种事件驱动模型来处理网络消息。配置文件中为我们提供了相关的指令来强制 Nginx 服务器选择哪种事件驱动模型进行消息处理，指令为 use，语法结构为：

```
use method;
```

其中，method 可选择的内容有：select、poll、kqueue、epoll、rtsig、/dev/poll 以及 eventport，其中几种模型是比较常用的，我们在后面用专门的章节讨论 Nginx 服务器的事件驱动模型。

注意

可以在编译时使用--with-select_module 和--without-select_module 设置是否强制编译 select 模块到 Nginx 内核；使用--with-poll_module 和--without-poll_module 设置是否强制编译 poll 模块到 Nginx 内核。

此指令只能在 events 块中进行配置。

2.4.10　配置最大连接数

指令 worker_connections 主要用来设置允许每一个 worker process 同时开启的最大连接数。其语法结构为：

```
worker_connections number;
```

此指令的默认设置为 512。

注意

这里的 number 不仅仅包括和前端用户建立的连接数，而是包括所有可能的连接数。另外，number 值不能大于操作系统支持打开的最大文件句柄数量。该指令在后边讨论 Nginx 服务器高级配置时还会再次提到。

此指令只能在 events 块中进行配置。

2.4.11　定义 MIME-Type

我们知道，在常用的浏览器中，可以显示的内容有 HTML、XML、GIF 及 Flash 等种类繁多的文本、媒体等资源，浏览器为区分这些资源，需要使用 MIME Type。换言之，MIME Type 是网络资源的媒体类型。Nginx 服务器作为 Web 服务器，必须能够识别前端请求的资源类型。

在默认的 Nginx 配置文件中，我们看到在 http 全局块中有以下两行配置：

```
include mime.types;
default_type application/octet-stream;
```

第一行从外部引用了 mime_types 文件，我们来看一下它的内容片段：

```
# cat mime.types
types {
    text/html                             html htm shtml;
    ...
    image/gif                             gif;
    ...
    application/x-javascript              js;
    ...
    audio/midi                            mid midi kar;
    ...
    video/3gpp                            3gpp 3gp;
    ...
}
```

从 mime_types 文件的内容片段可以看到，其中定义了一个 types 结构，结构中包含了浏览器能够识别的 MIME 类型以及对应于相关类型的文件后缀名。由于 mime_types 文件是主配置文件应用的第三方文件，因此，types 也是 Nginx 配置文件中的一个配置块，我们可以称之为 types 块，其用于定义

MIME 类型。MIME 的知识不是本文的重点，不在这里详述，感兴趣的读者可以自行阅读相关书籍。

第二行中使用指令 default_type 配置了用于处理前端请求的 MIME 类型，其语法结构为：

```
default_type mime-type;
```

其中，*mime-type* 为 types 块中定义的 MIME-type，如果不加此指令，默认值为 text/plain。此指令还可以在 http 块、server 块或者 location 块中进行配置。

2.4.12　自定义服务日志

在全局块中，我们介绍过 errer_log 指令，其用于配置 Nginx 进程运行时的日志存放和级别，此处所指的日志与常规的不同，它是指记录 Nginx 服务器提供服务过程应答前端请求的日志，我们将其称为服务日志以示区分。

Nginx 服务器支持对服务日志的格式、大小、输出等进行配置，需要使用两个指令，分别是 access_log 和 log_format 指令。

access_log 指令的语法结构为：

```
access_log path [format [buffer=size]];
```

- *path*，配置服务日志的文件存放的路径和名称。
- *format*，可选项，自定义服务日志的格式字符串，也可以通过“格式串的名称”使用 log_format 指令定义好的格式。“格式串的名称”在 log_format 指令中定义。
- *size*，配置临时存放日志的内存缓存区大小。

此指令可以在 http 块、server 块或者 location 块中进行设置。默认的配置为：

```
access_log logs/access.log combined;
```

其中，combined 为 log_format 指令默认定义的日志格式字符串的名称。

如果要取消记录服务日志的功能，则使用：

```
access_log off;
```

和 access_log 联合使用的另一个指令是 log_format，它专门用于定义服务日志的格式，并且可以为格式字符串定义一个名字，以便 access_log 指令可以直接调用。其语法格式为：

```
log_format name string ...;
```

- *name*，格式字符串的名字，默认的名字为 combined。
- *string*，服务日志的格式字符串。在定义过程中，可以使用 Nginx 配置预设的一些变量获取相关内容，变量的名称使用双引号括起来，*string* 整体使用单引号括起来。在 string 中可以使用的变量请参见本书“附录 A”的相关内容。

我们来看一个示例以加深理解：

```
log_format  exampleLog  '$remote_addr - [$time_local] $request '
                        '$status $body_bytes_sent $http_referer '
                        '$http_user_agent';
```

这条配置定义了服务日志文件的名称为 exampleLog。笔者对其测试的结果，输出了如下日志片段：

```
192.168.1.102 - [31/Oct/2011:20:41:39 +0800] "GET / HTTP/1.1" 200 151 "-" "Mozilla/5.0
(compatible; MSIE 10.0; Windows NT 6.2; Trident/6.0)"
```

```
192.168.1.102 - [31/Oct/2011:20:41:39 +0800] "GET /favicon.ico HTTP/1.1" 404 570 "-"
"Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; Trident/6.0)"
```

简单分析一下第二条日志，$remote_addr 获取到用户机的 IP 地址为 192.168.1.102，$time_local 获取到本地时间为 31/Oct/2011:20:41:39 +0800，$request 获取到请求为 GET /favicon.ico HTTP/1.1，$status 获取到请求状态为 404（未找到，这是笔者将请求的网页暂时移除造成的），$body_bytes_sent 获取到请求体的大小为 570B，$http_referer 未获取到任何内容，$http_user_agent 获取到用户使用 Mozilla 浏览器。

通过分析可以看到，在正常情况下，对于绝大多数的内置变量，Nginx 服务器都能够获取到相关内容，但也会出现空值的情况。

此指令只能在 http 块中进行配置。

2.4.13　配置允许 sendfile 方式传输文件

在 Apache、lighttd 等 Web 服务器配置中，都有和 sendfile 相关的配置，这里主要学习一下配置 sendfile 传输方式的相关指令 sendfile 和 sendfile_max_chunk 以及它们的语法结构：

```
sendfile on | off;
```

用于开启或者关闭使用 sendfile()传输文件，默认值为 off，可以在 http 块、server 块或者 location 块中进行配置。

```
sendfile_max_chunk size;
```

其中，*size* 值如果大于 0，Nginx 进程的每个 worker process 每次调用 sendfile()传输的数据量最大不能超过这个值；如果设置为 0，则无限制。默认值为 0。此指令可以在 http 块、server 块或 location 块中配置。

下面是第二个指令的配置示例：

```
sendfile_max_chunk 128k;
```

2.4.14　配置连接超时时间

与用户建立会话连接后，Nginx 服务器可以保持这些连接打开一段时间，指令 keepalive_timeout 就是用来设置此时间的，其语法结构是：

```
keepalive_timeout timeout [header_timeout];
```

- *timeout*，服务器端对连接的保持时间。默认值为 75s。
- *header_timeout*，可选项，在应答报文头部的 Keep-Alive 域设置超时时间："Keep-Alive: *timeout= header_timeout*"。报文中的这个指令可以被 Mozilla 或者 Konqueror 识别。

此指令还可以出现在 server 块和 location 块中，如下是一个配置示例：

```
keepalive_timeout 120s 100s;
```

其含义是，在服务器端保持连接的时间设置为 120 s，发给用户端的应答报文头部中 Keep-Alive 域的超时时间设置为 100 s。

此指令可以在 http 块、server 块或 location 块中配置。

2.4.15 单连接请求数上限

Nginx 服务器端和用户端建立会话连接后，用户端通过此连接发送请求。指令 keepalive_requests 用于限制用户通过某一连接向 Nginx 服务器发送请求的次数。其语法结构为：

```
keepalive_requests number;
```

此指令还可以出现在 server 块和 location 块中，默认设置为 100。

2.4.16 配置网络监听

配置监听使用指令 listen，其配置方法主要有三种，我们先分别介绍三种配置的语法结构，然后统一介绍涉及的相关变量和标识符。第一种配置监听的 IP 地址，语法结构为：

```
listen address[:port] [default_server] [setfib=number] [backlog=number] [rcvbuf=size]
[sndbuf=size] [deferred]
      [accept_filter=filter] [bind] [ssl];
```

第二种配置监听端口，其语法结构是：

```
listen port [default_server] [setfib=number] [backlog=number] [rcvbuf=size] [sndbuf=
size] [accept_filter=filter]
      [deferred] [bind] [ipv6only=on|off] [ssl];
```

第三种配置 UNIX Domain Socket（一种在原有 Socket 框架上发展起来的 IPC 机制，用于在单个主机上执行客户/服务器通信，这不是本书的重点，请读者自行参阅相关书籍），其语法结构为：

```
listen unix:path [default_server] [backlog=number] [rcvbuf=size] [sndbuf=size]
[accept_filter=filter] [deferred]
      [bind] [ssl];
```

- *address*，IP 地址，如果是 IPv6 的地址，需要使用中括号“[]”括起来，比如[fe80::1]等。
- *port*，端口号，如果只定义了 IP 地址没有定义端口号，就使用 80 端口。
- *path*，socket 文件路径，如/var/run/nginx.sock 等。
- default_server，标识符，将此虚拟主机设置为 *address:port* 的默认主机。

注意

在 Nginx-0.8.21 之前的版本，使用的是 default。

- setfib=*number*，Nginx-0.8.44 中使用这个变量为监听 socket 关联路由表，目前只对 FreeBSD 起作用，不常用。
- backlog=*number*，设置监听函数 listen()最多允许多少网络连接同时处于挂起状态，在 FreeBSD 中默认为-1，其他平台默认为 511。
- rcvbuf=*size*，设置监听 socket 接收缓存区大小。
- sndbuf=*size*，设置监听 socket 发送缓存区大小。
- deferred，标识符，将 accept()设置为 Deferred 模式。
- accept_filter=*filter*，设置监听端口对请求的过滤，被过滤的内容不能被接收和处理。本指令只在 FreeBSD 和 NetBSD 5.0+平台下有效。*filter* 可以设置为 dataready 或 httpready，有关这两个值的细节已经超出本书的范围，感兴趣的读者可以参阅 Nginx 的官方文档。

- bind，标识符，使用独立的 bind()处理此 *address:port*；一般情况下，对于端口相同而 IP 地址不同的多个连接，Nginx 服务器将只使用一个监听命令，并使用 bind()处理端口相同的所有连接。
- ssl，标识符，设置会话连接使用 SSL 模式进行，此标识符和 Nginx 服务器提供的 HTTPS 服务有关。有关 HTTPS 服务的内容，我们将在后文介绍。

listen 指令的使用看起来比较复杂，但其实在一般的使用过程中，相对来说比较简单，默认的设置为：

```
listen  *:80 | *:8000;
```

即监听所有 80 端口和 8000 端口。下面给出一些示例来说明 listen 的用法：

```
listen 192.168.1.10:8000;  #监听具体的 IP 和具体的端口上的连接
listen 192.168.1.10;  #监听具体 IP 的所有端口上的连接
listen 8000;  #监听具体端口上的所有 IP 连接，等同于 listen *:8000;
listen 192.168.1.10 default_server backlog=1024;
                                    #设置 192.168.1.10 的连接请求默认由此虚拟主机处理，
                                    #并且允许最多 1024 网络连接同时处于挂起状态。
```

2.4.17　基于名称的虚拟主机配置

这里的“主机”，就是指此 server 块对外提供的虚拟主机。设置了主机的名称并配置好 DNS，用户就可以使用这个名称向此虚拟主机发送请求了。配置主机名称的指令为 server_name，其语法结构为：

```
server_name name ...;
```

对于 *name* 来说，可以只有一个名称，也可以由多个名称并列，之间用空格隔开。每个名字就是一个域名，由两段或者三段组成，之间由点号“.”隔开。下面是一个简单的示例：

```
server_name myserver.com www.myserver.com;
```

在该例中，此虚拟主机的名称设置为 myserver.com 或 www. myserver.com。Nginx 服务器规定，第一个名称作为此虚拟主机的主要名称。

在 *name* 中可以使用通配符“*”，但通配符只能用在由三段字符串组成的名称的首段或尾段，或者由两段字符串组成的名称的尾段，如：

```
server_name *.myserver.com www.myserver.*;
```

在 *name* 中还可以使用正则表达式，并使用波浪号“~”作为正则表达式字符串的开始标记，如：

```
server_name ~^www\d+\.myserver\.com$;
```

在该例中，此虚拟主机的名称设置使用了正则表达式（使用“~”标记），正则表达式的含义是：以 www 开头（使用“^”标记），紧跟一个或多个 0～9 的数字（“\d+”的含义，其中，“\d”代表 0～9 的某一个数字，“+”代表之前的一个字符出现一次或者多次），再紧跟.myserver.co（由于“.”在正则表达式中有特殊含义，因此需要使用“\”进行转义），最后以 m 结束（由“$”标记）。

通过以上的解释，大家应该理解了此虚拟主机可以接受使用哪些域名的请求了。比如对通过 www1.myserver.com 访问 Nginx 服务器的请求就可以使用此主机处理，而通过 www.myserver.com 的就不可以。因为“www”字符串之后必须有一个或者多个 0～9 的数字才能被正则表达式成功匹配。

关于正则表达式的相关内容，笔者在本书的“附录 B”中进行了总结，方便读者查询。

注意

从 Nginx-0.7.40 开始，name 中的正则表达式支持字符串捕获功能，即可以将正则表达式匹配成功的名称中的一部分字符串拾取出来，放在固定的变量中供下文使用。拾取的标识为一对完整的小括号“( )”且后面不紧跟其他的正则表达式字符，括号中的内容就是被拾取的内容。一个正则表达式中可以存在多对不嵌套的小括号，这些内容会从左到右依次存放在变量$1、$2、$3……中。下文使用时，直接使用这些变量即可。这些变量的有效区域不超出本 server 块。

为了理解正则表达式的捕获功能，大家看一个示例。虚拟主机的名称设置如下：

```
server_name ~^www\.(.+)\.com$;
```

当请求通过 www.myserver.com 到达 Nginx 服务器端时，其将会被上面的正则表达式配置成功，其中的“myserver”将会被捕获，并记录到$1 中。在本 server 块的下文配置中，当需要“myserver”时，就可以使用$1 代替“myserver”了。

由于 server_name 指令支持使用通配符和正则表达式两种配置名称的方式，因此在包含有多个虚拟主机的配置文件中，可能会出现一个名称被多个虚拟主机的 server_name 匹配成功。那么，来自这个名称的请求到底要交给哪个虚拟主机处理呢？Nginx 服务器做出如下规定：

a. 对于匹配方式不同的，按照以下的优先级选择虚拟主机，排在前面的优先处理请求。

① 准确匹配 server_name

② 通配符在开始时匹配 server_name 成功

③ 通配符在结尾时匹配 server_name 成功

④ 正则表达式匹配 server_name 成功

b. 在以上四种匹配方式中，如果 server_name 被处于同一优先级的匹配方式多次匹配成功，则首次匹配成功的虚拟主机处理请求。

2.4.18 基于 IP 的虚拟主机配置

Linux 操作系统支持 IP 别名的添加。配置基于 IP 的虚拟主机，即为 Nginx 服务器提供的每台虚拟主机配置一个不同的 IP，因此需要将网卡设置为同时能够监听多个 IP 地址。在 Linux 平台中可以使用 ifconfig 工具为同一块网卡添加多个 IP 别名。

笔者的 Linux 平台上当前的网络配置为：

```
#ifconfig                                                    #打印网络配置信息
eth1      Link encap:Ethernet  HWaddr 00:0C:29:AA:6A:19
          inet addr:192.168.1.3  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:feaa:6a19/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:30367 errors:0 dropped:0 overruns:0 frame:0
          TX packets:15372 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3665744 (3.4 Mb)  TX bytes:2439099 (2.3 Mb)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
```

```
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:133 errors:0 dropped:0 overruns:0 frame:0
          TX packets:133 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:9512 (9.2 Kb)  TX bytes:9512 (9.2 Kb)
```

eth1 为使用中的网卡，其 IP 值为 192.168.1.3。现在笔者需要为 eth1 添加两个 IP 别名 192.168.1.31 和 192.168.1.32，分别用于 Nginx 服务器提供的两个虚拟主机，需要执行以下操作：

```
#ifconfig eth1:0 192.168.1.31 netmask 255.255.255.0 up
#ifconfig eth1:1 192.168.1.32 netmask 255.255.255.0 up
```

命令中的 up 表示立即启用此别名。

此时笔者的 Linux 平台的网络配置为：

```
#ifconfig
eth1      Link encap:Ethernet  HWaddr 00:0C:29:AA:6A:19
          inet addr:192.168.1.3  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:feaa:6a19/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:31581 errors:0 dropped:0 overruns:0 frame:0
          TX packets:15558 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3749064 (3.5 Mb)  TX bytes:2461311 (2.3 Mb)
eth1:0    Link encap:Ethernet  HWaddr 00:0C:29:AA:6A:19
          inet addr:192.168.1.31  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
eth1:1    Link encap:Ethernet  HWaddr 00:0C:29:AA:6A:19
          inet addr:192.168.1.32  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:133 errors:0 dropped:0 overruns:0 frame:0
          TX packets:133 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:9512 (9.2 Kb)  TX bytes:9512 (9.2 Kb)
```

可以看到，eth1 增加了两个别名，分别为 eth1:0 和 eth1:1，IP 也分别是我们希望得到的结果。

注意

按照如上方法为 eth1 设置的别名在系统重启后将不予保存，需要重新设置。为了做到一劳永逸，我们可以将以上两条命令添加到 Linux 系统的启动脚本 rc.local 中。在笔者的系统中运行：

```
# echo "ifconfig eth1:0 192.168.1.31 netmask 255.255.255.0 up" >> /etc/rc.local
# echo "ifconfig eth1:1 192.168.1.32 netmask 255.255.255.0 up" >> /etc/rc.local
```

这样，在系统重启后，eth1 的别名就自动设置好了。

为网卡设置好别名以后，就可以为 Nginx 服务器配置基于 IP 的虚拟主机了。使用的指令和配置基

于名称的虚拟主机的指令是相同的，即 server_name，语法结构也相同。而且不需要考虑通配符或者正则表达式的问题。

笔者的 Nginx 配置文件中配置了两台基于 IP 的虚拟主机，相关的配置片段为：

```
...
http
{
    ...
    server                                                    #第一台虚拟主机
    {
        listen: 80;
        server_name: 192.168.1.31;
        ...
    }
    server                                                    #第二台虚拟主机
    {
        listen: 80;
        server_name: 192.168.1.32;
        ...
    }
    ...
}
```

经过以上的配置，来自 192.168.1.31 的前端请求将由第一台虚拟主机接收和处理，来自 192.168.1.32 的前端请求将由第二台虚拟主机接收和处理。

2.4.19　配置 location 块

在 Nginx 的官方文档中定义的 location 的语法结构为：

```
location [ = | ~ | ~* | ^~ ] uri { ... }
```

