Nginx模块开发领域的里程碑之作，阿里巴巴资深Nginx技术专家多年工作经验结晶

深度还原Nginx设计思想，揭示快速开发简单高效Nginx模块的技巧；透彻解析Nginx架构，拓展开发高性能Web服务器的思路

揭秘 系列丛书 UNLEASH

Understanding Nginx

Modules Development and Architecture Resolving

深入理解Nginx

模块开发与架构解析

陶辉 著

机械工业出版社
China Machine Press

深入理解 Nginx：模块开发与架构解析

陶 辉 著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

深入理解 Nginx：模块开发与架构解析 / 陶辉著 . —北京：机械工业出版社，2013.3

ISBN 978-7-111-41478-0

Ⅰ . 深… Ⅱ . 陶… Ⅲ . Web 服务器 Ⅳ . TP393.09

中国版本图书馆 CIP 数据核字（2013）第 033020 号

本书是阿里巴巴资深 Nginx 技术专家呕心沥血之作，是作者多年的经验结晶，也是目前市场上唯一一本通过还原 Nginx 设计思想，剖析 Nginx 架构来帮助读者快速高效开发 HTTP 模块的图书。

本书首先通过介绍官方 Nginx 的基本用法和配置规则，帮助读者了解一般 Nginx 模块的用法，然后重点介绍如何开发 HTTP 模块（含 HTTP 过滤模块）来得到定制的 Nginx，其中包括开发一个功能复杂的模块所需要了解的各种知识，如 Nginx 的基础数据结构、配置项的解析、记录日志的工具以及 upstream、subrequest 的使用方法等。在此基础上，综合 Nginx 框架代码分析 Nginx 的架构，介绍其设计理念和技巧，进一步帮助读者自由、有效地开发出功能丰富、性能一流的 Nginx 模块。

机械工业出版社（北京市西城区百万庄大街 22 号　　邮政编码　100037）

责任编辑：马　超

印刷

2013 年 4 月第 1 版第 1 次印刷

186mm×240 mm • 36.5 印张

标准书号：ISBN 978-7-111-41478-0

定价：89.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991　88361066　　投稿热线：（010）88379604

购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

前　言

为什么要写这本书

当我试图在产品的关键位置设计一个高性能 Web 服务器时，我选择使用成熟的 Nginx。选择它的理由为：首先，它对服务器性能上的挖掘已经达到了很高水平，它能尽量使不同的硬件（包括网卡、硬盘、不同的 CPU 核心）并发运行，同时软件中又没有阻塞进程使之睡眠的代码，从性能上来说，它可以挑战任何服务器。其次，完全基于事件驱动的服务器开发效率往往很不理想，它们要处理的事件过于底层化、细节化，这使得各功能模块无法聚焦于业务，最终产品的功能都较为单一，不会有丰富的可选功能。但 Nginx 却不然，由于它在软件架构上具有优秀的设计，使得 Nginx 完全由许多简单的模块构成，各模块（特别是 HTTP 模块）不用介入底层细节，在尽享分阶段、无阻塞的事件驱动架构下，可以专注于业务功能的实现，这样最终为 Nginx 带来了大量的官方、第三方的功能模块，使得功能同样强大的 Nginx 在产品核心位置上足以担当重任，经受住海量请求的考验。

当 Nginx 已有模块提供的功能不能完全实现我的所有业务需求时，我可以在 Nginx 的后端再搭建一个实现了缺失功能的非 Nginx 服务器，将 Nginx 无法实现的请求反向代理到这台服务器上处理。但这样也有一定的弊端，首先增大了处理请求的开销，其次后端服务器的设计仍然制约着总体性能（它依然需要解决 Nginx 解决过的无阻塞问题，那样才能像 Nginx 一样高效），这样做仅适用于对性能要求不高的场景。唯有开发一个实现了所需功能的自定义 Nginx 模块嵌入到 Nginx 代码中，才能让自己的业务像 Nginx 一样充分挖掘服务器的硬件资源，及时地响应百万级别的并发 TCP 连接。

当我在开发 Nginx 模块之前，试图在市面上找到一本关于 Nginx 模块开发的书籍（无论是中文还是英文）时却一无所获。我只能找到如何使用 Nginx 及其已有模块的书籍。为了开发 Nginx 模块，我只能通过阅读 Nginx 极度缺少注释的源代码，并分析各种官方 Nginx 模块来逐步还原其设计思想，反复尝试、验证着怎样的模块能够使用 Nginx 的基础架构，和丰富的跨平台工具方法，同时符合 Nginx 设计思想，使 Nginx 拥有媲美 Linux 内核的一流效率。这个过程耗费了我很多的精力，因此，我希望今后的 Nginx 使用者、开发者在遇到同样的

问题时，不至于还要很痛苦地阅读源代码来找到模块开发方法，而是简单地按照章节查阅本书，就可以快速找到怎样简单、高效地开发 Nginx 模块，把精力放在业务的实现上。这是我写这本书的第一个目的。

当我们产品中运行的 Nginx 出现了问题时，往往是通过找到错误的配置项、使用方式来解决的，这样也的确能够修复大部分问题。但是更深层次的问题，或者是使用场景比较偏僻，抑或是 Nginx 自身代码考虑得不够全面时，这些问题往往只能由那些花费大量精力研究 Nginx 源代码的工程师来解决。我写作本书的第二个目的是希望通过透彻地解析 Nginx 架构，帮助读者深入理解 Nginx，既能够正确地使用它，也能在它出现任何问题时找到根本原因，进而用最合适的方法修复或者回避问题。

Nginx 是一个优秀的事件驱动框架，虽然它在 HTTP 的处理上非常出色，但它绝不仅仅用于 Web 服务器。Nginx 非常适合开发在传输层以 TCP 对外提供服务的服务器程序。基于 Nginx 框架开发程序有 5 个优势：

1）Nginx 将网络、磁盘及定时器等异步事件的驱动都做了非常好的封装，基于它开发将可以忽略这些事件处理的细节。

2）Nginx 封装了许多平台无关的接口、容器，适用于跨平台开发。

3）优秀的模块化设计，使得开发者可以轻易地复用各种已有的模块，其中既包括基本的读取配置、记录日志等模块，也包括处理请求的诸如 HTTP、mail 等高级功能模块。

4）Nginx 是作为服务器来设计其框架的，因此，它在服务器进程的管理上相当出色，基于它开发服务器程序可以轻松地实现程序的动态升级，子进程的监控、管理，配置项的动态修改生效等。

5）Nginx 充分考虑到各操作系统所擅长的“绝活”，能够使用特殊的系统调用更高效地完成任务时，绝不会去使用低效的通用接口。尤其对于 Linux 操作系统，Nginx 不遗余力地做了大量优化。

当我们期望编写一款能够以低负载处理高并发请求并且主要处理基于 TCP 的服务器程序时，推荐选择 Nginx，它可能会带给我们意外的惊喜。这本书的第三部分，将通过分析 Nginx 的内部架构，帮助读者了解怎样基于 Nginx 开发高效的 TCP 服务器程序：通过开发一种新的模块类型，实现一种新的功能框架来提供极佳的扩展性，使得功能子模块仅关注于业务的开发，忽视底层事件的处理。这是我写作本书的第三个目的。

除了这 3 个主要目的外，我还希望通过这本书向大家展示 Nginx 在服务器开发上的许多巧妙设计，它们或在抽象设计上精妙，或通过操作系统精确、节省地使用硬件资源，这些细节之处的设计都体现了 Igor Sysoev 的不凡功底。即使我们完全不使用 Nginx，学习这些技巧也将有助于我们服务器编程水平的提升。

读者对象

本书适合以下读者阅读。

- ❑ 对 Nginx 及如何将它搭建成一个高性能的 Web 服务器感兴趣的读者。
- ❑ 希望通过开发特定的 HTTP 模块实现高性能 Web 服务器的读者。
- ❑ 希望了解 Nginx 的架构设计，学习怎样充分使用服务器上的硬件资源的读者。
- ❑ 了解如何快速定位、修复 Nginx 中深层次 Bug 的读者。
- ❑ 希望利用 Nginx 提供的框架，设计出任何基于 TCP 的、无阻塞的、易于扩展的服务器的读者。

背景知识

如果仅希望了解怎样使用已有的 Nginx 功能搭建服务器，那么阅读本书不需要什么先决条件。但如果希望通过阅读本书的第二、第三部分，来学习 Nginx 的模块开发和架构设计技巧，则必须了解 C 语言的基本语法。在阅读本书第三部分时，需要读者对 TCP 有一个基本的了解，同时对 Linux 操作系统也应该有简单的了解。

如何阅读本书

我很希望将本书写成一本“step by step”式（循序渐进式）的书籍，因为这样最能节省读者的时间，然而，由于 3 个主要写作目的想解决的问题都不是那么简单，所以这本书只能做一个折中的处理。

在第一部分的前两章中，将只探讨如何使用 Nginx 这一问题。阅读这一部分的读者不需要了解 C 语言，就可以学习如何部署 Nginx，学习如何向其中添加各种官方、第三方的功能模块，如何通过修改配置文件来更改 Nginx 及各模块的功能，如何修改 Linux 操作系统上的参数来优化服务器性能，最终向用户提供企业级的 Web 服务器。这一部分介绍配置项的方式，更偏重于带领对 Nginx 还比较陌生的读者熟悉它，通过了解几个基本 Nginx 模块的配置修改方式，进而使读者可以通过查询官网、第三方网站来了解如何使用所有 Nginx 模块的用法。