其中，*uri* 变量是待匹配的请求字符串，可以是不含正则表达的字符串，如 /myserver.php 等；也可以是包含有正则表达的字符串，如 \.php$（表示以.php 结尾的 URL）等。为了下文叙述方便，我们约定，不含正则表达的 *uri* 称为“标准 *uri*”，使用正则表达式的 *uri* 称为“正则 *uri*”。

其中方括号里的部分，是可选项，用来改变请求字符串与 *uri* 的匹配方式。在介绍四种标识的含义之前，我们需要先了解不添加此选项时，Nginx 服务器是如何在 server 块中搜索并使用 location 块的 *uri* 和请求字符串匹配的。

在不添加此选项时，Nginx 服务器首先在 server 块的多个 location 块中搜索是否有标准 *uri* 和请求字符串匹配，如果有多个可以匹配，就记录匹配度最高的一个。然后，服务器再用 location 块中的正则 *uri* 和请求字符串匹配，当第一个正则 *uri* 匹配成功，结束搜索，并使用这个 location 块处理此请求；如果正则匹配全部失败，就使用刚才记录的匹配度最高的 location 块处理此请求。

了解了上面的内容，就可以解释可选项中各个标识的含义了：

- “=”，用于标准 *uri* 前，要求请求字符串与 *uri* 严格匹配。如果已经匹配成功，就停止继续向下搜索并立即处理此请求。

- “~”，用于表示 *uri* 包含正则表达式，并且区分大小写。
- “~*”，用于表示 *uri* 包含正则表达式，并且不区分大小写。

注意

如果 uri 包含正则表达式，就必须要使用“~”或者“~”标识。*

- “^~”，用于标准 *uri* 前，要求 Nginx 服务器找到标识 *uri* 和请求字符串匹配度最高的 location 后，立即使用此 location 处理请求，而不再使用 location 块中的正则 *uri* 和请求字符串做匹配。

注意

我们知道，在浏览器传送 URI 时对一部分字符进行 URL 编码，比如空格被编码为“%20”，问号被编码为“%3f”等。“^~”有一个特点是，它对 uri 中的这些符号将会进行编码处理。比如，如果 location 块收到的 URI 为“/html/%20/data”，则当 Nginx 服务器搜索到配置为“^~ /html/ /data”的 location 时，可以匹配成功。

2.4.20　配置请求的根目录

Web 服务器接收到网络请求之后，首先要在服务器端指定目录中寻找请求资源。在 Nginx 服务器中，指令 root 就是用来配置这个根目录的，其语法结构为：

```
root path;
```

其中，*path* 为 Nginx 服务器接收到请求以后查找资源的根目录路径。*path* 变量中可以包含 Nginx 服务器预设的大多数变量（请参见本书“附录 A”的相关内容），只有$document_root 和$realpath_root 不可以使用。

此指令可以在 http 块、server 块或者 location 块中配置。由于使用 Nginx 服务器多数情况下要配置多个 location 块对不同的请求分别做出处理，因此该指令通常在 location 块中进行设置。

这个指令的一个示例为：

```
location /data/
{
   root /locationtest1;
}
```

当 location 块接收到“/data/index.htm”的请求时，将在/locationtest1/data/目录下找到 index.htm 响应请求。

2.4.21　更改 location 的 URI

在 location 块中，除了使用 root 指令指明请求处理根目录，还可以使用 alias 指令改变 location 接收到的 URI 的请求路径，其语法结构为：

```
alias path;
```

其中，*path* 即为修改后的根路径。同样，此变量中也可以包含除了$document_root 和$realpath_root 之外的其他 Nginx 服务器预设变量。

这个指令的作用有点不好理解，我们来看一个示例：

```
location ~ ^/data/(.+\.(htm|htm))$
```

```
{
    alias /locationtest1/other/$1;
}
```

当此 location 块接收到“/data/index.htm”的请求时，匹配成功，之后根据 alias 指令的配置，Nginx 服务器将到/locationtest1/other 目录下找到 index.htm 并响应请求。可以看到，通过 alias 指令的配置，根路径已经从/data 更改为/locationtest1/other 了。

2.4.22　设置网站的默认首页

指令 index 用于设置网站的默认首页，它一般可以有两个作用：一是，用户在发出请求访问网站时，请求地址可以不写首页名称；二是，可以对一个请求，根据其请求内容而设置不同的首页。该指令的语法结构为：

```
index file ...;
```

其中，file 变量可以包括多个文件名，其间使用空格分隔，也可以包含其他变量。此变量默认为“index.html”。

看一个示例：

```
location ~ ^/data/(.+)/web/ $
{
    index index.$1.html index.my1.html index.html;
}
```

当 location 块接收到“/data/locationtest/web/”时，匹配成功，它首先将预置变量$1 置为“locationtest”，然后在/data/locationtest/web/路径下按照 index 的配置次序依次寻找 index.locationtest.html 页、index.my1.html 页和 index.html 页，首先找到哪个页面，就使用哪个页面响应请求。

2.4.23　设置网站的错误页面

如果用户端尝试查看网页时遇到问题，服务器会将 HTTP 错误从网站发送到 Web 浏览器。如果无法显示网页，Web 浏览器会显示网站发送的实际错误网页或 Web 浏览器内置的友好错误消息。Nginx 服务器支持自定义错误网页的显示内容。可以通过这一功能在网站发生错误时为用户提供人性化的错误显示页面。

一般来说，HTTP 2XX 代表请求正常完成，HTTP 3XX 代表网站重定向，HTTP 4XX 代表客户端出现错误，HTT 5XX 代表服务器端出现错误。笔者在 Microsoft 的官方网站查找到一些常见 HTTP 错误的说明，整理在表 2.3 中，方便读者查询。

表 2.3　常见的 HTTP 错误

HTTP 消息	代码	含义
已移动	HTTP 301	请求的数据具有新的位置，并且更改是永久的
已找到	HTTP 302	请求的数据临时具有不同 URI
请参阅其他	HTTP 303	可在另一 URI 下找到对请求的响应，并且应使用 GET 方法检索此响应
未修改	HTTP 304	未按预期修改文档

续表

HTTP 消息	代码	含义
使用代理	HTTP 305	必须通过位置字段中提供的代理来访问请求的资源
未使用	HTTP 306	不再使用，但保留此代码以便将来使用
无法找到网页	HTTP 400	可以连接到 Web 服务器，但是由于 Web 地址 (URL) 的问题，无法找到网页
网站拒绝显示此网页	HTTP 403	可以连接到网站，但 Internet Explorer 没有显示网页的权限
无法找到网页	HTTP 404	可以连接到网站，但找不到网页。导致此错误的原因有时可能是该网页暂时不可用或网页已被删除
网站无法显示该页面	HTTP 405	可以连接到网站，但网页内容无法下载到用户的计算机。这通常是由网页编写方式问题引起的
无法读取此网页格式	HTTP 406	能够从网站接收信息，但 Internet Explorer 不能识别其格式，因而无法正确地显示消息
该网站太忙，无法显示此网页	HTTP 408 或 409	服务器显示该网页的时间太长，或对同一网页的请求太多
网页不复存在	HTTP 410	可以连接到网站，但无法找到网页。与 HTTP 错误 404 不同，此错误是永久性的，而且由网站管理员打开
网站无法显示该页面	HTTP 500	正在访问的网站出现了服务器问题，该问题阻止了此网页的显示。常见的原因是网站正在维护或使用脚本的交互式网站上的程序出错
未执行	HTTP 501	没有将正在访问的网站设置为显示浏览器所请求的内容
不支持的版本	HTTP 505	该网站不支持浏览器用于请求网页的 HTTP 协议（最为常见的是 HTTP/1.1）

Nginx 服务器设置网站错误页面的指令为 error_page，语法结构为：

```
error_page code ... [=[response]] uri
```

- *code*，要处理的 HTTP 错误代码，常见的在表 2.2 中已经列出。
- *response*，可选项，将 *code* 指定的错误代码转化为新的错误代码 *response*。
- *uri*，错误页面的路径或者网站地址。如果设置为路径，则是以 Nginx 服务器安装路径下的 html 目录为根路径的相对路径；如果设置为网址，则 Nginx 服务器会直接访问该网址获取错误页面，并返回给用户端。

看几个 error_page 指令的示例：

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

设置 Nginx 服务器使用“Nginx 安装路径/html/404.html”页面响应 404 错误（“无法找到网页”错误）。再如：

```
error_page 403 http://somewebsite.com/forbidden.html;
```

设置 Nginx 服务器使用“http://somewebsite.com/forbidden.html”页面响应 403 错误（“拒绝显示网页”错误）。再如：

```
error_page 410 =301 /empty.gif
```

设置 Nginx 服务器产生 410 的 HTTP 消息时，使用“Nginx 安装路径/html/empty.gif”返回给用户端 301 消息（“已移动”消息）。

在前面对 error_page 指令的分析中我们看到，变量 uri 实际上是一个相对于 Nginx 服务器安装路径的相对路径。那么，如果不想将错误页面放到 Nginx 服务器的安装路径下管理，该怎么做呢？

其实这个很简单，只需要另外使用一个 location 指令定向错误页面到新的路径下面了就可以了。比如对于上面的第一个示例，我们希望 Nginx 服务器使用“/myserver/errorpages/404.html”页面响应 404 错误，那么在设置完：

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

之后，我们再添加这样一个 location 块：

```
location /404.html
{
    root /myserver/errorpages/
}
```

首先捕获“/404.html”请求，然后将请求定向到新的路径下面即可。

errer_page 指令可以在 http 块、server 块和 location 块中配置。

2.4.24 基于 IP 配置 Nginx 的访问权限

Nginx 配置通过两种途径支持基本访问权限的控制，其中一种是由 HTTP 标准模块 ngx_http_access_module 支持的，其通过 IP 来判断客户端是否拥有对 Nginx 的访问权限，这里有两个指令需要我们学习。

allow 指令，用于设置允许访问 Nginx 的客户端 IP，语法结构为：

```
allow address | CIDR | all;
```

- *address*，允许访问的客户端的 IP，不支持同时设置多个。如果有多个 IP 需要设置，需要重复使用 allow 指令。
- *CIDR*，允许访问的客户端的 CIDR 地址，例如 202.80.18.23/25，前面是 32 位 IP 地址，后面“/25”代表该 IP 地址中前 25 位是网络部分，其余位代表主机部分。
- *all*，代表允许所有客户端访问。

从 Nginx 0.8.22 版本以后，该命令也支持 IPv6 地址，比如：

```
allow 2620:100:e000::8001;
```

另一个指令是 deny，作用刚好和 allow 指令相反，它用于设置禁止访问 Nginx 的客户端 IP，语法结构为：

```
deny address | CIDR | all;
```

- *address*，禁止访问的客户端的 IP，不支持同时设置多个。如果有多个 IP 需要设置，需要重复使用 deny 指令。
- *CIDR*，禁止访问的客户端的 CIDR 地址。
- all，代表禁止所有客户端访问。

这两个指令可以在 http 块、server 块或者 location 块中配置。在使用这两个指令时，要注意设置为 all 的用法。请看下面的例子：

```
1  location / {
2    deny  192.168.1.1;
3    allow  192.168.1.0/24;
4    deny  all;
5  }
```

在上面的配置示例中我们首先配置禁止 192.168.1.1 访问 Nginx，然后配置允许 192.168.1.0/24 访问 Nginx，最后又使用 all 配置禁止所有 IP 的访问。那么，192.168.1.0/24 客户端到底可不可以访问呢？是可以的。Nginx 配置在解析的过程中，遇到 deny 指令或者 allow 指令是按照顺序对当前客户端的连接进行访问权限检查的。如果遇到匹配的配置时，则停止继续向下搜索相关配置。因此，当 192.168.1.0/24 客户端访问时，Nginx 在第 3 行解析配置发现允许该客户端访问，就不会继续向下解析第 4 行了。

2.4.25　基于密码配置 Nginx 的访问权限

Nginx 还支持基于 HTTP Basic Authentication 协议的认证。该协议是一种 HTTP 性质的认证办法，需要识别用户名和密码，认证失败的客户端不拥有访问 Nginx 服务器的权限。该功能由 HTTP 标准模块 ngx_http_auth_basic_module 支持，这里有两个指令需要学习。

auth_basic 指令，用于开启或者关闭该认证功能，语法结构为：

```
auth_basic string | off;
```

- *string*，开启该认证功能，并配置验证时的指示信息。
- off，关闭该认证功能。

auth_basic_user_file 指令，用于设置包含用户名和密码信息的文件路径，语法结构为：

```
auth_basic_user_file file;
```

其中，*file* 为密码文件的绝对路径。

这里的密码文件支持明文或者密码加密后的文件。明文的格式如下所示：

```
name1:password1
name2:password2:comment
name3:password3
```

加密密码可以使用 crypt()函数进行密码加密的格式，在 Linux 平台上可以使用 htpasswd 命令生成。在 PHP 和 Perl 等语言中，也提供 crypt()函数。使用 htpasswd 命令的一个示例为：

```
# htpasswd  -c  -d  /nginx/conf/pass_file  username
```

运行后输入密码即可。

2.5　Nginx 服务器基础配置实例

上一节我们对 Nginx 服务器默认配置文件的结构和涉及的基本指令做了详尽的阐述。通过对这些指令的合理配置，我们就可以让一台 Nginx 服务器正常工作，并提供基本的 Web 服务器功能了。

在本节中，笔者为大家准备了一个比较完整和最简单的基础配置实例，并进行了必要的注释说明，帮助大家在整体上对 nginx.conf 文件的结构加深理解，同时巩固上一节学习的指令及其配置。

以下是笔者准备的 nginx.conf 文件的完整内容：

```
#### 全局块 开始 #####
                                        # 配置允许运行 Nginx 服务器的用户和用户组
 user nobody nobody;
                                        # 配置允许 Nginx 进程生成的 worker process 数
worker_processes  3;
                                        # 配置 Nginx 服务器运行对错误日志存放路径
error_log  logs/error.log;
                                        # 配置 Nginx 服务器运行时的 pid 文件存放路径和名称
pid    nginx.pid;
##### 全局块 结束 #####
#### events 块 开始 ####
events
{
                                        # 配置事件驱动模型
   use epoll;
                                        # 配置最大连接数
    worker_connections  1024;
}
#### events 块 结束 ####
#### http 块 开始 ####
http {
                                        # 定义 MIME-Type
    include      mime.types;
    default_type  application/octet-stream;
                                        # 配置允许使用 sendfile 方式传输
    sendfile  on;
                                        # 配置连接超时时间
   keepalive_timeout  65;
                                        # 配置请求处理日志的格式
    log_format  access.log
'$remote_addr-[$time_local]-"$request"-"$http_user_agent"';
    #### server 块 开始 ####
   ## 配置虚拟主机 myServer1
   server {
                                        # 配置监听端口和主机名称（基于名称）
       listen      8081;
      server_name  myServer1;
                                        # 配置请求处理日志存放路径
       access_log   /myweb/server1/log/access.log;
                                        # 配置错误页面
       error_page  404    /404.html;
                                        # 配置处理/server1/location1 请求的 location
```

```
        location /server1/location1 {
            root  /myweb;
            index  index.svr1-loc1.htm;
        }
                                              # 配置处理/server1/location2 请求的 location
        location /server1/location2 {
            root  /myweb;
          index  index.svr1-loc2.htm;
        }
    }
                                              ## 配置虚拟主机 myServer2
     server {
        listen       8082;
       server_name  192.168.1.3;
        access_log   /myweb/server2/log/access.log;
       error_page   404  /404.html;          #对错误页面 404.html 做了定向配置
       location /server2/location1 {
            root  /myweb;
            index  index.svr2-loc1.htm;
        }
       location /svr2/loc2 {
           alias  /myweb/server2/location2/;          #对 location 的 URI 进行更改
           index  index.svr2-loc2.htm;
        }
                                              # 配置错误页面转向
        location = /404.html {
           root   /myweb/;
            index 404.html;
       }
    }
    #### server 块 结束 ####
}
93  #### http 块 结束 ####
```

在该实例中，我们配置了两个虚拟主机 myServer1 和 myServer2，前者是基于名称的，后者是基于 IP 的。在每个虚拟主机里，笔者又分别使用了 location 块对不同的请求进行处理。主机 myServer2 除了对一般的请求进行处理外，还对错误页面 404.html 做了定向配置，指向笔者自定义的 404 处理页面。

笔者为了测试此 Nginx 配置，构建了一个十分简单的静态网站站点。我们没有必要关心站点的具体内容，主要看一下它的目录组织结构：

```
myweb
    404.html
    server1
        location1
            index.svr1-loc1.htm
        location2
            index.svr1-loc2.htm
```

```
    log
        access.log
server2
    location1
        index.svr2-loc1.htm
    location2
        index.svr2-loc2.htm
    log
        access.log
```

上面的目录结构是比较清晰的，建议读者结合网站结构阅读上面的配置文件实例，加深理解。在结构中需要注意的是 404.html 这个文件。如果不使用 myServer2 中对错误页面定向的方法，而只是像 myServer1 那样简单设置错误页面的路径，那么 Nginx 服务器会在安装路径的 html 目录下的相关路径中寻找错误页面。在介绍 error_page 指令时我们已经提到过这一点，在这里再次强调一下。在该实例中出现的其他指令和配置细节都在前面详细讲解过，就不在此赘述了。

以下是笔者测试通过 Nginx 服务器访问 myweb 站点的一些结果。

2.5.1　测试 myServer1 的访问

在用户端浏览器中输入“myserver1:8081/svr1/location1/”，可以看到网站页面的显示如图 2.3 所示，这说明主机访问正常，也表明配置文件实例中 45 ~ 46 行及 54 ~ 58 行的配置正确。

同样，在用户端浏览器地址中输入“myserver1:8081/server1/location2/”，则可以看到网站页面的显示如图 2.4 所示。

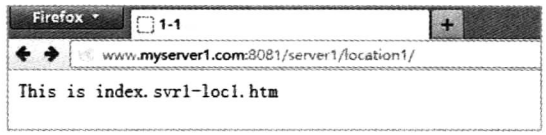

图 2.3　访问 myServer1（请求 location1）

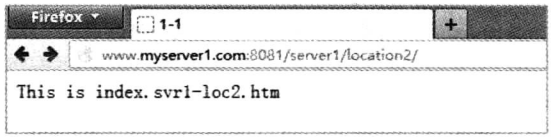

图 2.4　访问 myServer1（请求 location2）

注意

为了能够以主机名的方式访问站点，我们需要配置自己的 DNS 服务器，使得浏览器能够根据输入的域名地址（myserver1）查找到对应的 IP 地址（192.168.1.3）。

2.5.2　测试 myServer2 的访问

在用户端浏览器中输入“192.168.1.3:8082/server2/location1/”，可以看到网站页面的显示如图 2.5 所示，这说明主机访问正常，也表明配置文件实例中 69 ~ 70 行及 75 ~ 78 行的配置正确。