在第二部分的第 3 章～第 7 章中，都是以例子来介绍 HTTP 模块的开发方式的，这里有些接近于“step by step”的学习方式，我在写作这一部分时，会通过循序渐进的方式使读者能够快速上手，同时会穿插着介绍其常见用法的基本原理。

在第三部分，将开始介绍 Nginx 的完整框架，阅读到这里时将会了解第二部分中 HTTP 模块为何以此种方式开发，同时将可以轻易地开发出 Nginx 模块。这一部分并不仅仅满足于

阐述 Nginx 架构，而是会探讨其为何如此设计，只有这样才能抛开 HTTP 框架、邮件代理框架，实现一种新的业务框架、一种新的模块类型。

对于 Nginx 的使用还不熟悉的读者应当从第 1 章开始学习，前两章将帮助你快速了解 Nginx。

使用过 Nginx，但对如何开发 Nginx 的 HTTP 模块不太了解的读者可以直接从第 3 章开始学习，在这一章阅读完后，即可编写一个功能大致完整的 HTTP 模块。然而，编写企业级的模块必须阅读完第 4 章才能做到，这一章将会介绍编写产品线上服务器程序时必备的 3 个手段。第 5 章举例说明了两种编写复杂 HTTP 模块的方式，在第三部分会对这两种方式有进一步的说明。第 6 章介绍一种特殊的 HTTP 模块——HTTP 过滤模块的编写方法。第 7 章探讨基础容器的用法，这同样是复杂模块的必备工具。

如果读者对于普通 HTTP 模块的编写已经很熟悉，想深入地实现更为复杂的 HTTP 模块，或者想了解邮件代理服务器的设计与实现，或者希望编写一种新的处理其他协议的模块，或者仅仅想了解 Nginx 的架构设计，都可以直接从第 8 章开始学习，这一章会从整体上系统介绍 Nginx 的模块式设计。第 9 章的事件框架是 Nginx 处理 TCP 的基础，这一章无法跳过。阅读第 8、第 9 章时可能会遇到许多第 7 章介绍过的容器，这时可以回到第 7 章查询其用法和意义。第 10 章～第 12 章介绍 HTTP 框架，通过这 3 章的学习会对 HTTP 模块的开发有深入的了解，同时可以学习 HTTP 框架的优秀设计。第 13 章简单地介绍了邮件代理服务器的设计，它近似于简化版的 HTTP 框架。第 14 章介绍了进程间同步的工具。

为了不让读者陷入代码的“汪洋大海”中，在本书中大量使用了图表，这样可以使读者快速、大体地了解流程和原理。关键地方会直接给出代码，并添加注释加以说明。希望这种方式能够帮助读者减少阅读花费的时间，更快、更好地把握 Nginx，同时深入到细节中。

在本书开始写作时，由于 Nginx 的最新稳定版本是 1.0.14，所以本书是基于此版本来编写的。截止到本书编写完成时，Nginx 的稳定版本已经上升到了 1.2.4。但这不会对本书的阅读造成困扰，因为本书主要是在介绍 Nginx 的基本框架代码，以及怎样使用这些框架代码开发新的 Nginx 模块，而不是介绍 Nginx 的某些功能。在这些基本框架代码中，Nginx 一般不会做任何改变，否则已有的大量 Nginx 模块将无法工作，这种损失也是不可承受的。而且，Nginx 框架为具体的功能模块提供了足够的灵活性，修改功能时很少需要修改框架代码。

Nginx 是跨平台的服务器，然而这本书将只针对最常见的 Linux 操作系统进行分析，这样做一方面是篇幅所限，另一方面则是本书的写作目的主要在于告诉读者如何基于 Nginx 编写代码，而不是怎样在一个具体的操作系统上修改配置来使用 Nginx。因此，即使本书以 Linux 系统为代表讲述 Nginx，也不会影响使用其他操作系统的读者阅读，因为操作系统的差别对阅读本书的影响实在是非常小。

勘误和支持

由于作者的水平有限，加之编写的时间也很仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。为此，我特意创建了一个在线支持与应急方案的二级站点：http://nginx.weebly.com。读者可以将书中的错误发布在 Bug 勘误表页面中，同时如果你遇到任何问题，也可以访问 Q&A 页面，我将尽量在线上为读者提供最满意的解答。书中的全部源文件都将发布在这个网站上，我也会将相应的功能更新及时发布出来。如果你有更多的宝贵意见，也欢迎你发送邮件至我的邮箱 russelltao@foxmail.com，期待能够听到读者的真挚反馈。

致谢

我首先要感谢 Igor Sysoev，他在 Nginx 设计上展现的功力令人折服，正是他的工作成果才让本书的诞生有了意义。

lisa 是机械工业出版社华章公司的优秀编辑，非常值得信任。在这半年的写作过程中，她花费了很多时间、精力来阅读我的书稿，指出了许多文字和格式上的错误，她提出的建议大大提高了本书的可读性。

在这半年时间内，一边工作一边写作给我带来了很大的压力，所以我要感谢我的父母在生活上对我无微不至的照顾，使我可以全力投入到写作中。繁忙的工作之余，写作又占用了休息时间的绝大部分，感谢我的太太对我的体谅和鼓励，让我始终以高昂的斗志投入到本书的写作中。

感谢我工作中的同事们，正是在与他们一起工作的日子里，我才不断地对技术有新的感悟；正是那些充满激情的岁月，才使得我越来越热爱服务器技术的开发。

谨以此书，献给我最亲爱的家人，以及众多热爱 Nginx 的朋友。

陶　辉

目　　录

第一部分

Nginx 能帮我们做什么

本部分内容

第 1 章　研究 Nginx 前的准备工作

第 2 章　Nginx 的配置

第 1 章　研究 Nginx 前的准备工作

2012 年，Nginx 荣获年度云计算开发奖（2012 Cloud Award for Developer of the Year），并成长为世界第二大 Web 服务器。全世界流量最高的前 1000 名网站中，超过 25% 都使用 Nginx 来处理海量的互联网请求。Nginx 已经成为业界高性能 Web 服务器的代名词。

那么，什么是 Nginx ？它有哪些特点？我们选择 Nginx 的理由是什么？如何编译安装 Nginx ？这种安装方式背后隐藏的又是什么样的思想呢？本章将会回答上述问题。

1.1　Nginx 是什么

人们在了解新事物时，往往习惯通过类比来帮助自己理解事物的概貌。那么，我们在学习 Nginx 时也采用同样的方式，先来看看 Nginx 的竞争对手——Apache、Lighttpd、Tomcat、Jetty、IIS，它们都是 Web 服务器，或者叫做 WWW（World Wide Web）服务器，相应地也都具备 Web 服务器的基本功能：基于 REST 架构风格[㊀]，以统一资源描述符（Uniform Resource Identifier，URI）或者统一资源定位符（Uniform Resource Locator，URL）作为沟通依据，通过 HTTP 为浏览器等客户端程序提供各种网络服务。然而，由于这些 Web 服务器在设计阶段就受到许多局限，例如当时的互联网用户规模、网络带宽、产品特点等局限，并且各自的定位与发展方向都不尽相同，使得每一款 Web 服务器的特点与应用场合都很鲜明。[1]

Tomcat 和 Jetty 面向 Java 语言，先天就是重量级的 Web 服务器，它的性能与 Nginx 没有可比性，这里略过。

IIS 只能在 Windows 操作系统上运行。Windows 作为服务器在稳定性与其他一些性能上都不如类 UNIX 操作系统，因此，在需要高性能 Web 服务器的场合下，IIS 可能会被“冷落”。

Apache 的发展时期很长，而且是目前毫无争议的世界第一大 Web 服务器，图 1-1 中是 12 年来（2010 ～ 2012 年）世界 Web 服务器的使用排名情况。

从图 1-1 中可以看出，Apache 目前处于领先地位。

Apache 有许多优点，如稳定、开源、跨平台等，但它出现的时间太长了，在它兴起的年代，互联网的产业规模远远比不上今天，所以它被设计成了一个重量级的、不支持高并发的 Web 服务器。在 Apache 服务器上，如果有数以万计的并发 HTTP 请求同时访问，就会导致服务器上消耗大量内存，操作系统内核对成百上千的 Apache 进程做进程间切换也会消耗大量 CPU 资源，并导致 HTTP 请求的平均响应速度降低，这些都决定了 Apache 不可能成为

㊀ 参见 Roy Fielding 博士的论文《Architectural Styles and the Design of Network-based Software Architectures》，可在 http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm 查看原文。

高性能 Web 服务器，这也促使了 Lighttpd 和 Nginx 的出现。观察图 1-1 中 Nginx 成长的曲线，体会一下 Nginx 抢占市场时的“咄咄逼人”吧。