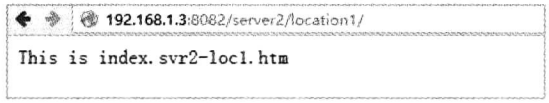

图 2.5　访问 myServer2（请求 location1）

在用户端浏览器中输入“192.168.1.3:8082/svr2/loc2/”，则可以看到网站页面的显示如图 2.6 所示，这说明主机访问正常，也表明配置文件实例中 80 ~ 83 行对 location 块的 URI 路径更改生效。

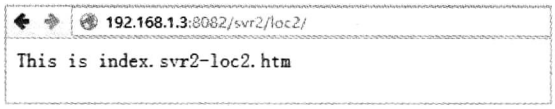

图 2.6　访问 myServer2（请求 location2）

如果在用户浏览器中输入“http://192.168.1.3:8082/server2/location2/”，由于使用 location 块对 URL 路径做了更改，导致无法找到网页，Nginx 服务器返回 404 错误消息，如图 2.7 所示。404 错误的页面是笔者自己定义的，这说明配置文件实例中 85 ~ 89 行对 404 错误页面的定向生效。

图 2.7　返回自定义的错误页面

2.6　本章小结

到此，我们对 Nginx 服务器的安装部署和基础配置就介绍完了。在本章中，我们首先学习了如何获取和安装部署 Nginx 服务器，然后学习了 Nginx 服务的启停控制以及平滑重启、平滑升级等操作；之后我们从 nginx.conf 文件的默认内容出发，首先学习了 Nginx 配置文件的组成结构和每个配置块的功能；接着结合示例对 nginx.conf 文件中出现的基本配置指令的功能、用法和作用域等做了详细的介绍，并且使用一个简单的实例来验证和巩固前面介绍的各种指令的配置方法。熟练使用这些指令就能够让 Nginx 提供基本的 Web 服务器功能了。

在下一章中，我们将接触到 Nginx 服务器的主要特点之———模块化架构。这一章的内容对我们进一步理解和记忆本章介绍的 Nginx 服务器基础配置有着重要的作用。

第 3 章

Nginx 服务器架构初探

通过前面两章的介绍，相信大家对 Nginx 服务器的来历发展、功能服务有了大致的认识，同时也掌握了初步使用 Nginx 服务器的一些知识。但是，在进行了一系列应用实践后，相信读者会发现自己对前面的相关知识产生了似懂非懂的感觉，一方面可以比较熟练地使用 Nginx 服务器提供基本服务，另一方面却没有很清晰的思路讲清楚 Nginx 服务器是怎样完成强大的 Web 服务功能的。那么，在这一章中，我们将首次接触 Nginx 服务器的架构设计，它将作为我们深入探索 Nginx 世界的开端。

在本章中，我们将学习到以下主要内容：

- 模块化结构的相关知识。
- Nginx 如何处理 Web 请求。
- Nginx 的事件驱动模型。
- Nginx 设计架构的概览。

3.1 模块化结构

从初学者到资深程序员，对于“模块化”概念的理解可谓仁者见仁、智者见智，市面上也有不少书籍专门研究相关的理论和应用。“模块化”思想不是本书讨论的重点，但是谈到 Nginx 服务器的架构，就不能不谈到它。于是，在本章伊始，我们先使用较少的篇幅对模块化简单介绍一下，作为后面深入介绍 Nginx 架构的预备知识，如果想深入了解的话则可以进一步学习模块化设计相关的知识。

3.1.1 什么是“模块化设计”

到底什么是“模块化设计”呢？还是那句老话，“没有统一的定义”。在 20 世纪 50 年代，欧美一

些国家正式提出“模块化设计”的概念，把模块化设计提到理论高度来研究。目前，模块化设计的思想已渗透到许多领域。在计算机领域，针对程序设计，常见的说法是把“模块化设计”定义为“以功能块为单位进行程序设计，实现其求解算法的方法”。从这个一般提法中，我们能够看到几层意思。

第一，“功能块”是对模块的描述，一个模块就是一个功能块，应该只负责一个功能，在设计模式理论中类似于经常提到的“单一职责原则”。

第二，如果要体现模块化，就免不了将程序进行分解，这也是模块化编程的另一个原则——自顶向下，逐步求精原则。

第三，一个程序被分解为多个模块，那么它们之间一定要存在一定的依赖关系，但是这个依赖不能太强，否则也就不能称之为“模块化”了。于是，又涉及到模块化编程的一条原则：高内聚、低耦合原则。事实上，在设计模式理论中，也有对应的一条设计原则叫“迪米特原则”。

这个提法把“模块化编程”定义为程序设计的一种方法，这种方法的结果是用一系列以功能块为单位的算法来描述和实现程序。相对于这个一般提法，笔者更认为“模块化”是一种思想，它将人们在日常生活中处理事情的习惯性思维应用到了程序设计当中，这是程序设计历史上一次里程碑式的思想飞跃。接触过面向对象理论的读者对此应该深有体会，也更能够理解模块化编程的意义所在。

模块化设计支持分布式开发。模块化的思想导致大量功能独立的模块出现，这些模块可以分布在世界上的任何角落。无论是开发中小型的应用程序，还是构建大型的诸如操作系统之类的程序，都可以采用分布式开发模型，集合任何可用模块为己所用。

模块化设计支持团队协同合作。在采用模块化思想进行程序设计时，最终的产品由小的、分散的功能块组成，每一块都是基本独立的。这些功能块可以由不同的团队根据他们自己的时间表和生命周期进行开发，互不影响。最终的产品则可以由另一个独立的个体——发行者进行集成。

模块化设计支持应用扩展和升级。应用模块化思想设计出来的程序，就如同用积木搭起来的房子。各个模块之间既能保持自己的独立性，也能通过接口保持联系。在对应用扩展时，只要实现规范的接口，就可以不断加入新功能；在对其中某些模块进行升级时，只要保持原有接口不变，就能够在不影响其他模块的前提下进行。

本小节主要对“模块化设计”思想的定义、应该遵循的一些原则以及它的应用意义做了简单阐述。Nginx 服务器的开发完全遵循模块化设计思想。在本书的后续内容中，相信读者能够深刻感受到这一思想给 Nginx 带来的巨大优越性，尤其是在应用扩展和升级方面，体现得更加淋漓尽致。

3.1.2　Nginx 模块化结构

在前面相关章节中多次提到了 Nginx 的模块。习惯上将 Nginx 涉及到的模块分为核心模块、标准 HTTP 模块、可选 HTTP 模块、邮件服务模块以及第三方模块等五大类。

核心模块是指 Nginx 服务器正常运行必不可少的模块，它们提供了 Nginx 最基本最核心的服务，如进程管理、权限控制、错误日志记录等；标准 HTTP 模块是通过第 2 章介绍的方法快速编译 Nginx 后包含的模块，其支持 Nginx 服务器的标准 HTTP 功能；可选 HTTP 模块主要用于扩展标准的 HTTP 功能，使其能够处理一些特殊的 HTTP 请求；邮件服务模块主要用于支持 Nginx 的邮件服务；第三方模块是为了扩展 Nginx 服务器应用，完成特殊功能而由第三方机构或者个人编写的可编译到 Nginx 中

的模块。Nginx 的每个模块都基本符合单一职责原则，在具体环境中可以根据实际情况裁减和加入。

核心模块和标准 HTTP 模块在 Nginx 快速编译后就包含在 Nginx 中。在 Linux 系统中，将工作目录定位到第 2 章中编译 Nginx-1.2.3 的路径/Nginx_123/nginx-1.2.3 下，可以看到 objs 目录。objs 目录中包含了这些内容：

```
# ls objs/
Makefile  autoconf.err  nginx  nginx.8  ngx_auto_config.h  ngx_auto_headers.h
ngx_modules.c  ngx_modules.o  src
```

在此目录中的 ngx_modules.c 文件中包含了此版本 Nginx 快速编译后包括的所有固有模块的声明。这些模块声明以 extern 关键字修饰：

```
# cat ngx_modules.c|grep extern|cat -n
   extern ngx_module_t  ngx_core_module;
   extern ngx_module_t  ngx_errlog_module;
   extern ngx_module_t  ngx_conf_module;
   extern ngx_module_t  ngx_events_module;
   extern ngx_module_t  ngx_event_core_module;
   extern ngx_module_t  ngx_epoll_module;
   extern ngx_module_t  ngx_regex_module;
   extern ngx_module_t  ngx_http_module;
   extern ngx_module_t  ngx_http_core_module;
   extern ngx_module_t  ngx_http_log_module;
   extern ngx_module_t  ngx_http_upstream_module;
   extern ngx_module_t  ngx_http_static_module;
   extern ngx_module_t  ngx_http_autoindex_module;
   extern ngx_module_t  ngx_http_index_module;
   extern ngx_module_t  ngx_http_auth_basic_module;
   extern ngx_module_t  ngx_http_access_module;
   extern ngx_module_t  ngx_http_limit_conn_module;
   extern ngx_module_t  ngx_http_limit_req_module;
   extern ngx_module_t  ngx_http_geo_module;
   extern ngx_module_t  ngx_http_map_module;
   extern ngx_module_t  ngx_http_split_clients_module;
   extern ngx_module_t  ngx_http_referer_module;
   extern ngx_module_t  ngx_http_rewrite_module;
   extern ngx_module_t  ngx_http_proxy_module;
   extern ngx_module_t  ngx_http_fastcgi_module;
   extern ngx_module_t  ngx_http_uwsgi_module;
   extern ngx_module_t  ngx_http_scgi_module;
   extern ngx_module_t  ngx_http_memcached_module;
   extern ngx_module_t  ngx_http_empty_gif_module;
   extern ngx_module_t  ngx_http_browser_module;
   extern ngx_module_t  ngx_http_upstream_ip_hash_module;
   extern ngx_module_t  ngx_http_upstream_least_conn_module;
   extern ngx_module_t  ngx_http_upstream_keepalive_module;
   extern ngx_module_t  ngx_http_write_filter_module;
```

```
extern ngx_module_t  ngx_http_header_filter_module;
extern ngx_module_t  ngx_http_chunked_filter_module;
extern ngx_module_t  ngx_http_range_header_filter_module;
extern ngx_module_t  ngx_http_gzip_filter_module;
extern ngx_module_t  ngx_http_postpone_filter_module;
extern ngx_module_t  ngx_http_ssi_filter_module;
extern ngx_module_t  ngx_http_charset_filter_module;
extern ngx_module_t  ngx_http_userid_filter_module;
extern ngx_module_t  ngx_http_headers_filter_module;
extern ngx_module_t  ngx_http_copy_filter_module;
extern ngx_module_t  ngx_http_range_body_filter_module;
extern ngx_module_t  ngx_http_not_modified_filter_module;
```

由于使用 extern 关键字修饰，因此各模块均可以被其他模块访问。

这里简要说明一下 Nginx 中模块的命名习惯。一般以 ngx_作为前缀，_module 作为后缀，中间使用一个或者多个英文单词描述模块的功能。比如 ngx_core_module，中间的 core 表明该模块提供了 Nginx 程序的核心功能；再如 ngx_events_module，中间的 events 表明该模块提供了解析配置文件中 events 块的功能；再如 ngx_http_core_module，中间的 http_core 表明该模块提供了 Nginx 程序 http 服务的核心功能，等等。了解了 Nginx 中模块的命名习惯，再阅读上面列出的模块，就可以大致了解 Nginx 服务器在发布时能提供的主要服务了。

所有固有模块的源码放在编译目录下的 src 目录中。在 src 目录中，我们看到一共分成了 core、event、http、mail、misc 和 os 等 6 个目录。从这里看到，源码中包含了邮件服务的模块，但在快速编译时默认不将其编译到 Nginx 中。

1. 核心模块

3.1.2 小节加粗部分的模块提供 Nginx 的核心功能。详细来说，核心模块主要包含对两类功能的支持，一类是主体功能，包括进程管理、权限控制、错误日志记录、配置解析等，另一类是用于响应请求事件必需的功能，包括事件驱动机制、正则表达式解析等。

2 .标准 HTTP 模块

在第 1 章中，我们提到 Nginx 服务器主要提供基本 HTTP 服务、高级 HTTP 服务和邮件服务等。这一模块对应于基本 HTTP 服务。

这些模块在默认情况下是被编译到 Nginx 中的，除非在配置时添加--without-XXX 参数声明不编译。在表 3.1 中，笔者对上面列表中比较重要的标准 HTTP 模块进行了梳理并添加了说明，方便读者查询和理解。

表 3.1　常用标准 HTTP 模块

模块	功能
ngx_http_core	配置端口、URI 分析、服务器响应错误处理、别名控制以及其他 HTTP 核心事务
ngx_http_access_module	基于 IP 地址的访问控制（允许/拒绝）
ngx_http_auth_basic_module	基于 HTTP 的身份认证

续表

模块	功能
ngx_http_autoindex_module	处理以“/”结尾的请求并自动生成目录列表
ngx_http_browser_module	解析 HTTP 请求头中的“User-Agent”域的值
ngx_http_charset_module	指定网页编码
ngx_http_empty_gif_module	从内存创建一个 1×1 的透明 gif 图片，可以快速调用
ngx_http_fastcgi_module	对 FastCGI 的支持（有关 FastCGI 的细节参见相关章节）
ngx_http_geo_module	将客户端请求中的参数转化为键值对变量
ngx_http_gzip_module	压缩请求响应，可以减少数据传输
ngx_http_headers_filter_module	设置 HTTP 响应头
ngx_http_index_module	处理以“/”结尾的请求，如果没有找到该目录下的 index 页，就将请求转给 ngx_http_autoindex_module 模块处理；如果 Nginx 服务器开启了 ngx_http_random_index_module 模块，则随机选择 index 页
ngx_http_limit_req_module	限制来自客户端的请求的响应和处理速率
ngx_http_limit_conn_module	限制来自客户端的连接的响应和处理速率
ngx_http_log_module	自定义 access 日志（参见 2.3.2 节中“自定义服务日志”的相关内容）
ngx_http_map_module	创建任意键值对变量
ngx_http_memcached_module	对 Memcached 的支持（有关 Memcached 的细节参见相关章节）
ngx_http_proxy_module	支持代理服务（有关代理服务的细节参见相关章节）
ngx_http_referer_module	过滤 HTTP 头中“Referer”域值为空的 HTTP 请求
ngx_http_rewrite_module	通过正则表达式重定向请求（有关请求重定向的细节参见相关章节）
ngx_http_scgi_module	对 SCGI 的支持（有关 SCGI 的细节参见相关章节）
ngx_http_ssl_module	对 HTTPS 的支持
ngx_http_upstream_module	定义一组服务器，可以接收来自代理、Fastcgi、Memcached 的重定向，主要用于负载均衡（有关负载均衡的细节参见相关章节）

对于其他未涉及的标准 HTTP 模块，有兴趣的读者可以访问 Nginx 的官方网站查找相关内容。

3. 可选 HTTP 模块

可选 HTTP 模块在目前的 Nginx 发行版本中只提供源码，但在快速编译时默认不编译。如果想使用相关模块，就必须在配置时使用--with-XXX 参数声明。

在表 3.2 中，笔者对常见的可选 HTTP 模块进行了整理和说明，便于读者查询。

表 3.2 常用可选 HTTP 模块

模块	功能
ngx_http_addition_module	在响应请求的页面开始或者结尾添加文本信息
ngx_http_degradation_module	在低内存的情形下允许 Nginx 服务器返回 444 错误或 204 错误

续表

模块	功能
ngx_http_perl_module	在 Nginx 的配置文件中可以使用 Perl 脚本
ngx_http_flv_module	支持将 Flash 多媒体信息按照流文件传输，可以根据客户端指定的开始位置返回 Flash
ngx_http_geoip_module	支持解析基于 GeoIP 数据库的客户端请求（关于 GeoIP 数据库的细节请查看其官方网站 http://www.maxmind.com/）
ngx_google_perftools_module	支持 Google Performance Tools（Google Performance Tools 是 Google 公司开发的一套用于 C++ Profile 的工具集，细节请查看其官方网站 http://code.google.com/p/gperftools/）
ngx_http_gzip_module	支持在线实时压缩响应客户端的输出数据流
ngx_http_gzip_static_module	搜索并使用预压缩的以“.gz”为后缀名的文件代替一般文件响应客户端请求
ngx_http_image_filter_module	支持改变 JPEG、GIF 和 PNG 图片的尺寸和旋转方向
ngx_http_mp4_module	支持将 H.264/AAC 编码的多媒体信息（后缀名通常为 mp4、m4v 或 m4a）按照流文件传输，常与 ngx_http_flv_module 模块一起使用
ngx_http_random_index_module	Nginx 接收到以“/”结尾的请求时，在对应的目录下随机选择一个文件作为 index 文件。参见表 3.1 中的 ngx_http_index_module 模块
ngx_http_secure_link_module	支持对请求链接的有效性检查
ngx_http_ssl_module	对 HTTPS/SSL 的支持
ngx_http_stub_status_module	支持返回 Nginx 服务器的统计信息，一般包括处理连接的数量、连接成功的数量、处理的请求数、读取和返回的 Header 信息数等信息
ngx_http_sub_module	使用指定的字符串替换响应信息中的信息
ngx_http_dav_module	支持 HTTP 协议和 WebDAV 协议中 PUT、DELETE、MKCOL、COPY 和 MOVE 方法
ngx_http_xslt_module	将 XML 响应信息使用 XSLT（扩展样式表转换语言）进行转换

4. 邮件服务模块

在第 1 章中提到，邮件服务是 Nginx 服务器提供的主要服务之一。但是在目前的 Nginx 发行版本中，快速编译时默认并不会编译邮件服务模块。

和 Nginx 服务器提供的邮件服务相关的模块有：

- ngx_mail_core_module
- ngx_mail_pop3_module
- ngx_mail_imap_module
- ngx_mail_smtp_module
- ngx_mail_auth_http_module
- ngx_mail_proxy_module
- ngx_mail_ssl_module

这些模块完成了邮件服务的主要功能，包括对 POP3 协议、IMAP 协议和 SMTP 协议的支持，对身份认证、邮件代理和 SSL 安全服务的提供。

5. 第三方模块

由于 Nginx 支持自定义模块编程，第三方模块不断得到扩充，功能也非常丰富。目前，记录在 wiki 站点的就多达 90 个，而且还有一些模块是没有包含在内的。对于繁多的第三方模块，我们不打算在这里一一列举，读者可以根据自己的需要从 wiki 站点自行查找。

在第三方模块的开发作者中，深受广大 Nginx 用户推崇的要算是一位笔名为 agentzh 的工程师了。他开发的 echo-nginx-module 模块（支持在 Nginx 配置文件中使用 echo、sleep、time 及 exec 等类 Shell 命令）、memc-nginx-module 模块（对标准 HTTP 模块 ngx_http_memcached_module 的扩展，支持 set、add、delete 等更多的命令）、rds-json-nginx-module 模块（使 Nginx 支持 Json 数据的处理）、lua-nginx-module 模块（使 Nginx 支持 lua 脚本语言）等都是笔者在日常工作中经常使用的。有兴趣的读者可以到 wiki 站点下载使用。