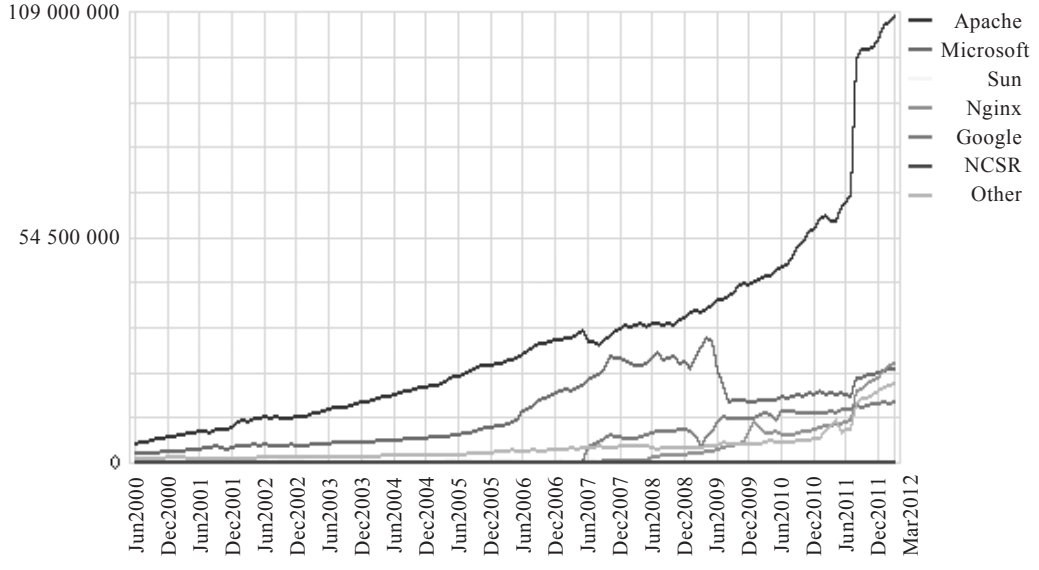


图 1-1　Netcraft 对于 644 275 754 个站点 31.4M 个域名 Web 服务器使用情况的调查结果（2012 年 3 月）

Lighttpd 和 Nginx 一样，都是轻量级、高性能的 Web 服务器，欧美的业界开发者比较钟爱 Lighttpd，而国内的公司更青睐 Nginx，Lighttpd 使用得比较少。

在了解了 Nginx 的竞争对手之后，相信大家对 Nginx 也有了直观感受，下面让我们来正式地认识一下 Nginx 吧。

提示　Nginx 发音：engine ['ɛndʒɪn] X。

来自俄罗斯的 Igor Sysoev 在为 Rambler Media（http://www.rambler.ru/）工作期间，使用 C 语言开发了 Nginx。Nginx 作为 Web 服务器，一直为俄罗斯著名的门户网站 Rambler Media 提供着出色、稳定的服务。

Igor Sysoev 将 Nginx 的代码开源，并且赋予其最自由的 2-clause BSD-like license[⊖]许可

⊖ BSD（Berkeley Software Distribution）许可协议是自由软件（开源软件的一个子集）中使用最广泛的许可协议之一。与其他许可协议相比，BSD 许可协议从 GNU 通用公共许可协议（GPL）到限制重重的著作权（copyright）都要宽松一些，事实上，它跟公有领域更为接近。BSD 许可协议被认为是 copycenter（中间版权），界于标准的 copyright 与 GPL 的 copyleft 之间。

2-clause BSD-like license 是 BSD 许可协议中最宽松的一种，它对开发者再次使用 BSD 软件只有两个基本的要求：一是如果再发布的产品中包含源代码，则在源代码中必须带有原来代码中的 BSD 协议；二是如果再发布的只是二进制类库 / 软件，则需要在类库 / 软件的文档和版权声明中包含原来代码中的 BSD 协议。

证。由于 Nginx 使用基于事件驱动的架构能够并发处理百万级别的 TCP 连接，高度模块化的设计和自由的许可证使得扩展 Nginx 功能的第三方模块层出不穷，而且优秀的设计带来了极佳的稳定性，因此其作为 Web 服务器被广泛应用到大流量的网站上，包括腾讯、新浪、网易、淘宝等访问量巨大的网站。

2012 年 2 月和 3 月 Netcraft 对 Web 服务器的调查如表 1-1 所示，可以看出，Nginx 的市场份额越来越大。

表 1-1 Netcraft 对于 Web 服务器市场占有率前 4 位软件的调查（2012 年 2 月和 3 月）

Web 服务器	2012 年 2 月	市场占有率	2012 年 3 月	市场占有率	占有率变化
Apache	106 664 061	57.45%	108 035 584	57.46%	0.01
Nginx	23 590 737	12.71%	24 011 199	12.77%	0.06
Microsoft IIS	22 363 730	12.05%	22 537 872	11.99%	–0.06
Google Web Server	14 316 485	7.71%	14 438 358	7.68%	–0.03

Nginx 是一个跨平台的 Web 服务器，可运行在 Linux、FreeBSD、Solaris、AIX、Mac OS、Windows 等操作系统上，并且它还可以使用当前操作系统特有的一些高效 API 来提高自己的性能。

例如，对于高效处理大规模并发连接，它支持 Linux 上的 epoll（epoll 是 Linux 上处理大并发网络连接的利器，9.6.1 节中将会详细说明 epoll 的工作原理）、Solaris 上的 event ports 和 FreeBSD 上的 kqueue 等。

又如，对于 Linux，Nginx 支持其独有的 sendfile 系统调用，这个系统调用可以高效地把硬盘中的数据发送到网络上（不需要先把硬盘数据复制到用户态内存上再发送），这极大地减少了内核态与用户态数据间的复制动作。

种种迹象都表明，Nginx 以性能为王。

2011 年 7 月，Nginx 正式成立公司，由 Igor Sysoev 担任 CTO，立足于提供商业级的 Web 服务器。

1.2 为什么选择 Nginx

为什么选择 Nginx ？因为它具有以下特点：

（1）更快

这表现在两个方面：一方面，在正常情况下，单次请求会得到更快的响应；另一方面，在高峰期（如有数以万计的并发请求），Nginx 可以比其他 Web 服务器更快地响应请求。

实际上，本书第三部分中大量的篇幅都是在说明 Nginx 是如何做到这两点的。

（2）高扩展性

Nginx 的设计极具扩展性，它完全是由多个不同功能、不同层次、不同类型且耦合度极

低的模块组成。因此，当对某一个模块修复 Bug 或进行升级时，可以专注于模块自身，无须在意其他。而且在 HTTP 模块中，还设计了 HTTP 过滤器模块：一个正常的 HTTP 模块在处理完请求后，会有一串 HTTP 过滤器模块对请求的结果进行再处理。这样，当我们开发一个新的 HTTP 模块时，不但可以使用诸如 HTTP 核心模块、events 模块、log 模块等不同层次或者不同类型的模块，还可以原封不动地复用大量已有的 HTTP 过滤器模块。这种低耦合度的优秀设计，造就了 Nginx 庞大的第三方模块，当然，公开的第三方模块也如官方发布的模块一样容易使用。

Nginx 的模块都是嵌入到二进制文件中执行的，无论官方发布的模块还是第三方模块都是如此。这使得第三方模块一样具备极其优秀的性能，充分利用 Nginx 的高并发特性，因此，许多高流量的网站都倾向于开发符合自己业务特性的定制模块。

（3）高可靠性

高可靠性是我们选择 Nginx 的最基本条件，因为 Nginx 的可靠性是大家有目共睹的，很多家高流量网站都在核心服务器上大规模使用 Nginx。Nginx 的高可靠性来自于其核心框架代码的优秀设计、模块设计的简单性；另外，官方提供的常用模块都非常稳定，每个 worker 进程相对独立，master 进程在 1 个 worker 进程出错时可以快速“拉起”新的 worker 子进程提供服务。

（4）低内存消耗

一般情况下，10 000 个非活跃的 HTTP Keep-Alive 连接在 Nginx 中仅消耗 2.5MB 的内存，这是 Nginx 支持高并发连接的基础。

从第 3 章开始，我们会接触到 Nginx 在内存中为了维护一个 HTTP 连接所分配的对象，届时将会看到，实际上 Nginx 一直在为用户考虑（尤其是在高并发时）如何使得内存的消耗更少。

（5）单机支持 10 万以上的并发连接

这是一个非常重要的特性！随着互联网的迅猛发展和互联网用户数量的成倍增长，各大公司、网站都需要应付海量并发请求，一个能够在峰值期顶住 10 万以上并发请求的 Server，无疑会得到大家的青睐。理论上，Nginx 支持的并发连接上限取决于内存，10 万远未封顶。当然，能够及时地处理更多的并发请求，是与业务特点紧密相关的，本书第 8 ～ 11 章将会详细说明如何实现这个特点。

（6）热部署

master 管理进程与 worker 工作进程的分离设计，使得 Nginx 能够提供热部署功能，即可以在 7×24 小时不间断服务的前提下，升级 Nginx 的可执行文件。当然，它也支持不停止服务就更新配置项、更换日志文件等功能。

（7）最自由的 BSD 许可协议

这是 Nginx 可以快速发展的强大动力。BSD 许可协议不只是允许用户免费使用 Nginx，

它还允许用户在自己的项目中直接使用或修改 Nginx 源码，然后发布。这吸引了无数开发者继续为 Nginx 贡献自己的智慧。

以上 7 个特点当然不是 Nginx 的全部，拥有无数个官方功能模块、第三方功能模块使得 Nginx 能够满足绝大部分应用场景，这些功能模块间可以叠加以实现更加强大、复杂的功能，有些模块还支持 Nginx 与 Perl、Lua 等脚本语言集成工作，大大提高了开发效率。这些特点促使用户在寻找一个 Web 服务器时更多考虑 Nginx。