到此，我们介绍完了 Nginx 的五大类模块，大家对前面四种包含的具体模块及其功能有了比较清晰的认识。从前面的内容我们能够看出，Nginx 服务器在功能定制和扩展上具有其他 Web 服务器无法媲美的巨大优势，从核心功能到一般功能，再到扩展功能，几乎都可以使用“模块化”技术实现。

那么，这些模块彼此是如何组织在一起，为 Nginx 服务器提供支持的呢？从下一节开始，我们更进一步从 Nginx 服务器的设计架构出发，来全面了解各个模块之间的联系。

3.2　Nginx 服务器的 Web 请求处理机制

从设计架构上来说，Nginx 服务器是与众不同的。不同之处一方面体现在它的模块化设计，另一方面，也是更重要的一方面，体现在它对客户端请求的处理机制上。

Web 服务器和客户端是一对多的关系，Web 服务器必须有能力同时为多个客户端提供服务。一般来说，完成并行处理请求工作有三种方式可供选择：多进程方式、多线程方式和异步方式。

3.2.1　多进程方式

多进程方式是指，服务器每当接收到一个客户端时，就由服务器主进程生成一个子进程出来和该客户端建立连接进行交互，直到连接断开，该子进程就结束了。

多进程方式的优点在于，设计和实现相对简单，各个子进程之间相互独立，处理客户端请求的过程彼此不受到干扰，并且当一个子进程产生问题时，不容易将影响漫延到其他进程中，这保证了提供服务的稳定性。当子线程退出时，其占用资源会被操作系统回收，也不会留下任何垃圾。而其缺点也是很明显的。操作系统中生成一个子进程需要进行内存复制等操作，在资源和时间上会产生一定的额外开销，因此，如果 Web 服务器接收大量并发请求，就会对系统资源造成压力，导致系统性能下降。

初期的 Apache 服务器就是采用这种方式对外提供服务的。为了应对大量并发请求，Apache 服务器采用“预生成进程”的机制对多进程方式进行了改进。“预生成进程”的工作方式很好理解。它将生成子进程的时机提前，在客户端请求还没有到来之前就预先生成好，当请求到来时，主进程分配一个

子进程和该客户端进行交互，交互完成之后，该进程也不结束，而被主进程管理起来等待下一个客户端请求的到来。改进的多进程方式在一定程度上缓解了大量并发请求情形下 Web 服务器对系统资源造成的压力。但是由于 Apache 服务器在最初的架构设计上采用了多进程方式，因此这不能从根本上解决问题。

3.2.2　多线程方式

多线程方式和多进程方式相似，它是指，服务器每当接收到一个客户端时，会由服务器主进程派生一个线程出来和该客户端进行交互。

由于操作系统产生一个线程的开销远远小于产生一个进程的开销，所以多线程方式在很大程度上减轻了 Web 服务器对系统资源的要求。该方式使用线程进行任务调度，开发方面可以遵循一定的标准，这相对来说比较规范和有利于协作。但在线程管理方面，该方式有一定的不足。多个线程位于同一个进程内，可以访问同样的内存空间，彼此之间相互影响；同时，在开发过程中不可避免地要由开发者自己对内存进行管理，其增加了出错的风险。服务器系统需要长时间连续不停地运转，错误的逐渐积累可能最终对整个服务器产生重大影响。

IIS 服务器使用了多线程方式对外提供服务，它的稳定性相对来说还是不错的，但对于经验丰富的 Web 服务器管理人员而言，他们通常还是会定期检查和重启服务器，以预防不可预料的故障发生。

3.2.3　异步方式

异步方式是和多进程方式及多线程方式完全不同的一种处理客户端请求的方式。在介绍该方式之前，我们先复习一下同步、异步以及阻塞、非阻塞的概念。

网络通信中的同步机制和异步机制是描述通信模式的概念。同步机制，是指发送方发送请求后，需要等待接收到接收方发回的响应后，才接着发送下一个请求；异步机制，和同步机制正好相反，在异步机制中，发送方发出一个请求后，不等待接收方响应这个请求，就继续发送下个请求。在同步机制中，所有的请求在服务器端得到同步，发送方和接收方对请求的处理步调是一致的；在异步机制中，所有来自发送方的请求形成一个队列，接收方处理完成后通知发送方。

阻塞和非阻塞用来描述进程处理调用的方式，在网络通信中，主要指网络套接字 Socket 的阻塞和非阻塞方式，而 Socket 的实质也就是 IO 操作。Socket 的阻塞调用方式为，调用结果返回之前，当前线程从运行状态被挂起，一直等到调用结果返回之后，才进入就绪状态，获取 CPU 后继续执行；Socket 的非阻塞调用方式和阻塞调用方式正好相反，在非阻塞方式中，如果调用结果不能马上返回，当前线程也不会被挂起，而是立即返回执行下一个调用。

在网络通信中，经常可以看到有人将同步和阻塞等同、异步和非阻塞等同。事实上，这两对概念有一定的区别，不能混淆。两对概念的组合，就会产生四个新的概念，同步阻塞、异步阻塞、同步非阻塞、异步非阻塞。

- 同步阻塞方式，发送方向接收方发送请求后，一直等待响应；接收方处理请求时进行的 IO 操作如果不能马上得到结果，就一直等到返回结果后，才响应发送方，期间不能进行其他工作。比如，在超市排队付账时，客户（发送方）向收款员（接收方）付款（发送请求）后需要等待收款员找零，期间不能做其他的事情；而收款员要等待收款机返回结果（IO 操作）

后才能把零钱取出来交给客户（响应请求），期间也只能等待，不能做其他事情。这种方式实现简单，但是效率不高。

- 同步非阻塞方式，发送方向接收方发送请求后，一直等待响应；接收方处理请求时进行的 IO 操作如果不能马上得到结果，就立即返回，去做其他事情，但由于没有得到请求处理结果，不响应发送方，发送方一直等待。一直到 IO 操作完成后，接收方获得结果响应发送方后，接收方才进入下一次请求过程。在实际中不使用这种方式。
- 异步阻塞方式，发送方向接收方发送请求后，不用等待响应，可以接着进行其他工作；接收方处理请求时进行的 IO 操作如果不能马上得到结果，就一直等到返回结果后，才响应发送方，期间不能进行其他工作。这种方式在实际中也不使用。
- 异步非阻塞方式，发送方向接收方发送请求后，不用等待响应，可以继续其他工作；接收方处理请求时进行的 IO 操作如果不能马上得到结果，也不等待，而是马上返回去做其他事情。当 IO 操作完成以后，将完成状态和结果通知接收方，接收方再响应发送方。继续使用在超市排队付账的例子。客户（发送方）向收款员（接收方）付款（发送请求）后在等待收款员找零的过程中，还可以做其他事情，比如打电话、聊天等；而收款员在等待收款机处理交易（IO 操作）的过程中可以帮助客户将商品打包，当收款机产生结果后，收款员给客户结账（响应请求）。在四种方式中，这种方式是发送方和接收方通信效率最高的一种。

3.2.4 Nginx 服务器如何处理请求

Nginx 服务器的一个显著优势是能够同时处理大量并发请求。它结合多进程机制和异步机制对外提供服务。异步机制使用的是异步非阻塞方式。

在 2.3.2 节中，我们介绍过 Nginx 服务器启动后，可以产生一个主进程（master process）和多个工作进程（worker processes），其中可以在配置文件中指定产生的工作进程数量。Nginx 服务器的所有工作进程都用于接收和处理客户端的请求。这类似于 Apache 使用的改进的多进程机制，预先生成多个工作进程，等待处理客户端请求。

注意

实际上，Nginx 服务器的进程模型有两种：Single 模型和 Master-Worker 模型。Single 模型为单进程方式，性能较差，一般在实际工作中不使用。Master-Worker 模型实际上被更广泛地称为 Master-Slave 模型。在 Nginx 服务器中，充当 Slave 角色的是工作进程。

每个工作进程使用了异步非阻塞方式，可以处理多个客户端请求。当某个工作进程接收到客户端的请求以后，调用 IO 进行处理，如果不能立即得到结果，就去处理其他的请求；而客户端在此期间也无需等待响应，可以去处理其他的事情；当 IO 调用返回结果时，就会通知此工作进程；该进程得到通知，暂时挂起当前处理的事务，去响应客户端请求。

客户端请求数量增长、网络负载繁重时，Nginx 服务器使用多进程机制能够保证不增长对系统资源的压力；同时使用异步非阻塞方式减少了工作进程在 I/O 调用上的阻塞延迟，保证了不降低对请求的处理能力。

3.2.5　Nginx 服务器的事件处理机制

在上一节中我们提到，Nginx 服务器的工作进程调用 IO 后，就去进行其他工作了；当 IO 调用返回后，会通知工作进程。这里有一个问题，IO 调用是如何把自己的状态通知给工作进程的呢?

一般解决这个问题的方案有两种。一是，让工作进程在进行其他工作的过程中间隔一段时间就去检查一下 IO 的运行状态，如果完成，就去响应客户端，如果未完成，就继续正在进行的工作；二是，IO 调用在完成后能主动通知工作进程。对于前者，虽然工作进程在 IO 调用过程中没有等待，但不断的检查仍然在时间和资源上导致了不小的开销，最理想的解决方案是第二种。

具体来说，select/poll/epoll/kqueue 等这样的系统调用就是用来支持第二种解决方案的。这些系统调用，也常被称为事件驱动模型，它们提供了一种机制，让进程可以同时处理多个并发请求，不用关心 IO 调用的具体状态。IO 调用完全由事件驱动模型来管理，事件准备好之后就通知工作进程事件已经就绪。

3.3　Nginx 服务器的事件驱动模型

事件驱动模型是 Nginx 服务器保障完整功能和具有良好性能的重要机制之一。

3.3.1　事件驱动模型概述

实际上，事件驱动并不是计算机编程领域的专业词汇，它是一种比较古老的响应事件的模型，在计算机编程、公共关系、经济活动等领域均有很广泛的应用。顾名思义，事件驱动就是在持续事务管理过程中，由当前时间点上出现的事件引发的调动可用资源执行相关任务，解决不断出现的问题，防止事务堆积的一种策略。在计算机编程领域，事件驱动模型对应一种程序设计方式，Event-driven programming，即事件驱动程序设计。

如图 3.1 所示，事件驱动模型一般是由事件收集器、事件发送器和事件处理器三部分基本单元组成。

其中，事件收集器专门负责收集所有的事件，包括来自用户的（如鼠标单击事件、键盘输入事件等）、来自硬件的（如时钟事件等）和来自软件的（如操作系统、应用程序本身等）。事件发送器负责将收集器收集到的事件分发到目标对象中。目标对象就是事件处理器所处的位置。事件处理器主要负责具体事件的响应工作，它往往要到实现阶段才完全确定。

在程序设计过程中，对事件驱动机制的实现方式有多种，这里介绍 batch programming，即批次程序设计。批次的程序设计是一种比较初级的程序设计方式。使用批次程序设计的软件，其流程是由程序设计师在设计编码过程中决定的，也就是说，在程序运行的过程中，事件的发生、事件的发送和事件的处理都是预先设计好的。由此可见，事件驱动程序设计更多的关注了事件产生的随机性，使得应用程序能够具备相当的柔性，可以应付种种来自用户、硬件和系统的离散随机事件，这在很大程度上增强了用户和软件的交互性和用户操作的灵活性。

事件驱动程序可以由任何编程语言来实现，只是难易程度有别。如果一个系统是以事件驱动程序模型作为编程基础的，那么，它的架构基本上是这样的：预先设计一个事件循环所形成的程序，这个

事件循环程序构成了如图 3.1 中所示的“事件收集器”，它不断地检查目前要处理的事件信息，然后使用“事件发送器”传递给“事件处理器”。“事件处理器”一般运用虚函数机制来实现。

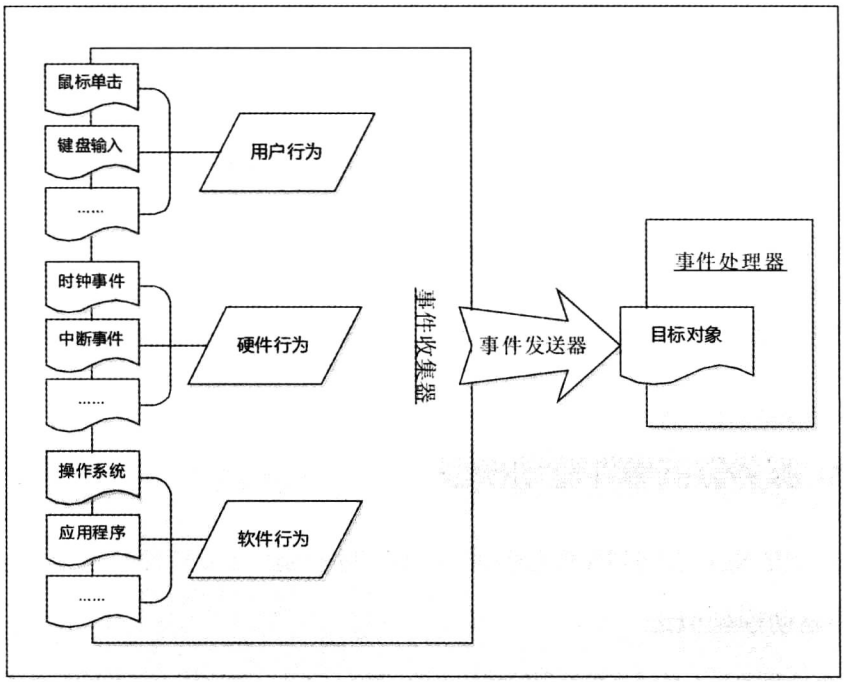

图 3.1　事件驱动模型

我们日常使用的 Windows 操作系统，就是基于事件驱动程序设计的典型实例。Windows 操作系统中的视图（通常叫做“窗口”），是我们所说的事件发送器的目标对象。视图接收事件并能够对其进行相应的处理。当我们将事件发送到具体的某一个视图的时候，实际上我们完成了从传统的流线型程序结构到事件触发方式的转变。

在事件驱动程序的基本单元中，事件收集器已经由 Windows 操作系统完成；因为 Windows 操作系统是用 C 语言实现的，而不是 C++语言编写的，所以没有对象的概念，这导致了 Windows 操作系统只能将发生的事件发送到所谓的“窗口函数”中。事实上，事件尽管不是被发送到具体的对象，但应该说，这是 C 语言对面向对象方式实现的一个变体。在这里我们可以看到，事件发送器也已经由 Windows 操作系统完成了部分内容，其中，确定事件的目标所要做的工作的复杂程度可能要超出我们的想象。

3.3.2　Nginx 中的事件驱动模型

Nginx 服务器响应和处理 Web 请求的过程，就是基于事件驱动模型的，它也包含事件收集器、事件发送器和事件处理器等三部分基本单元。通过上面的内容，大家应该已经了解了事件驱动模型的基本概念。那么，Nginx 服务器是如何使用事件驱动模型来工作的呢？它的“事件收集器”和“事件发送器”的实现没有太大的特点，我们重点介绍一下它的“事件处理器”。

通常，我们在编写服务器处理模型的程序时，基于事件驱动模型，“目标对象”中的“事件处理器”

可以有以下几种实现办法：

- “事件发送器”每传递过来一个请求，“目标对象”就创建一个新的进程，调用“事件处理器”来处理该请求。
- “事件发送器”每传递过来一个请求，“目标对象”就创建一个新的线程，调用“事件处理器”来处理该请求。
- “事件发送器”每传递过来一个请求，“目标对象”就将其放入一个待处理事件的列表，使用非阻塞 I/O 方式调用“事件处理器”来处理该请求。

上面的三种处理方式，各有特点，第一种方式，由于创建新的进程的开销比较大，会导致服务器性能比较差，但其实现相对来说比较简单；第二种方式，由于要涉及到线程的同步，故可能会面临死锁、同步等一系列问题，编码比较复杂；第三种方式，在编写程序代码时，逻辑比前面两种都复杂。大多数网络服务器采用了第三种方式，逐渐形成了所谓的“事件驱动处理库”。

事件驱动处理库又被称为多路 IO 复用方法，最常见的包括以下三种：select 模型、poll 模型和 epoll 模型。Nginx 服务器还支持 rtsig 模型、kqueue 模型、dev/poll 模型和 eventport 模型等。通过 Nginx 配置可以使得 Nginx 服务器支持这几种事件驱动处理模型。我们在这里详细介绍一下它们，让大家对 Nginx 服务器的事件处理机制有较为清晰的了解，也为我们后文介绍 Nginx 服务器源码做铺垫。

3.3.3　select 库

select 库，是各个版本的 Linux 和 Windows 平台都支持的基本事件驱动模型库，并且在接口的定义上也基本相同，只是部分参数的含义略有差异。使用 select 库的步骤一般是：

首先，创建所关注事件的描述符集合。对于一个描述符，可以关注其上面的读（Read）事件、写（Write）事件以及异常发生（Exception）事件，所以要创建三类事件描述符集合，分别用来收集读事件的描述符、写事件的描述符和异常事件的描述符。

其次，调用底层提供的 select()函数，等待事件发生。这里需要注意的一点是，select 的阻塞与是否设置非阻塞 I/O 是没有关系的。

然后，轮询所有事件描述符集合中的每一个事件描述符，检查是否有相应的事件发生，如果有，就进行处理。

Nginx 服务器在编译过程中如果没有为其指定其他高性能事件驱动模型库，它将自动编译该库。我们可以使用--with-select_module 和--without-select_module 两个参数强制 Nginx 是否编译该库。

3.3.4　poll 库

poll 库，作为 Linux 平台上的基本事件驱动模型，是在 Linux 2.1.23 中引入的。Windows 平台不支持 poll 库。

poll 与 select 的基本工作方式是相同的，都是先创建一个关注事件的描述符集合，再去等待这些事件发生，然后再轮询描述符集合，检查有没有事件发生，如果有，就进行处理。

poll 库与 select 库的主要区别在于，select 库需要为读事件、写事件和异常事件分别创建一个描述符集合，因此在最后轮询的时候，需要分别轮询这三个集合。而 poll 库只需要创建一个集合，在每个

描述符对应的结构上分别设置读事件、写事件或者异常事件，最后轮询的时候，可以同时检查这三种事件是否发生。可以说，poll 库是 select 库的优化实现。

Nginx 服务器在编译过程中如果没有为其指定其他高性能事件驱动模型库，它将自动编译该库。我们可以使用--with-poll_module 和--without-poll_module 两个参数强制 Nginx 是否编译该库。

3.3.5 epoll 库

epoll 库是 Nginx 服务器支持的高性能事件驱动库之一，它是公认的非常优秀的事件驱动模型，和 poll 库及 select 库有很大的不同。epoll 属于 poll 库的一个变种，是在 Linux 2.5.44 中引入的，在 Linux 2.6 及以上的版本都可以使用它。poll 库和 select 库在实际工作中，最大的区别在于效率。