当然，选择 Nginx 的核心理由还是它能在支持高并发请求的同时保持高效的服务。

如果 Web 服务器的业务访问量巨大，就需要保证在数以百万计的请求同时访问服务时，用户可以获得良好的体验，不会出现并发访问量达到一个数字后，新的用户无法获取服务，或者虽然成功地建立起了 TCP 连接，但大部分请求却得不到响应的情况。

通常，高峰期服务器的访问量可能是正常情况下的许多倍，若有热点事件的发生，可能会导致正常情况下非常顺畅的服务器直接“挂死”。然而，如果在部署服务器时，就预先针对这种情况进行扩容，又会使得正常情况下所有服务器的负载过低，这会造成大量的资源浪费。因此，我们会希望在这之间取得平衡，也就是说，在低并发压力下，用户可以获得高速体验，而在高并发压力下，更多的用户都能接入，可能访问速度会下降，但这只应受制于带宽和处理器的速度，而不应该是服务器设计导致的软件瓶颈。

事实上，由于中国互联网用户群体的数量巨大，致使对 Web 服务器的设计往往要比欧美公司更加困难。例如，对于全球性的一些网站而言，欧美用户分布在两个半球，欧洲用户活跃时，美洲用户通常在休息，反之亦然。而国内巨大的用户群体则对业界的程序员提出更高的挑战，早上 9 点和晚上 20 点到 24 点这些时间段的并发请求压力是非常巨大的。尤其节假日、寒暑假到来之时，更会对服务器提出极高的要求。

另外，国内业务上的特性，也会引导用户在同一时间大并发地访问服务器。例如，许多 SNS 网页游戏会在固定的时间点刷新游戏资源或者允许“偷菜”等好友互动操作。这些会导致服务器处理高并发请求的压力增大。

上述情形都对我们的互联网服务在大并发压力下是否还能够给予用户良好的体验提出了更高的要求。若要提供更好的服务，那么可以从多方面入手，例如，修改业务特性、引导用户从高峰期分流或者把服务分层分级、对于不同并发压力给用户提供不同级别的服务等。但最根本的是，Web 服务器要能支持大并发压力下的正常服务，这才是关键。

快速增长的互联网用户群以及业内所有互联网服务提供商越来越好的用户体验，都促使我们在大流量服务中用 Nginx 取代其他 Web 服务器。Nginx 先天的事件驱动型设计、全异步的网络 I/O 处理机制、极少的进程间切换以及许多优化设计，都使得 Nginx 天生善于处理高并发压力下的互联网请求，同时 Nginx 降低了资源消耗，可以把服务器硬件资源“压榨”到极致。

1.3　准备工作

由于 Linux 具有免费、使用广泛、商业支持越来越完善等特点，本书将主要针对 Linux 上运行的 Nginx 来进行介绍。需要说明的是，本书不是使用手册，而是介绍 Nginx 作为 Web 服务器的设计思想，以及如何更有效地使用 Nginx 达成目的，而这些内容在各操作系统上基本是相通的（除了第 9 章关于事件驱动方式以及第 14 章的进程间同步方式在类 UNIX 操作系统上略有不同以外）。

1.3.1　Linux 操作系统

首先我们需要一个内核为 Linux 2.6 及以上版本的操作系统，因为 Linux 2.6 及以上内核才支持 epoll，而在 Linux 上使用 select 或 poll 来解决事件的多路复用，是无法解决高并发压力问题的。

我们可以使用 uname -a 命令来查询 Linux 内核版本，例如：

```
:wehf2wng001:root > uname -a
Linux wehf2wng001 2.6.18-128.el5 #1 SMP Wed Jan 21 10:41:14 EST 2009 x86_64 x86_64
x86_64 GNU/Linux
```

执行结果表明内核版本是 2.6.18，符合我们的要求。

1.3.2　使用 Nginx 的必备软件

如果要使用 Nginx 的常用功能，那么首先需要确保该操作系统上至少安装了如下软件。

（1）GCC 编译器

GCC（GNU Compiler Collection）可用来编译 C 语言程序。Nginx 不会直接提供二进制可执行程序（1.2.x 版本中已经开始提供某些操作系统上的二进制安装包了，不过，本书探讨如何开发 Nginx 模块是必须通过直接编译源代码进行的），这有许多原因，本章后面会详述。我们可以使用最简单的 yum 方式安装 GCC，例如：

```
yum install -y gcc
```