从前面的介绍我们知道，它们的处理方式都是创建一个待处理事件列表，然后把这个列表发给内核，返回的时候，再去轮询检查这个列表，以判断事件是否发生。这样在描述符比较多的应用中，效率就显得比较低下了。一种比较好的做法是，把描述符列表的管理交由内核负责，一旦有某种事件发生，内核把发生事件的描述符列表通知给进程，这样就避免了轮询整个描述符列表。epoll 库就是这样一种模型。

首先，epoll 库通过相关调用通知内核创建一个有 *N* 个描述符的事件列表；然后，给这些描述符设置所关注的事件，并把它添加到内核的事件列表中去，在具体的编码过程中也可以通过相关调用对事件列表中的描述符进行修改和删除。

完成设置之后，epoll 库就开始等待内核通知事件发生了。某一事件发生后，内核将发生事件的描述符列表上报给 epoll 库。得到事件列表的 epoll 库，就可以进行事件处理了。

epoll 库在 Linux 平台上是高效的。它支持一个进程打开大数目的事件描述符，上限是系统可以打开文件的最大数目；同时，epoll 库的 IO 效率不随描述符数目增加而线性下降，因为它只会对内核上报的“活跃”的描述符进行操作。

3.3.6 rtsig 模型

rtsig 是 Real-Time Signal 的缩写，是实时信号的意思。从严格意义上说，rtsig 模型并不是常用的事件驱动模型，但 Nginx 服务器提供了使用实时信号对事件进行响应的支持，官方文档中将 rtsig 模型与其他的事件驱动模型并列，因此我们也将该模型放到这一节来介绍。rtsig 模型在 Linux 2.2.19 及以上的版本中可以使用。

使用 rtsig 模型时，工作进程会通过系统内核建立一个 rtsig 队列用于存放标记事件发生（在 Nginx 服务器应用中特指客户端请求发生）的信号。每个事件发生时，系统内核就会产生一个信号存放到 rtsig 队列中等待工作进程的处理。

需要指出的是，rtsig 队列有长度限制，超过该长度后就会发生溢出。默认情况下，Linux 系统事件信号队列的最大长度设置为 1 024，也就是同时最多可以存放 1 024 个发生事件的信号。在 Linux 2.6.6-mm2 之前的版本中，系统各个进程的事件信号队列是由内核统一管理的，用户可以通过修改内核参数/proc/sys/kernel/rtsig-max 来自定义该长度设置。在 Linux 2.6.6-mm2 之后的版本中，该内核参数被取消，系统各个进程分别拥有各自的事件信号队列，这个队列的大小由 Linux 系统的 RLIMIT_SIGPENDING 参数定义，在执行 setrlimit() 系统调用时确定该大小。Nginx 提供了

worker_rlimit_sigpending 参数用于调节这种情况下的事件信号队列长度。

当 rtsig 队列发生溢出时，Nginx 将暂时停止使用 rtsig 模型，而调用 poll 库处理未处理的事件，直到 rtsig 信号队列全部清空，然后再次启动 rtsig 模型，以防止新的溢出发生。

Nginx 在配置文件中提供了相关的参数对 rtsig 模型的使用进行配置，细节内容在后边讨论 Nginx 服务器事件驱动模型时将会详细阐述。编译 Nginx 服务器时，使用-with-rtsig_module 配置选项来启用 rtsig 模型的编译。

3.3.7　其他事件驱动模型

除了以上四种主要的事件驱动模型，Nginx 服务器针对特定的 Linux 平台提供了响应的事件驱动模型支持。目前实现的主要有 kqueue 模型、/dev/poll 模型和 eventport 模型等。

- kqueue 模型，是用于支持 BSD 系列平台的高效事件驱动模型，主要用在 FreeBSD 4.1 及以上版本、OpenBSD 2.9 及以上版本、NetBSD 2.0 及以上版本以及 Mac OS X 平台上。该模型也是 poll 库的一个变种，其和 epoll 库的处理方式没有本质上的区别，都是通过避免轮询操作提供效率。该模型同时支持条件触发（level-triggered，也叫水平触发，只要满足条件就触发一个事件）和边缘触发（edge-triggered，每当状态变化时，触发一个事件）。如果大家在这些平台下使用 Nginx 服务器，建议选择该模型用于请求处理，以提高 Nginx 服务器的处理性能。
- /dev/poll 模型，是用于支持 Unix 衍生平台的高效事件驱动模型，其主要在 Solaris7 11/99 及以上版本、HP/UX 11.22 及以上版本、IRIX 6.5.15 及以上版本和 Tru64 UNIX 5.1A 及以上版本的平台中使用。该模型是 Sun 公司在开发 Solaris 系列平台时提出的用于完成事件驱动机制的方案，它使用了虚拟的/dev/poll 设备，开发人员可以将要监视的文件描述符加入这个设备，然后通过 ioctl()调用来获取事件通知。在以上提到的平台中，建议使用该模型处理请求。
- eventport 模型，是用于支持 Solaris 10 及以上版本平台的高效事件驱动模型。该模型也是 Sun 公司在开发 Solaris 系列平台时提出的用于完成事件驱动机制的方案，它可以有效防止内核崩溃等情况的发生，Nginx 服务器为此提供了支持。

以上就是 Nginx 服务器支持的事件驱动库。可以看到，Nginx 服务器针对不同的 Linux 或 Unix 衍生平台提供了多种事件驱动模型的处理，尽量发挥系统平台本身的优势，最大程度地提高处理客户端请求事件的能力。在实际工作中，我们需要根据具体情况和应用情景选择使用不同的事件驱动模型，以保证 Nginx 服务器的高效运行。

在本书后文中，我们还会多次提到上面介绍的几种事件驱动的配置。根据不同的环境需求选择不同的事件驱动方式，可以发挥 Nginx 服务器处理事件的最佳能力。

3.4　设计架构概览

Nginx 服务器灵活强大的功能扩展特性是其巨大的优势。本节我们着眼于 Nginx 服务器的架构，深入了解 Nginx 服务器的设计思想。架构是一门高深的学问，尤其对于 Nginx 这样优秀的软件，更是

需要在长期的使用中不断学习和总结。本节内容是笔者在自己理解的基础上，结合一些知名学者的观点完成的，希望广大读者在阅读和学习的过程中能够加入我们的讨论，共同探讨 Nginx 的“架构之美”。

在 3.1 节中我们介绍了 Nginx 服务器的模块化设计，对常见的标准 HTTP 模块、可选 HTTP 模块、邮件服务模块和第三方模块有了一定的认识。我们看到，从服务器的核心服务到一般应用功能，都是由这些模块支持的，这些模块在功能上彼此独立，在逻辑上又能相互影响、相互联系、共同协作，从而构成一套功能强大完整的服务器程序。那么，Nginx 服务器是如何达到这样的效果的呢？这依赖于 Nginx 服务器的设计架构。

3.4.1　Nginx 服务器架构

Nginx 服务器启动后，产生一个主进程（master process），主进程执行一系列工作后产生一个或者多个工作进程（worker processes）。主进程主要进行 Nginx 配置文件解析、数据结构初始化、模块配置和注册、信号处理、网络监听生成、工作进程生成和管理等工作；工作进程主要进行进程初始化、模块调用和请求处理等工作，是 Nginx 服务器提供服务的主体。

在客户端请求动态站点的过程中，Nginx 服务器还涉及和后端服务器的通信。Nginx 服务器将接收到的 Web 请求通过代理转发到后端服务器，由后端服务器进行数据处理和页面组织，然后将结果返回。

另外，Nginx 服务器为了提高对请求的响应效率，进一步降低网络压力，采用了缓存机制，将历史应答数据缓存到本地。在每次 Nginx 服务器启动后的一段时间内，会启动专门的进程对本地缓存的内容重建索引，保证对缓存文件的快速访问。

根据上面的分析，我们可以将 Nginx 服务器的结构大致分为主进程、工作进程、后端服务器和缓存等部分。图 3.2 展示了各个部分之间的联系和交互。

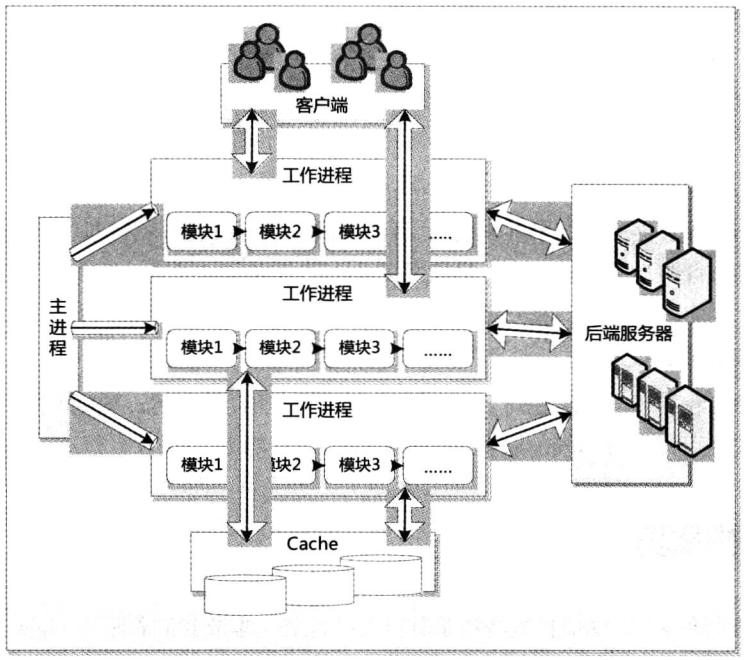

图 3.2　Nginx 服务器架构示意图

在该示意图中，有几个方面的内容我们需要重点阐述，包括 Nginx 服务器的进程、进程交互和 Run-Loop 事件处理循环机制等。

3.4.2　Nginx 服务器的进程

到目前为止，我们一共提到 Nginx 服务器的三大类进程：一类是主进程，另一类是由主进程生成的工作进程，还有刚才提到的用于为缓存文件建立索引的进程。

1. 主进程（Master Process）

Nginx 服务器启动时运行的主要进程。它的主要功能是与外界通信和对内部其他进程进行管理，具体来说有以下几点：

- 读取 Nginx 配置文件并验证其有效性和正确性。
- 建立、绑定和关闭 Socket。
- 按照配置生成、管理和结束工作进程。
- 接收外界指令，比如重启、升级及退出服务器等指令。
- 不中断服务，实现平滑重启，应用新配置。
- 不中断服务，实现平滑升级，升级失败进行回滚处理。
- 开启日志文件，获取文件描述符。
- 编译和处理 Perl 脚本。

2. 工作进程（Worker Process）

由主进程生成，生成数量可以通过 Nginx 配置文件指定，正常情况下生存于主进程的整个生命周期。该进程的主要工作有以下几项：

- 接收客户端请求。
- 将请求依次送入各个功能模块进行过滤处理。
- IO 调用，获取响应数据。
- 与后端服务器通信，接收后端服务器处理结果。
- 数据缓存，访问缓存索引、查询和调用缓存数据。
- 发送请求结果，响应客户端请求。
- 接收主程序指令，比如重启、升级和退出等指令。

工作进程完成的工作还有很多，我们在这里列出了主要的几项。从这些工作中可以看到，该进程是 Nginx 服务器提供 Web 服务、处理客户端请求的主要进程，完成了 Nginx 服务器的主体工作。因此，在实际使用中，作为服务器管理者，我们应该重点监视工作进程的运行状态，保证 Nginx 服务器对外提供稳定的 Web 服务。

3. 缓存索引重建及管理进程（Cache Loader & Cache Manager）

图 3.1 中的 Cache 模块，主要由缓存索引重建（Cache Loader）和缓存索引管理（Cache Manager）两类进程完成工作。缓存索引重建进程是在 Nginx 服务启动一段时间之后（默认是 1 分钟）由主进程生成，在缓存元数据重建完成后就自动退出；缓存索引管理进程一般存在于主进程的整个生命周期，

负责对缓存索引进行管理。

缓存索引重建进程完成的主要工作是，根据本地磁盘上的缓存文件在内存中建立索引元数据库。该进程启动后，对本地磁盘上存放缓存文件的目录结构进行扫描，检查内存中已有的缓存元数据是否正确，并更新索引元数据库。

缓存索引管理进程主要负责在索引元数据更新完成后，对元数据是否过期做出判断。

这两个进程维护的内存索引元数据库，为工作进程对缓存数据的快速查询提供了便利。

3.4.3　进程交互

Nginx 服务器在使用 Master-Worker 模型时，会涉及主进程与工作进程（Master-Worker）之间的交互和工作进程（Worker-Worker）之间的交互。这两类交互都依赖于管道（channel）机制，交互的准备工作都是在工作进程生成时完成的。

1. Master–Worker 交互

工作进程是由主进程生成的（使用了 fork 函数，具体的源码实现我们在后边的相关章节中完整解析）。Nginx 服务器启动以后，主进程根据配置文件决定生成的工作进程的数量，然后建立一张全局的工作进程表用于存放当前未退出的所有工作进程。

在主进程生成工作进程后，将新生成的工作进程加入到工作进程表中，并建立一个单向管道并将其传递给该工作进程。该管道与普通的管道不同，它是由主进程指向工作进程的单向管道，包含了主进程向工作进程发出的指令、工作进程 ID、工作进程在工作进程表中的索引和必要的文件描述符等信息。

主进程与外界通过信号机制进行通信，当接收到需要处理的信号时，它通过管道向相关的工作进程发送正确的指令。每个工作进程都有能力捕获管道中可读事件，当管道中有可读事件时，工作进程从管道读取并解析指令，然后采取相应的措施。这样就完成了 Master-Worker 的交互。

2. Worker–Worker 交互

Worker-Worker 交互在实现原理上和 Master-Worker 交互基本是一样的。只要工作进程之间能够得到彼此的信息，建立管道，即可通信。由于工作进程之间是相互隔离的，因此一个进程要想知道另一个进程的信息，只能通过主进程来设置了。

为了达到工作进程之间交互的目的，主进程在生成工作进程后，在工作进程表中进行遍历，将该新进程的 ID 以及针对该进程建立的管道句柄传递给工作进程表中的其他进程，为工作进程之间的交互做准备。每个工作进程捕获管道中可读事件，根据指令采取响应的措施。

当工作进程 W1 需要向 W2 发送指令时，首先在主进程给它的其他工作进程信息中找到 W2 的进程 ID，然后将正确的指令写入指向 W2 的通道。工作进程 W2 捕获到管道中的事件后，解析指令并采取相应措施。这样就完成了 Worker-Worker 交互。

3.4.4　Run Loops 事件处理循环模型

Run Loops，指的是进程内部用来不停地调配工作，对事件进行循环处理的一种模型。它属于进程或者线程的基础架构部分。该模型对事件的处理不是自动的，需要在设计代码过程中，在适当的时候

启动 Run-Loop 机制对输入的事件作出响应。

该模型是一个集合，集合中的每一个元素称为一个 Run-Loop。每个 Run-Loop 可运行在不同的模式下，其中可以包含它所监听的输入事件源、定时器以及在事件发生时需要通知的 Run-Loop 监听器（Run-Loop Observers）。为了监听特定的事件，可以在 Run Loops 中添加相应的 Run-Loop 监听器。当被监听的事件发生时，Run-Loop 会产生一个消息，被 Run-Loop 监听器捕获，从而执行预定的动作。

Nginx 服务器在工作进程中实现了 Run-Loop 事件处理循环模型的使用，用来处理客户端发来的请求事件。该部分的实现可以说是 Nginx 服务器程序实现中最为复杂的部分，包含了对输入事件繁杂的响应和处理过程，并且这些处理过程都是基于异步任务处理的。

我们不打算在这一章深入介绍 Nginx 服务器的 Run-Loop 模型。这部分的实现过程结合相关代码实现会更容易理解，因此我们在探讨 Nginx 服务器源码实现时再对它进行详细介绍，这里大家知道 Run-Loop 模型的作用就可以了。

通过学习 Nginx 服务器的整体架构，我们对 Nginx 服务器各个模块的作用和联系有了比较清晰的认识。可以看到，Nginx 服务器提供了异步的、非阻塞的 Web 服务，系统中的模块各司其职，彼此之间通常使用网络、管道和信号等机制进行通信，从而保持了松耦合的关系。工作进程中事件处理机制的使用，在很大程度上降低了在网络负载繁重的情况下 Nginx 服务器对内存、磁盘的压力，同时保证了对客户端请求的及时响应。

这里需要提及的一点是，笔者在实际使用 Nginx 服务器的过程中发现，当磁盘没有足够的性能处理大量 IO 调用时，工作进程仍然可能因为磁盘读写调用而阻塞，进而导致客户端请求超时失败等问题。目前可以通过多种方法来降低对磁盘 IO 的调用，比如引入异步输入/输出（Asynchronous Input/Output，AIO）机制等，但这些处理办法没有从根本上解决问题。为了尽量避免产生这种问题，大家在实际部署 Nginx 服务器时，应当对其运行环境有一个基本的了解，针对不同的网络负载环境选择相匹配的硬件环境，并对 Nginx 服务器进行合理的配置。

3.5　本章小结

在本章中我们对 Nginx 服务器的整体架构和各个主要架构部件以及各部件之间的关系和交互进行了阐述。我们首先了解了 Nginx 服务器的模块化结构思想，并将 Nginx 服务器的模块大致分为核心模块、标准 HTTP 模块、可选 HTTP 模块、邮件服务模块以及第三方模块等五大类。之后对 Nginx 服务器的 Web 服务提供机制进行了详细的讨论，其中需要重点关注的是 Nginx 服务器采用的事件处理机制，这在下一章中我们还会提到。最后，我们重点学习了 Nginx 服务器的整体设计架构，对 Nginx 服务器各架构部件的功能、之间的联系进行了详细说明。

相信通过本章的学习，大家对 Nginx 服务器的整体框架和流程有了一定的认识。在接下来的一章中，我们回到 Nginx 服务器应用这个中心点上，介绍对 Nginx 服务器的应用优化。学习了本章内容，大家对下一章中要讨论的内容就更容易理解。

同时，本书的第一篇到此也结束了。在第一篇中，我们着眼于 Nginx 服务器的基础，对它的历史背景、发展现状进行简要回顾后，通过大量应用实例学习了如何编译、配置、运行 Nginx 服务器，通

过这些学习就可以轻松部署一套可以提供基本 Web 服务的 Nginx 服务器。紧接着我们学习了 Nginx 服务器的整体设计架构，重点对模块化思想和 Web 请求处理机制进行了阐述，探讨了它如何提供稳定的 Web 服务和良好的功能扩展，使我们对 Nginx 服务器的整体运行机制有了一定的了解。

从下一章开始，我们进入 Nginx 服务器学习的提高阶段，仍然围绕 Nginx 服务器的应用这一中心，更深入一步地探讨 Nginx 服务器的高级应用和理论知识。

第 4 章

Nginx 服务器的高级配置

Nginx 服务器的配置远不仅是我们在之前提到的那些内容。Nginx 服务器运行的实际环境和提供的功能千差万别，如何根据自己的工作经验和具体的使用环境为 Nginx 服务器配置恰当的指令，才是 Nginx 配置的核心内容。本章根据 Nginx 的官方文档和笔者的实际工作经验，为大家展示了一些与实际运行环境相关联的 Nginx 服务器高级配置，希望大家能够熟练掌握。这些配置和服务器运行的操作系统及硬件环境有关，大家在实际操作过程中请自行做出调整。

我们将在本章学习到以下知识：

- 针对 IPv4 的内核参数优化
- 针对处理器的指令配置
- 针对网络连接的指令配置
- 与事件驱动相关的指令配置

4.1 针对 IPv4 的内核 7 个参数的配置优化

这里提及的参数是和 IPv4 网络有关的 Linux 内核参数。我们可以将这些内核参数的值追加到 Linux 系统的/etc/sysctl.conf 文件中，然后使用如下命令使修改生效：

```
#/sbin/sysctl -p
```

这些常用的参数包括以下这些。

1. net.core.netdev_max_backlog 参数

参数 net.core.netdev_max_backlog，表示当每个网络接口接收数据包的速率比内核处理这些包的速率快时，允许发送到队列的数据包的最大数目。一般默认值为 128（可能不同的 Linux 系统该数值也

不同）。Nginx 服务器中定义的 NGX_LISTEN_BACKLOG 默认为 511。我们可以将它调整一下：

```
net.core.netdev_max_backlog = 262144
```

2. net.core.somaxconn 参数

该参数用于调节系统同时发起的 TCP 连接数，一般默认值为 128。在客户端存在高并发请求的情况下，该默认值较小，可能导致链接超时或者重传问题，我们可以根据实际需要结合并发请求数来调节此值。笔者系统设置为：

```
net.core.somaxconn = 262144
```

3. net.ipv4.tcp_max_orphans 参数

该参数用于设定系统中最多允许存在多少 TCP 套接字不被关联到任何一个用户文件句柄上。如果超过这个数字，没有与用户文件句柄关联的 TCP 套件字将立即被复位，同时给出警告信息。这个限制只是为了防止简单的 DoS（Denial of Service，拒绝服务）攻击。一般在系统内存比较充足的情况下，可以增大这个参数的赋值：

```
net.ipv4.tcp_max_orphans = 262144
```

4. net.ipv4.tcp_max_syn_backlog 参数

该参数用于记录尚未收到客户端确认信息的连接请求的最大值。对于拥有 128 MB 内存的系统而言，此参数的默认值是 1 024，对小内存的系统则是 128。一般在系统内存比较充足的情况下，可以增大这个参数的赋值：

```
net.ipv4.tcp_max_syn_backlog = 262144
```

5. net.ipv4.tcp_timestamps 参数

该参数用于设置时间戳，这可以避免序列号的卷绕。在一个 1Gb/s 的链路上，遇到以前用过的序列号的概率很大。当此值赋值为 0 时，禁用对于 TCP 时间戳的支持。在默认情况下，TCP 协议会让内核接受这种“异常”的数据包。针对 Nginx 服务器来说，建议将其关闭：

```
net.ipv4.tcp_timestamps = 0
```

6. net.ipv4.tcp_synack_retries 参数

该参数用于设置内核放弃 TCP 连接之前向客户端发送 SYN+ACK 包的数量。为了建立对端的连接服务，服务器和客户端需要进行三次握手，第二次握手期间，内核需要发送 SYN 并附带一个回应前一个 SYN 的 ACK，这个参数主要影响这个过程，一般赋值为 1，即内核放弃连接之前发送一次 SYN+ACK 包，可以设置其为：

```
net.ipv4.tcp_synack_retries = 1
```

7. net.ipv4.tcp_syn_retries 参数

该参数的作用与上一个参数类似，设置内核放弃建立连接之前发送 SYN 包的数量，它的赋值和上个参数一样即可：

```
net.ipv4.tcp_syn_retries = 1
```

4.2 针对 CPU 的 Nginx 配置优化的 2 个指令

处理器已经进入多核时代。多内核是指在一枚处理器中集成两个或多个完整的计算引擎，多核处

理器是单枚芯片。一枚多核处理器上可以承载多枚内核，但只需要单一的处理器插槽就可以工作。同时，目前流行的操作系统都已经可以利用这样的资源，将每个执行内核作为分立的逻辑处理器，通过在多个执行内核之间划分任务，在特定的时钟周期内执行更多任务，提高并行处理任务的能力。

在 Nginx 配置文件中，有这样两个指令：worker_processes 和 worker_cpu_affinity，它们可以针对多核 CPU 进行配置优化。

1.worker_processes 指令

worker_processes 指令用来设置 Nginx 服务的进程数。官方文档建议此指令一般设置为 1 即可，赋值太多会影响系统的 IO 效率，降低 Nginx 服务器的性能。根据笔者的经验，为了让多核 CPU 能够很好地并行处理任务，我们可以将 worker_processes 指令的赋值适当地增大一些，最好是赋值为机器 CPU 的倍数。当然，这个值并不是越大越好，Nginx 进程太多可能增加主进程调度负担，也可能影响系统的 IO 效率。针对双核 CPU，建议设置为 2 或 4。笔者的机器为四核 CPU，设置为：

```
worker_processes  4;
```

设置好 worker_processes 指令之后，就很有必要设置 worker_cpu_affinity 指令。

2. worker_cpu_affinity 指令

worker_cpu_affinity 指令用来为每个进程分配 CPU 的工作内核。这个指令的设置方法有些麻烦。我们先来看一张笔者根据官方文档画的示意图。如图 4.1 所示，worker_cpu_affinity 指令的值是由几组二进制值表示的。其中，每一组代表一个进程，每组中的每一位表示该进程使用 CPU 的情况，1 代表使用，0 代表不使用。注意，二进制位排列顺序和 CPU 的顺序是相反的。建议将不同的进程平均分配到不同的 CPU 运行内核上。

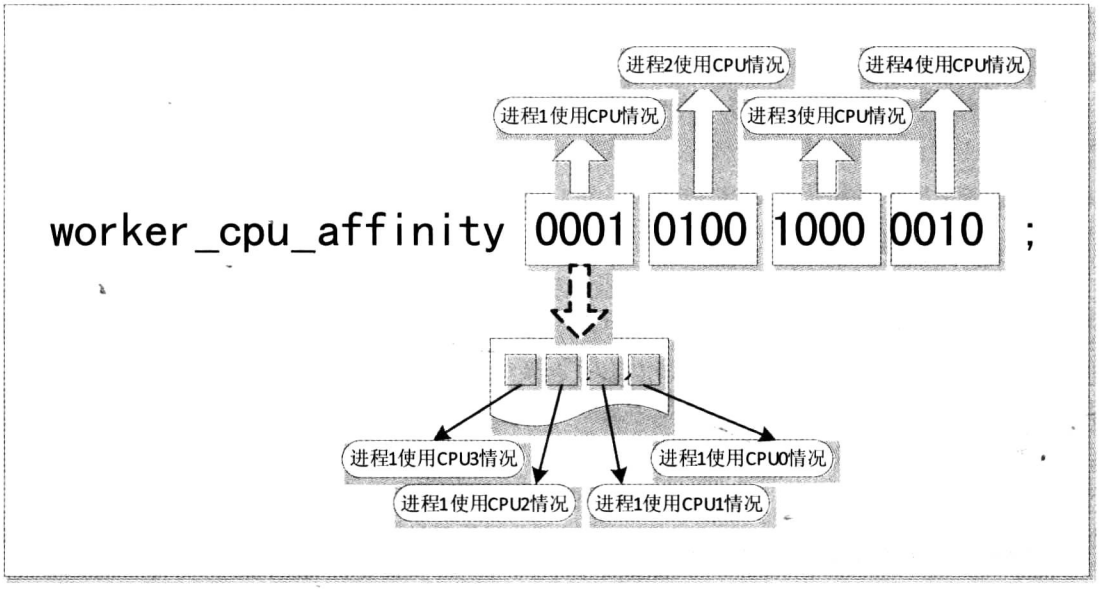

图 4.1　worker_cpu_affinity 指令示意图

笔者设置 Nginx 服务的进程数为 4，CPU 是四核，因此会有四组值，并且每组有四位，所以，此指令的设置为：

```
worker_cpu_affinity 0001 0100 1000 0010;
```

四组二进制数值分别对应 4 个进程，第一个进程对应 0001，表示使用第一个 CPU 内核；第二个进程对应 0010，表示使用第二个 CPU 内核，以此类推。

如果笔者将 worker_processes 指令的值赋值为 8，即赋值为 CPU 内核个数的两倍，则 worker_cpu_affinity 指令的设置可以是：

```
worker_cpu_affinity 0001 0010 0100 1000 0001 0010 0100 1000;
```

如果一台机器的 CPU 是八核 CPU，并且 worker_processes 指令的值赋值为 8，那么 worker_cpu_affinity 指令的设置可以是：

```
worker_cpu_affinity 00000001 00000010 00000100 00001000 00010000 00100000 01000000 10000000;
```

以上两例的具体含义不再赘述，相信大家通过前面的介绍能够明白。

4.3 与网络连接相关的配置的 4 个指令

1. keepalive_timeout 指令

该指令用于设置 Nginx 服务器与客户端保持连接的超时时间。

这个指令支持两个选项，中间用空格隔开。第一个选项指定客户端连接保持活动的超时时间，在这个时间之后，服务器会关闭此连接；第二个选项可选，其指定了使用 Keep-Alive 消息头保持活动的有效时间，如果不设置它，Nginx 服务器不会向客户端发送 Keep-Alive 消息头以保持与客户端某些浏览器（如 Mozilla、Konqueror 等）的连接，超过设置的时间后，客户端就可以关闭连接，而不需要服务器关闭了。你可以根据自己的实际情况设置此值，建议从服务器的访问数量、处理速度以及网络状况方面考虑。下面是此指令的设置示例：

```
keepalive_timeout 60 50;
```

该设置表示 Nginx 服务器与客户端连接保持活动的时间是 60 s，60 s 后服务器与客户端断开连接；使用 Keep-Alive 消息头保持与客户端某些浏览器（如 Mozilla、Konqueror 等）的连接时间为 50 s，50 s 后浏览器主动与服务器断开连接。

2.send_timeout 指令

该指令用于设置 Nginx 服务器响应客户端的超时时间，这个超时时间仅针对两个客户端和服务器之间建立连接后，某次活动之间的时间。如果这个时间后客户端没有任何活动，Nginx 服务器将会关闭连接。此指令的设置需要考虑服务器访问数量和网络状况等方面。下面是此指令的设置示例：

```
send_timeout 10s;
```

该设置表示 Nginx 服务器与客户端建立连接后，某次会话中服务器等待客户端响应超过 10 s，就会自动关闭连接。

3. client_header_buffer_size 指令

该指令用于设置 Nginx 服务器允许的客户端请求头部的缓冲区大小，默认为 1KB。此指令的赋值可以根据系统分页大小来设置。分页大小可以用以下命令取得：

```
#getconf PAGESIZE
```

有过 Nginx 服务器工作经验的朋友可能遇到过 Nginx 服务器返回 400 错误的情况。查找 Nginx 服务器的 400 错误原因比较困难，因为此错误并不是每次都会出现，出现错误的时候，通常在浏览器和

日志里也看不到任何有关提示信息。根据实际的经验来看，有很大一部分情况是客户端的请求头部过大造成的。请求头部过大，通常是客户端 cookie 中写入了较大的值引起的。于是适当增大此指令的赋值，允许 Nginx 服务器接收较大的请求头部，可以改善服务器对客户端的支持能力。笔者一般将此指令赋值为 4KB 大小，即：

```
client_header_buffer_size 4k;
```

4. multi_accept 指令

该指令用于配置 Nginx 服务器是否尽可能多地接收客户端的网络连接请求，默认值为 off。

4.4　与事件驱动模型相关的配置的 8 个指令

本节涉及的指令与第 3 章介绍的 Nginx 服务器的事件驱动模型密切相关，建议大家在充分学习 Nginx 服务器的事件驱动模型知识之后再学习本节的内容。

1. use 指令

use 指令用于指定 Nginx 服务器使用的事件驱动模型。

2. worker_connections 指令

该指令用于设置 Nginx 服务器的每个工作进程允许同时连接客户端的最大数量，语法为：

```
worker_connections number
```

其中，number 为设置的最大数量。结合 worker_processes 指令，我们可以计算出 Nginx 服务器允许同时连接的客户端最大数量 Client = worker_processes * worker_connections / 2。

在使用 Nginx 服务器的过程中，笔者曾经遇到过无法访问 Nginx 服务器的情况，查看日志发现一直在报如下错误：

```
[alert] 24082#0: 1024 worker_connections is not enough while accepting new connection
on 0.0.0.0:81
```

根据报错信息，推测可能是 Nginx 服务器的最大访问连接数设置小了。此指令设置的就是 Nginx 服务器能接受的最大访问量，其中包括前端用户连接也包括其他连接，这个值在理论上等于此指令的值与它允许开启的工作进程最大数的乘积。此指令一般设置为 65535：

```
worker_connections 65535;
```

此指令的赋值与 linux 操作系统中进程可以打开的文件句柄数量有关系。笔者曾经遇到过这样的情况，按照以上设置修改了此项赋值以后，Nginx 服务器报如下错误：

```
[warn]: 8192 worker_connections are more than open file resource limit: 1024 '
```

究其原因， Linux 系统中有一个系统指令 open file resource limit，它设置了进程可以打开的文件句柄数量。worker_connections 指令的赋值当然不能超过 open file resource limit 的赋值。可以使用以下命令查看在你的 Linux 系统中 open file resource limit 指令的值：

```
# cat /proc/sys/fs/file-max
```

可以通过以下命令将 open file resource limit 指令的值设为 2390251：

```
#echo "2390251" > /proc/sys/fs/file-max; sysctl -p
```

这样，Nginx 的 worker_connections 指令赋值为 65535 就没问题了。

3. worker_rlimit_sigpending 指令

该指令用于设置 Linux 2.6.6-mm2 版本之后 Linux 平台的事件信号队列长度上限。其语法结构为：

```
worker_rlimit_sigpending limit
```

其中，*limit* 为 Linux 平台事件信号队列的长度上限值。

该指令主要影响事件驱动模型中 rtsig 模型可以保存的最大信号数。Nginx 服务器的每一个工作进程有自己的事件信号队列用于暂存客户端请求发生信号，如果超过长度上限，Nginx 服务器自动转用 poll 模型处理未处理的客户端请求。为了保证 Nginx 服务器对客户端请求的高效处理，请大家根据实际的客户端并发请求数量和服务器运行环境的处理能力设定该值。设置示例为：

```
worker_rlimit_sigpending 1024;
```

4. devpoll_changes 和 devpoll_events 指令

这两个指令用于设置在/dev/poll 事件驱动模式下 Nginx 服务器可以与内核之间传递事件的数量，前者设置传递给内核的事件数量，后者设置从内核获取的事件数量，语法结构为：

```
devpoll_changes number
devpoll_events number
```

其中，*number* 为要设置的数量，默认值均为 32。

5. kqueue_changes 和 kqueue_events 指令

这两个指令用于设置在 kqueue 事件驱动模式下 Nginx 服务器可以与内核之间传递事件的数量，前者设置传递给内核的事件数量，后者设置从内核获取的事件数量，其语法结构为：

```
kqueue_changes number
kqueue_events number
```

其中，*number* 为要设置的数量，默认值均为 512。

使用 kequeue_changes 方式，可以设置与内核之间传递事件的数量。

6. epoll_events 指令

该指令用于设置在 epoll 事件驱动模式下 Nginx 服务器可以与内核之间传递事件的数量，其语法结构为：

```
epoll _changes number
```

其中，*number* 为要设置的数量，默认值均为 512。

注意

与其他事件驱动模型不同，在 epoll 事件驱动模式下 Nginx 服务器向内核传递事件的数量和从内核传递的事件数量是相等的，因此没有类似 epoll_changes 这样的指令。

7. rtsig_signo 指令

该指令用于设置 rtsig 模式使用的两个信号中的第一个，第二个信号是在第一个信号的编号上加 1，语法为：

```
rtsig_signo signo
```

默认的第一个信号设置为 SIGRTMIN+10。

提示

在 Linux 中可以使用以下命令查看系统支持的 SIGRTMIN 有哪些。笔者的系统运行结果如下所示：

```
# kill -l | grep SIGRTMIN
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
```

8. rtsig_overflow_* 指令

该指令代表三个具体的指令，分别为 rtsig_overflow_events 指令、rtsig_overflow_test 指令和 rtsig_overflow_threshold 指令。这些指令用来控制当 rtsig 模式中信号队列溢出时 Nginx 服务器的处理方式，在前面章节我们已经介绍过，这里不再赘述。它们的语法结构为：

```
rtsig_overflow_* number
```

其中，*number* 是要设定的值。

rtsig_overflow_events 指令指定队列溢出时使用 poll 库处理的事件数，默认值为 16。

rtsig_overflow_test 指令指定 poll 库处理完第几件事件后将清空 rtsig 模型使用的信号队列，默认值为 32。

rtsig_overflow_threshold 指令指定 rtsig 模式使用的信号队列中的事件超过多少时就需要清空队列了。该指令只对 Linux 2.4.x 及以下版本有效。在这些版本中包含两个参数：分别是 proc/sys/kernel/ rtsig-nr 和/proc/sys/kernel/rtsig-max/rtsig_overflow_threshold，后者就是该指令设定的值。当 Nginx 服务器检测到前者大于后者时，将清空队列。该指令默认值为 10，代表 rtsig-max 的 1/10。

4.5　本章小结

本章内容不多，主要是对第 2 章的补充和延伸。我们在第 3 章中学习了 Nginx 服务器架构相关的基础知识，尤其对 Nginx 服务器在处理 Web 请求、事件驱动方面的内容有了了解。本章基于这些知识，为大家展示了 Nginx 服务器与运行软硬件环境相关的高级配置。这些配置对于 Nginx 服务器高效和充分利用系统资源是非常有帮助的，当然也需要用户能够充分掌握服务器的运行环境。

同时，我们也体会到 Nginx 服务器为用户提供的配置接口是相当丰富和全面的，这增强了自身系统的灵活性和可定制能力，以及对实际生产环境的适应能力。这也是 Nginx 服务器能够受到广大用户喜欢的另一个重要原因。

第 5 章

Nginx 服务器的 Gzip 压缩

在 Nginx 配置文件中可以配置 Gzip 的使用，相关指令可以在配置文件的 http 块、server 块或者 location 块中设置，Nginx 服务器通过 ngx_http_gzip_module 模块、ngx_http_gzip_static_module 模块和 ngx_http_gunzip_module 模块对这些指令进行解析和处理。

本章我们要学习到的主要内容有：

- Gzip 各模块支持的配置指令
- Gzip 压缩功能的使用
- Gzip 压缩功能常见问题的解决

5.1 由 ngx_http_gzip_module 模块处理的 9 个指令

ngx_http_gzip_module 模块主要负责 Gzip 功能的开启和设置，对响应数据进行在线实时压缩。该模块包含以下主要指令。

1. gzip 指令

该指令用于开启或者关闭 Gzip 功能，语法结构为：

```
gzip on | off;
```

默认情况下，该指令设置为 off，即不启用 Gzip 功能。只有将该指令设置为 on 时，下列各指令设置才有效。

2. gzip_buffers 指令

该指令用于设置 Gzip 压缩文件使用缓存空间的大小，语法结构为：

```
gzip_buffers number size;
```

- *number*，指定 Nginx 服务器需要向系统申请缓存空间的个数。
- *size*，指定每个缓存空间的大小。

根据该配置项，Nginx 服务器在对响应输出数据进行 Gzip 压缩时需向系统申请 *number* * *size* 大小的空间用于存储压缩数据。从 Nginx 0.7.28 开始，默认情况下 *number* * *size* 的值为 128，其中 *size* 的值取系统内存页一页的大小，为 4KB 或者 8KB，即：

```
gzip_buffers 32 4k | 16 8k;
```

3. gzip_comp_level 指令

该指令用于设定 Gzip 压缩程度，包括级别 1 到级别 9。级别 1 表示压缩程度最低，压缩效率最高；级别 9 表示压缩程度最高，压缩效率最低，最费时间。其语法结构为：

```
gzip_comp_level level;
```

默认值设置为级别 1。

4. gzip_disable 指令

针对不同种类客户端发起的请求，可以选择性地开启和关闭 Gzip 功能。该指令从 Nginx 0.6.23 启用，用于设置一些客户端种类。Nginx 服务器在响应这些种类的客户端请求时，不使用 Gzip 功能缓存响应输出数据。其语法结构为：

```
gzip_disable regex ...;
```

其中，*regex* 根据客户端的浏览器标志（User-Agent，UA）进行设置，支持使用正则表达式。笔者总结了常见的 PC 浏览器及手机浏览器的 UA 字符串，具体参见表 4.1。

表 4.1　常见的浏览器标志（UA）举例

浏览器	UA 字符串
Internet Explorer 系列浏览器	
Internet Explorer10	Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; Trident/6.0)
Internet Explorer9	Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0)
Internet Explorer8	Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0)
Internet Explorer7	Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0;)
Internet Explorer6	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
火狐（FireFox）系列浏览器	
Firefox7（Windows 平台）	Mozilla/5.0 (Windows NT 6.1; Intel Mac OS X 10.6; rv7.0.1) Gecko/20100101 Firefox/7.0.1
Firefox7（Mac 平台）	Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv7.0.1) Gecko/20100101 Firefox/7.0.1
Firefox4（Windows 平台）	Mozilla/5.0 (Windows NT 6.1; rv2.0.1) Gecko/20100101 Firefox/4.0.1
Firefox4（Mac 平台）	Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv2.0.1) Gecko/20100101 Firefox/4.0.1
谷歌（Chrome）浏览器	
Chrome 10 及之前版本（Windows 平台）	Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/=*version* (KHTML, like Gecko) Chrome/*version* Safari/534.16

续表

浏览器	UA 字符串
Chrome 10 及之前版本（Linux 平台）	Mozilla/5.0 (X11; U; Linux x86_64; en-US) AppleWebKit/*version* (KHTML, like Gecko) Chrome/*version* Safari/534.16
Chrome 11（Windows 平台）	Mozilla/5.0 (Windows NT 6.0; WOW64) AppleWebKit/534.24 (KHTML, like Gecko) Chrome/*version* Safari/534.24
Chrome 11（Linux 平台）	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/534.24 (KHTML, like Gecko) Chrome/*version* Safari/534.24
Opera 浏览器	
Opera 7（Windows 平台）	Opera/7.54 (Windows NT 5.1; U) [en]
Opera 8（Windows 平台）	Opera/8.0 (Windows NT 5.1; U; en)
Opera 9（Windows 平台）	Mozilla/5.0 (Windows NT 5.1; U; en; rv:1.8.1) Gecko/20061208 Firefox/2.0.0 Opera 9.50Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; en)
Opera 10（Windows 平台）	Opera/9.80 (Macintosh; Intel Mac OS X; U; en) Presto/2.2.15 Version/10.00
iOS 系列平台浏览器	
iOS 6（iPhone）	Mozilla/5.0 (iPhone; CPU iPhone OS 6_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/6.0 Mobile/10A403 Safari/8536.25
iOS 6（iPad）	Mozilla/5.0 (iPad; CPU iPhone OS 6_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/6.0 Mobile/10A403 Safari/8536.25
iOS 5（iPhone）	Mozilla/5.0 (iPhone; CPU iPhone OS 5_0 like Mac OS X) AppleWebkit/534.46 (KHTML like Gecko) Version/5.1 Mobile/9A334 Safari/7534.48.3
iOS 5（iPad）	Mozilla/5.0 (iPad; CPU OS 5_0 like Mac OS X) AppleWebkit/534.46 (KHTML like Gecko) Version/5.1 Mobile/9A334 Safari/7534.48.3
iOS 4（iPhone）	Mozilla/5.0 (iPhone; CPU iPhone OS 4_3_2 like Mac OS X; en-us) AppleWebkit/533.17.9 (KHTML like Gecko) Version/5.0.2 Mobile/8H7 Safari/6533.18.5
iOS 4（iPad）	Mozilla/5.0 (iPad; CPU OS 4_3_2 like Mac OS X; en-us) AppleWebkit/533.17.9 (KHTML like Gecko) Version/5.0.2Mobile/8H7 Safari/6533.18.5
Android 平台浏览器	
Android 4.1.2	Mozilla/5.0 (Linux; U; Android 4.1.2; zh-cn; Nexus S Build/JZO54K) AppleWebKit/534.30 (KHTML, like Gecko) Version/4.0 Mobile Safari/534.30
Android 4.0.2	Mozilla/5.0 (Linux; U; Android 4.0.2; en-us; Galaxy Nexus Build/ICL53F) AppleWebKit/534.30 (KHTML, like Gecko) Version/4.0 Mobile Safari/534.30
Android 2.3.6	Mozilla/5.0 (Linux; U; Android 2.3.2; en-us; Nexus S Build/GRK39F) AppleWebKit/533.1 (KHTML, like Gecko) Version/4.0 Mobile Safari/533.1

注意

由于浏览器的 UA 可以人为更改，因此对于实际情况还要进行具体分析。表格中列举的常见浏览器及 UA 字符串只符合默认情况。

下面为 gzip_disable 指令的一个设置示例：

```
gzip_disable MSIE [4-6]\.
```

该设置使用了正则表达式，其可以匹配 UC 字符串中包含 MSIE 4、MSIE 5 和 MSIE6 的所有浏览器。响应这些浏览器发出的请求时，Nginx 服务器不进行 Gzip 压缩。

5. gzip_http_version 指令

早期的一些浏览器或者 HTTP 客户端，可能不支持 Gzip 自解压，因此用户有时会看到乱码，所以针对不同的 HTTP 协议版本，需要选择性地开启或者关闭 Gzip 功能。该指令用于设置开启 Gzip 功能的最低 HTTP 协议版本。其语法结构为：

```
gzip_http_version 1.0|1.1;
```

默认设置为 1.1 版本，即只有客户端使用 1.1 及以上版本的 HTTP 协议时，才使用 Gzip 功能对响应输出数据进行压缩。从目前来看，绝大多数的浏览器都支持 Gzip 自解压，一般采用默认值即可。

6. gzip_min_length 指令

Gzip 压缩功能对大数据的压缩效果明显，但是如果压缩很小的数据，可能出现越压缩数据量越大的情况（许多压缩算法都有这样的情况发生），因此应该根据响应页面的大小，选择性地开启或者关闭 Gzip 功能。该指令设置页面的字节数，当响应页面的大小大于该值时，才启用 Gzip 功能。响应页面的大小通过 HTTP 响应头部中的 Content-Length 指令获取，但是如果使用了 Chunk 编码动态压缩，Content-Length 或不存在或被忽略，该指令不起作用。其语法结构为：

```
gzip_min_length length;
```

默认设置为 20，设置为 0 时表示不管响应页面大小如何统统压缩。笔者建议将该值设置为 1KB 或以上，以防止出现数据越压越大的情况：

```
gzip_min_length 1024;
```

7. gzip_proxied 指令

该指令在使用 Nginx 服务器的反向代理功能时有效，前提是在后端服务器返回的响应页头部中，Requests 部分包含用于通知代理服务器的 Via 头域。关于 Nginx 服务器的反向代理功能将在本书后面的专门章节介绍。它主要用于设置 Nginx 服务器是否对后端服务器返回的结果进行 Gzip 压缩。该指令的语法结构为：

```
gzip_proxied off | expired | no-cache | no-store | private | no_last_modified | no_etag
| auth | any ...;
```

- *off*，关闭 Nginx 服务器对后端服务器返回结果的 Gzip 压缩，这是默认设置。
- *expired*，当后端服务器响应页头部包含用于指示响应数据过期时间的 expired 头域时，启用对响应数据的 Gzip 压缩。
- *no-cache*，当后端服务器响应页头部包含用于通知所有缓存机制是否缓存的 Cache-Control 头域、且其指令值为 no-cache 时，启用对响应数据的 Gzip 压缩。
- *no-store*，当后端服务器响应页头部包含用于通知所有缓存机制是否缓存的 Cache-Control 头域、且其指令值为 no-store 时，启用对响应数据的 Gzip 压缩。
- *private*，当后端服务器响应页头部包含用于通知所有缓存机制是否缓存的 Cache-Control 头域、且其指令值为 private 时，启用对响应数据的 Gzip 压缩。
- *no_last_modified*，当后端服务器响应页头部不包含用于指明需要获取数据最后修改时间的 Last-Modified 头域时，启用对响应数据的 Gzip 压缩。

- *no_etag*，当后端服务器响应页头部不包含用于标示被请求变量的实体值的 ETag 头域时，启用对响应数据的 Gzip 压缩。
- *auth*，当后端服务器响应页头部包含用于标示 HTTP 授权证书的 Authorization 头域时，启用对响应数据的 Gzip 压缩。
- *any*，无条件启用对后端服务器响应数据的 Gzip 压缩。

该指令的设置需要对 HTTP 协议的 HTTP Header 消息有基本的了解，关于 HTTP Header 消息头域的概念及相关的指令不在本书的讨论范围内，这里不再多述。笔者给大家推荐一个网址，其中对 HTTP Header 的各个头域进行了详细的说明：http://en.wikipedia.org/wiki/HTTP_headers。

8. gzip_types 指令

Nginx 服务器可以根据响应页的 MIME 类型选择性地开启 Gzip 压缩功能。该指令用来设置 MIME 类型，被设置的类型将被压缩。其语法结构为：

```
gzip_types mime-type ...;
```

其中，mime-type 变量的取值默认为 text/html，但实际上，在 gzip 指令设置为 on 时，Nginx 服务器会对所有的 text/html 类型页面数据进行 Gzip 压缩。该变量还可以取“*”，表示对所有 MIME 类型的页面数据进行 Gzip 压缩。一般情况下我们压缩常规的文件类型时，可以设置为：

```
gzip_types text/plain application/x-javascript text/css text/html application/xml;
```

9. gzip_vary 指令

该指令用于设置在使用 Gzip 功能时是否发送带有“Vary: Accept-Encoding”头域的响应头部。该头域的主要功能是告诉接收方发送的数据经过了压缩处理。开启后的效果是在响应头部添加了 Accept-Encoding: gzip，这对于本身不支持 Gzip 压缩的客户端浏览器是有用的。其语法结构为：

```
gzip_vary on | off;
```

默认设置为 off。事实上，我们可以通过 Nginx 配置的 add_header 指令强制 Nginx 服务器在响应头部添加“Vary: Accept-Encoding”头域，以达到相同的效果：

```
add_header Vary Accept-Encoding gzip;
```

注意

该指令在使用过程中存在 bug，会导致 IE 4 及以上浏览器的数据缓存功能失效。

5.2 由 ngx_http_gzip_static_module 模块处理的指令

ngx_http_gzip_static_module 模块主要负责搜索和发送经过 Gzip 功能预压缩的数据。这些数据以“.gz”作为后缀名存储在服务器上。如果客户端请求的数据在之前被压缩过，并且客户端浏览器支持 Gzip 压缩，就直接返回压缩后的数据。

该模块与 ngx_http_gzip_module 模块的不同之处主要在于，该模块使用的是静态压缩，在 HTTP 响应头部包含 Content-Length 头域来指明报文体的长度，用于服务器可确定响应数据长度的情况；而后者默认使用 Chunked 编码的动态压缩，其主要适用于服务器无法确定响应数据长度的情况，比如大文件下载的情形，这时需要实时生成数据长度。

与该模块有关的指令主要有以下几个：gzip_static、gzip_http_version、gzip_proxied、gzip_disable 和 gzip_vary 等。

其中的 gzip_static 指令，用于开启和关闭该模块的功能，其语法结构为：

```
gzip_static on | off | always;
```

- *on*，开启该模块的功能。
- *off*，关闭该模块的功能。
- *always*，一直发送 Gzip 预压缩文件，而不检查客户端浏览器是否支持 Gzip 压缩。

其他指令与 ngx_http_gzip_module 模块下的使用方式相同，请参阅上一节的内容。需要注意的是，gzip_proxied 指令只接收以下设置：

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

另外，对于该模块下的 gzip_vary 指令，开启以后只给未压缩的内容添加“Vary: Accept-Encoding”头域，而不是对所有内容都添加。如果需要给所有的响应头添加该头域，可以通过 Nginx 配置的 add_header 指令实现。

注意

该模块是 Nginx 服务器的可选 HTTP 模块，如果要使用，必须在 Nginx 程序配置时添加--with-http_gzip_static_module 指令。

5.3　由 ngx_http_gunzip_module 模块处理的 2 个指令

Nginx 服务器支持对响应输出数据流进行 Gzip 压缩，这对客户端浏览器来说，需要有能力解压和处理 Gzip 压缩数据，但如果客户端本身不支持该功能，就需要 Nginx 服务器在向其发送数据之前先将该数据解压。这些压缩数据可能来自于后端服务器压缩产生或者 Nginx 服务器预压缩产生。ngx_http_gunzip_module 模块便是用来针对不支持 Gzip 压缩数据处理的客户端浏览器，对压缩数据进行解压处理的，与它有关的指令主要有以下几个：gunzip、gunzip_buffers、gzip_http_version、gzip_proxied、gzip_disable 和 gzip_vary 等。

1.gunzip 指令

该指令用于开启或者关闭该模块的功能，其语法结构为：

```
gunzip_static on | off;
```

- *on*，开启该模块的功能。
- *off*，关闭该模块的功能。

该指令默认设置为关闭功能。当功能开启时，如果客户端浏览器不支持 Gzip 处理，Nginx 服务器将返回解压后的数据；如果客户端浏览器支持 Gzip 处理，Nginx 服务器忽略该指令的设置，仍然返回压缩数据。

注意

在某些特殊情况下可能要求 Nginx 服务器始终返回非压缩的数据（比如使用 HTTP 可选模块 ngx_http_addition_module 时，该模块在第 3 章中提到，用于在响应请求的页面开始或者结尾添加文本

信息，它只有在 Nginx 服务器返回非压缩数据时才能正常使用），这就需要在编译 Nginx 程序之前对源码进行适当的处理。

当客户端浏览器不支持 Gzip 数据处理时，使用该模块可以解决数据解析的问题，同时保证 Nginx 服务器与后端服务器交互数据或者本身存储数据时仍然使用压缩数据，从而减少了服务器之间的数据传输量，降低了本地存储空间和缓存的使用率。

2. gunzip_buffers 指令

该指令与 ngx_http_gzip_module 模块中的 gzip_buffers 指令非常类似，都是用于设置 Nginx 服务器解压 Gzip 文件使用缓存空间的大小的，语法结构为：

```
gunzip _buffers number size;
```

- *number*，指定 Nginx 服务器需要向系统申请缓存空间的个数。
- *size*，指定每个缓存空间的大小。

根据该配置项，Nginx 服务器在对 Gzip 数据进行减压时需向系统申请 *number* * *size* 大小的空间。默认情况下 *number* * *size* 的值为 128，其中 *size* 的值也取系统内存页一页的大小，为 4KB 或者 8KB，即：

```
gunzip _buffers 32 4k | 16 8k;
```

其他指令同 ngx_http_gzip_module 模块下的使用方法相同。

注意

该模块是 Nginx 服务器的可选 HTTP 模块，如果要使用，必须在 Nginx 程序配置时添加 --with-http_gunzip_module 指令。

5.4　Gzip 压缩功能的使用

前边介绍了使用 Nginx 服务器的 Gzip 压缩功能的各种指令，列举了一些使用示例。在这一节中，笔者为大家准备了一个完整的 Gzip 压缩实例，以加深大家对该功能的理解。同时，由于 Gzip 压缩功能在使用过程中容易发生一些莫名其妙的问题，笔者对常见的两类问题进行了分析和说明，一类问题涉及客户端浏览器 Gzip 压缩功能的支持，另一类问题涉及 Nginx 服务器作为代理服务器时与其他服务器在交互过程中 Gzip 压缩功能的支持。

5.4.1　Gzip 压缩功能综合配置实例

我们沿用第 2 章的配置文件，在其中加入对 Gzip 压缩功能的配置。以下是配置文件的详细内容，其中加粗斜体部分是新加入的内容：

```
user nobody  nobody;
worker_processes  3;
error_log  logs/error.log;
pid  nginx.pid;
events
{
    use epoll;
```

```
        worker_connections  1024;
}
  http
  {
       include  mime.types;
       default_type  application/octet-stream;
       sendfile  on;
      keepalive_timeout  65;
       log_format  access.log  '$remote_addr-[$time_local]-"$request"-"$http_user_
agent"';
                                                                    #gzip 功能的配置
       gzip on;                                                     #开启 gzip 功能
       gzip_min_length  1024;                                       #响应页数据上限
       gzip_buffers  4 16k;                                         #缓存空间大小
       gzip_comp_level  2;                                          #压缩级别为 2
       gzip_types  text/plain application/x-javascript text/css application/xml;  #压缩源文件类型
       gzip_vary  on;                                               #启用压缩标识
       gunzip_static on;                                            #检查预压缩文件
       server {
             listen  8081;
             server_name  myServer1;
            access_log  /myweb/server1/log/access.log;
             error_page  404   /404.html;
             location /server1/location1 {
                   root  /myweb;
                   index  index.svr1-loc1.htm;
             }
             location /server1/location2 {
                  root  /myweb;
                   index  index.svr1-loc2.htm;
             }
       }
       server {
             listen  8082;
             server_name  192.168.1.3;
             access_log  /myweb/server2/log/access.log;
             error_page  404  /404.html;
            location /server2/location1 {
                   root  /myweb;
                   index  index.svr2-loc1.htm;
             }
             location /svr2/loc2 {
                  alias  /myweb/server2/location2/;
                   index  index.svr2-loc2.htm;
            }
             location = /404.html {
                  root  /myweb;
```

```
                index 404.html;
            }
        }
}
```

在该实例中，我们配置开启 gzip 功能（代码：gzip_min_length　1024），Nginx 服务器用于 Gzip 压缩的缓存空间大小为 4 × 16KB=64KB（代码：　gzip_buffers 4 16k;）。当响应页数据大于 1KB 时（代码：gzip_min_length　1024 行）对类型为 TXT 数据、JS 数据、CSS 数据和 XML 数据（代码：, gzip_types text/plain application/x-javascript text/css application/xml）进行级别为 2 的快速 Gzip 压缩（代码：gzip_comp_level　2;），并在返回数据头部添加“Vary: Accept-Encoding”头域通知客户端浏览器使用了 Gzip 压缩（代码：gzip_vary　on）。如果检测到客户端浏览器不支持 Gzip 压缩功能，Nginx 服务器自动将预压缩过的数据解压后再发送。

为了使得 Nginx 服务器能够在全局范围内应用 Gzip 压缩功能，笔者将其放在了 http 全局块中。如果要对各个虚拟主机差别性对待，我们可以在对应的 server 块中添加各自的 Gzip 配置指令；对于少数虚拟主机的差别性对待，也可以在 http 全局块中配置 Gzip 指令后在对应的少数 server 块中添加配置。比如，在上例中，我们假设需要设置虚拟主机 192.168.1.3 不开启 Gzip 压缩功能，根据在第 2 章中提到的配置文件的“就近原则”，可以简单地添加相关配置，如下列代码所示：

```
...
    server {
        listen  8082;
        server_name  192.168.1.3;
        gzip off;
        access_log   /myweb/server2/log/access.log;
        error_page   404  /404.html;
...
```

http 全局块中的 Gzip 配置不需要改动。

5.4.2　Gzip 压缩功能与 IE6 浏览器运行脚本的兼容问题

该问题涉及客户端浏览器是否支持 Gzip 压缩功能。本节以 IE 6 浏览器为例，分析产生相关问题的原因，说明处理此类问题的方法。

IE 6 浏览器对 Gzip 压缩功能的支持非常不完善，导致包括 Nginx 服务器在内的所有 Web 服务器的 Gzip 压缩功能与 IE 6 的兼容性问题一直困扰许多用户。虽然如今 IE 6 浏览器被使用得越来越少了，但笔者认为还是有必要总结一下相关的问题和解决方法，对大家加深理解 Gzip 压缩功能有一定的帮助。

根据笔者的经验和相关资料的整理，Nginx 服务器开启对 JavaScript 脚本的 Gzip 压缩功能后，遇到 IE 6 浏览器处理响应数据时，经常会遭遇 JavaScript 脚本运行不正常的问题。

第一种情况是，当页面中存在多个 iframe，并且这些 iframe 独立使用相同的 JavaScript 脚本时，可能会出现脚本不运行或者运行不正常的情况。这个问题的发生与 IE 6 在请求 JavaScript 脚本时共享资源有关。有人提出预加载多个 iframe 共用的脚本文件来避免 IE 6 共享 JavaScript 脚本资源时产生的问题，可以比较有效地解决这种情况。预加载 JavaScript 脚本可以在页面中使用：

```
<!--[if IE 6.0]>
<script src="public.js" type="text/c"></script>
<![endif]-->
```

其中，text/c 保证脚本加载但不运行，从而不会对客户端性能造成影响。

第二种情况是，IE 6 通过各种方式从一个页面跳转到另一个页面，并且跳转页响应数据头部的 cache-control 头域设置了 no-cache 指令，这也可能造成 JavaScript 脚本不运行或者运行不正常。该问题的发生与 IE 6 处理缓存数据有关。在微软的 MSDN 中给出了这种情况的解决办法，可以去掉 no-cache 指令，或者使用 Expires 值指定缓存的过期时间。具体内容大家可以查看 http://support.microsoft.com/kb/327286 上的相关内容。

第三种情况是，JavaScript 脚本中的汉字没有被正确编码导致解析失败，因而造成脚本不运行或者运行不正常。这种问题在首次加载页面运行脚本时不会发生，因为 IE 6 可以使用正确的编码对脚本中的汉字进行解析，使脚本运行正常，但当使用刷新功能时，由于 IE 6 不会再次记录上次使用的编码规则，使得脚本中的汉字被解析成乱码，导致脚本运行出现异常。当然，如果强制刷新页面的话，该问题也不会出现。

以上是 IE 6 浏览器运行 Gzip 压缩脚本时可能出现问题的三种常见情况。可以看出，IE 6 对 Gzip 压缩的支持很有缺陷，并且有些情况下没有可靠的解决方案。鉴于如今使用 IE 6 浏览器的客户端相对来说已经很少了，为了避免不必要的麻烦，笔者建议大家通过合理配置 Nginx 服务器，从而当遇到客户端浏览器是 IE 6 及更陈旧的浏览器版本时不要使用 Gzip 压缩功能了。具体的配置办法是使用 gzip_disable 指令，在上节的配置文件中添加下面斜体字所示的内容：

```
...
    gzip_comp_level  2;
    gzip_types  text/plain application/x-javascript text/css application/xml;
    gzip_vary  on;
    gunzip_static on;
    gzip_disable "MSIE [1-6]\.";
...
```

5.4.3　Nginx 与其他服务器交互时产生的 Gzip 压缩功能相关问题

该类问题的产生原因又可以分为两类：一类是多层服务器同时开启 Gzip 压缩功能导致；另一类是多层服务器之间对 Gzip 压缩功能支持能力不同导致。

Gzip 压缩功能在各种服务器产品中应用广泛，包括 IIS、Tomcat 和 Apache 等多种服务器都支持该功能。Nginx 服务器作为前端服务器接收后端服务器返回的数据时，如果 Nginx 和后端服务器同时开启 Gzip 压缩功能，会产生怎样的问题呢?

在笔者的测试过程中发现，如果 Nginx 服务器与后端服务器（比如 Tomcat）同时开启 Gzip 压缩功能对 JavaScript 脚本进行压缩的话，在大多数浏览器中刷新页面会导致脚本运行发生异常，唯一可以运行的浏览器是谷歌的 Chrome 浏览器。笔者分析发现，在客户端首次加载页面时，JavaScript 脚本传输和运行正常，但刷新页面时传输的 JavaScript 脚本发生了缺失现象，返回状态为 304，即请求的网页与上次比没有更新。笔者将其中的一台服务器的 Gzip 压缩功能关闭，这样客户端页面显示和脚本运行都正常了。

因此，该问题的解决办法是，对于包含多层服务器的系统来说，Nginx 服务器作为前端服务器如果开启了 Gzip 压缩功能，后端服务器最好就不要再开启了，否则会导致客户端浏览器在刷新过程中数据下载不完整的问题发生。

另一类情况，Nginx 服务器作为后端服务器和前端服务器进行交互，两类服务器对 Gzip 压缩功能的支持不同导致问题产生也是比较常见的。我们以 Nginx 服务器和 Squid 服务器为例来说明问题的产生。

提示

Squid 服务器通常被用作 Web 缓存服务器，我们将在本书后面的相关章节介绍该服务器与 Nginx 服务器整合应用的相关案例。

3.0 之前版本的 Squid 服务器对 HTTP 1.1 的支持不是特别完善，它仅支持静态压缩，要求 HTTP 响应头中必须包含 Content-length 头域，不能使用 Chunked 编码，但 Nginx 服务器使用由 ngx_http_gzip_module 模块实现 Gzip 压缩功能的情况居多，该模块下默认使用 Chunked 编码进行动态压缩，于是就导致了两类服务器在交互过程中因为对 Gzip 压缩方式支持程度不同产生了问题。

知道了这类问题产生的原因，就可以寻找解决办法了。为了使得 Squid 服务器能够正确接收 Nginx 服务器的数据，我们需要开启 Nginx 服务器 ngx_http_gzip_static_module 模块的功能，对输出数据流进行静态压缩：

```
gzip_static on;
gzip_http_version 1.0;
...
```

我们知道，在 HTTP 1.0 及之前版本中，Content-length 域可有可无。在 HTTP 1.1 及之后版本中，在 Keep-Alive 模式下的 HTTP 响应头中必然会在 Content-length 域和 Chunk 编码中二选一。在非 Keep-Alive 模式下，和 HTTP 1.0 一样，Content-length 可有可无。于是，如果我们使用 HTTP 1.0，客户端浏览器就会得到未经过压缩的数据；如果使用 HTTP 1.1，并且发送 Vary 头域，客户端浏览器就会得到经过压缩的数据。

这里需要提醒大家的是，我们在讲解 ngx_http_gzip_static_module 模块时提到，对于该模块下的 gzip_vary 指令，开启以后针对未压缩的数据在 HTTP 响应头添加“Vary: Accept-Encoding”头域，而不是对所有数据都添加，于是，我们不使用 gzip_vary 指令，而通过 Nginx 配置的 add_header 指令来声明 Gzip 压缩：

```
add_header Vary Accept-Encoding gzip;
```

5.5　本章小结

本章重点介绍了 Nginx 服务器的 Gzip 压缩功能。Gzip 压缩功能依赖于 ngx_http_gunzip_module 模块，它是 Nginx 服务器提升网络请求响应速度的重要功能之一。还介绍了模块支持的 Nginx 配置指令，并提供了 Gzip 压缩功能在实际应用中的配置实例。同时，对常见的应用问题进行了分析，提出了解决方案。相信大家通过本章的学习，能够比较熟练地使用 Nginx 服务器的 Gzip 压缩功能了。

第 6 章

Nginx 服务器的 Rewrite 功能

Rewrite 功能是大多数 Web 服务器支持的一项功能，其在提供重定向服务时起到主要作用。Nginx 服务器的 Rewrite 功能是比较完善的，并且在配置上灵活自由，定制性非常高。

本章主要介绍 Nginx 服务器上 Rewrite 功能的配置和使用，具体的内容包括：

- 后端服务器组的配置指令
- Rewrite 功能的配置指令
- Rewrite 功能的多种应用

6.1 Nginx 后端服务器组的配置的 5 个指令

Nginx 服务器支持设置一组服务器作为后端服务器，在学习 Nginx 服务器的反向代理、负载均衡等重要功能时会经常涉及后端服务器。另外，在介绍 Nginx 服务器的缓存服务时也会有所提及。

服务器组的设置包括几个指令，它们是由标准 HTTP 模块 ngx_http_upstream_module 进行解析和处理的，我们在这里分别介绍一下。

1. upstream 指令

该指令是设置后端服务器组的主要指令，其他的指令都在该指令中进行配置。upstream 指令类似于之前提到的 http 块、server 块等，其语法结构为：

```
upstream name { ... }
```

其中，*name* 是给后端服务器组起的组名。花括号中列出后端服务器组中包含的服务器，其中可以使用下面介绍的其他指令。

默认情况下，某个服务器组接收到请求以后，按照轮叫调度（Round-Robin，RR）策略顺序选择

组内服务器处理请求。如果一个服务器在处理请求的过程中出现错误，请求会被顺次交给组内的下一个服务器进行处理，以此类推，直到返回正常响应。但如果所有的组内服务器都出错，则返回最后一个服务器的处理结果。当然我们可以根据各个服务器处理能力或者资源配置情况的不同，给各个服务器配置不同的权重，让能力强的服务器多处理请求，能力弱的少处理。配置权重的变量包含在 server 指令中。

2. server 指令

该指令用于设置组内的服务器，其语法结构为：

```
server address [parameters];
```

- *address*，服务器的地址，可以是包含端口号的 IP 地址（IP:Port）、域名或者以“unix:”为前缀用于进程间通信的 Unix Domain Socket。
- *parameters*，为当前服务器配置更多属性。这些属性变量包括以下内容：
  - weight=*number*，为组内服务器设置权重，权重值高的服务器被优先用于处理请求。此时组内服务器的选择策略为加权轮叫策略。组内所有服务器的权重默认设置为 1，即采用一般轮叫调度原则处理请求。
  - max_fails=*number*，设置一个请求失败的次数。在一定时间范围内，当对组内某台服务器请求失败的次数超过该变量设置的值时，认为该服务器无效（down）。请求失败的各种情况与 proxy_next_upstream 指令（在学习 Nginx 服务器的缓存机制时，是 fastcgi_next_upstream 指令或者 memcached_next_upstream 指令）的配置相匹配。默认设置为 1。如果设置为 0，则不使用上面的办法检查服务器是否有效。

注意

HTTP 404 状态不被认为是请求失败。

  - fail_timeout=*time*，有两个作用，一是设置 max_fails 指令尝试请求某台组内服务器的时间，即学习 max_fails 指令时提到的“一定时间范围内”；另一个作用是在检查服务器是否有效时，如果一台服务器被认为是无效（down）的，该变量设置的时间为认为服务器无效的持续时间。在这个时间内不再检查该服务器的状态，并一直认为它是无效（down）的。默认设置为 10 s。
  - backup，将某台组内服务器标记为备用服务器，只有当正常的服务器处于无效（down）状态或者繁忙（busy）状态时，该服务器才被用来处理客户端请求。
  - down，将某台组内服务器标记为永久的无效状态，通常与 ip_hash 指令配合使用。该指令在 Nginx 0.6.7 及以上版本中提供。

我们来通过示例加深对 server 指令配置的理解：

```
upstream backend
{
  server backend1.example.com weight=5;
  server 127.0.0.1:8080 max_fails=3 fail_timeout=30s;
  server unix:/tmp/backend3;
}
```

在该示例中，我们设置了一个名为 backend 的服务器组，组内包含三台服务器，分别是基于域名

的 backend1.example.com、基于 IP 地址的 127.0.0.1:8080 和用于进程间通信的 Unix Domain Socket。backend1.example.com 的权重设置为 5，为组内最大，优先接收和处理请求；对本地服务器 127.0.0.1:8080 的状态检查设置是，如果在 30 s 内连续产生 3 次请求失败，则该服务器在之后的 30s 内被认为是无效（down）状态。

3. ip_hash 指令

该指令用于实现会话保持功能，将某个客户端的多次请求定向到组内同一台服务器上，保证客户端与服务器之间建立稳定的会话。只有当该服务器处于无效（down）状态时，客户端请求才会被下一个服务器接收和处理。其语法结构为：

```
ip_hash;
```

ip_hash 技术在一些情况下非常有用，能够避免我们关心的服务器组内各服务器之间会话共享的问题。但是 ip_hash 技术在实际使用过程中也有限制。

首先，ip_hash 指令不能与 server 指令中的 weight 变量一起使用。其次，由于 ip_hash 技术主要根据客户端 IP 地址分配服务器，因此在整个系统中，Nginx 服务器应该是处于最前端的服务器，这样才能获取到客户端的 IP 地址，否则它得到的 IP 地址将是位于它前面的服务器地址，从而就会产生问题。同时要注意，客户端 IP 地址必须是 C 类地址。Nginx 1.3.2 开发版本和 Nginx 1.2.2 稳定版本开始支持 IPv6 地址。

我们来看下面这个示例：

```
upstream backend
{
  ip_hash;
  server myback1.proxy.com;
  server myback2.proxy.com;
}
```

该示例中配置了一个名为 backend 的服务器组，包含两台后端服务器 myback1.proxy.com 和 myback2.proxy.com。在添加 ip_hash 指令后，我们使用同一个客户端向 Nginx 服务器发送请求，将会看到一直是由服务器 myback1.proxy.com 响应；如果注释掉 ip_hash 指令后进行相同的操作，发现组内的两台服务器轮流响应请求。

4. keepalive 指令

该指令用于控制网络连接保持功能。通过该指令，能够保证 Nginx 服务器的工作进程为服务器组打开一部分网络连接，并且将数量控制在一定的范围之内。其语法结构为：

```
keepalive connections;
```

其中，*connections* 为 Nginx 服务器的每一个工作进程允许该服务器组保持的空闲网络连接数的上限值。如果超过该值，工作进程将采用最近最少使用的策略关闭网络连接。

该指令从 Nginx 1.1.4 开始被支持。

注意

该指令不能限制 Nginx 服务器工作进程能够为服务器组开启的总网络连接数。connections 变量在设置时也不要设置得太大，否则会影响服务器组与新网络连接的建立。

5. least_conn 指令

该指令用于配置 Nginx 服务器使用负载均衡策略为网络连接分配服务器组内的服务器。该指令在功能上实现了最少连接负载均衡算法，在选择组内的服务器时，考虑各服务器权重的同时，每次选择的都是当前网络连接最少的那台服务器，如果这样的服务器有多台，就采用加权轮叫原则选择权重最大的服务器。其语法结构为：

```
least_conn;
```

6.2 Rewrite 功能的配置

Rewrite 是 Nginx 服务器提供的一个重要基本功能，其在 Web 服务器产品中几乎是必备的功能，用于实现 URL 的重写。URL 的重写是非常有用的功能，比如它可以让我们在改变网站结构后，无需要求客户端用户修改原来的书签，也无需其他网站修改对我们网站的友情链接；它还可以在一定程度上提高网站的安全性；能够让我们的网站显得更加专业。

适当利用 Rewrite 功能，可以给我们带来很多好处，这一节我们就重点学习一下 Nginx 服务器的 Rewrite 功能。Nginx 服务器的 Rewrite 功能的实现依赖于 PCRE（Perl Compatible Regular Expressions，Perl 兼容的正则表达式）的支持，因此在编译安装 Nginx 服务器之前，需要安装 PCRE 库。

提示

有关 PCRE 的介绍和 PCRE 库的相关下载请参阅其官方网站 http://www.pcre.org/。

Nginx 服务器使用 ngx_http_rewrite_module 模块解析和处理 Rewrite 功能的相关配置。

6.2.1 “地址重写”与“地址转发”

“地址重写”与“转发”在计算机网络领域是两个重要概念，经常被大家提起。但许多人对这两个概念的区别不清楚，甚至混为一谈。在学习 Nginx 服务器的 Rewrite 功能之前，我们有必要先将这对概念进行一下分析和对比，让大家能够有一个清晰的认识。

“地址重写”，实际上是为了实现地址标准化。那么，什么是地址标准化呢？我们来举一个例子。比如在访问 Google 首页的时候，我们在地址栏中可以输入 www.google.com，也可以输入 google.cn，它们都能够准确地指向 Google 首页，从客户端来看，Google 首页同时对应了两个地址，实际上，Google 服务器是在不同的地址中选择了确定的一个，即 www.google.com，进而返回服务器响应的。这个过程就是地址标准化的过程。google.cn 这个地址在服务器中被改变为 www.google.com 的过程就是地址重定向的过程。

“转发”的概念最初和网页的访问并没有太大关系，它是指在网络数据传输过程中数据分组到达路由器或者桥接器后该设备通过检查分组地址并将数据转到相邻局域网上的过程。后来该概念被用在网页访问中，出现了“地址转发”的说法。“地址转发”，是指将一个域名指到另一个已有站点的过程。

从上面的解释，我们可以看到“地址重写”和“地址转发”代表的两个过程是不同的。我们可以总结这两个过程的几点区别：

- 地址转发后客户端浏览器地址栏中的地址显示是不改变的；而地址重写后客户端浏览器地址