GCC 是必需的编译工具。在第 3 章会提到如何使用 C++ 来编写 Nginx HTTP 模块，这时就需要用到 G++ 编译器了。G++ 编译器也可以用 yum 安装，例如：

```
yum install -y gcc-c++
```

Linux 上有许多软件安装方式，yum 只是其中比较方便的一种，其他方式这里不再赘述。

（2）PCRE 库

PCRE（Perl Compatible Regular Expressions，Perl 兼容正则表达式）是由 Philip Hazel 开发的函数库，目前为很多软件所使用，该库支持正则表达式。它由 RegEx 演化而来，实际

上，Perl 正则表达式也是源自于 Henry Spencer 写的 RegEx。

如果我们在配置文件 nginx.conf 里使用了正则表达式，那么在编译 Nginx 时就必须把 PCRE 库编译进 Nginx，因为 Nginx 的 HTTP 模块要靠它来解析正则表达式。当然，如果你确认不会使用正则表达式，就不必安装它。其 yum 安装方式如下：

```
yum install -y pcre pcre-devel
```

pcre-devel 是使用 PCRE 做二次开发时所需要的开发库，包括头文件等，这也是编译 Nginx 所必须使用的。

（3）zlib 库

zlib 库用于对 HTTP 包的内容做 gzip 格式的压缩，如果我们在 nginx.conf 里配置了 gzip on，并指定对于某些类型（content-type）的 HTTP 响应使用 gzip 来进行压缩以减少网络传输量，那么，在编译时就必须把 zlib 编译进 Nginx。其 yum 安装方式如下：

```
yum install -y zlib zlib-devel
```

同理，zlib 是直接使用的库，zlib-devel 是二次开发所需要的库。

（4）OpenSSL 开发库

如果我们的服务器不只是要支持 HTTP，还需要在更安全的 SSL 协议上传输 HTTP，那么就需要拥有 OpenSSL 了。另外，如果我们想使用 MD5、SHA1 等散列函数，那么也需要安装它。其 yum 安装方式如下：

```
yum install -y openssl openssl-devel
```

上面所列的 4 个库只是完成 Web 服务器最基本功能所必需的。

Nginx 是高度自由化的 Web 服务器，它的功能是由许多模块来支持的。而这些模块可根据我们的使用需求来定制，如果某些模块不需要使用则完全不必理会它。同样，如果使用了某个模块，而这个模块使用了一些类似 zlib 或 OpenSSL 等的第三方库，那么就必须先安装这些软件。

1.3.3 磁盘目录

要使用 Nginx，还需要在 Linux 文件系统上准备以下目录。

（1）Nginx 源代码存放目录

该目录用于放置从官网上下载的 Nginx 源码文件，以及第三方或我们自己所写的模块源代码文件。

（2）Nginx 编译阶段产生的中间文件存放目录

该目录用于放置在 configure 命令执行后所生成的源文件及目录，以及 make 命令执行后生成的目标文件和最终连接成功的二进制文件。默认情况下，configure 命令会将该目录命名

为 objs，并放在 Nginx 源代码目录下。

（3）部署目录

该目录存放实际 Nginx 服务运行期间所需要的二进制文件、配置文件等。默认情况下，该目录为 /usr/local/nginx。

（4）日志文件存放目录

日志文件通常会比较大，当研究 Nginx 的底层架构时，需要打开 debug 级别的日志，这个级别的日志非常详细，会导致日志文件的大小增长得极快，需要预先分配一个拥有更大磁盘空间的目录。

1.3.4　Linux 内核参数的优化

由于默认的 Linux 内核参数考虑的是最通用的场景，这明显不符合用于支持高并发访问的 Web 服务器的定义，所以需要修改 Linux 内核参数，使得 Nginx 可以拥有更高的性能。

在优化内核时，可以做的事情很多，不过，我们通常会根据业务特点来进行调整，当 Nginx 作为静态 Web 内容服务器、反向代理服务器或是提供图片缩略图功能（实时压缩图片）的服务器时，其内核参数的调整都是不同的。这里只针对最通用的、使 Nginx 支持更多并发请求的 TCP 网络参数做简单说明。

首先，需要修改 /etc/sysctl.conf 来更改内核参数。例如，最常用的配置：

```
fs.file-max = 999999
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_keepalive_time = 600
net.ipv4.tcp_fin_timeout = 30
net.ipv4.tcp_max_tw_buckets = 5000
net.ipv4.ip_local_port_range = 1024    61000
net.ipv4.tcp_rmem = 4096 32768 262142
net.ipv4.tcp_wmem = 4096 32768 262142
net.core.netdev_max_backlog = 8096
net.core.rmem_default = 262144
net.core.wmem_default = 262144
net.core.rmem_max = 2097152
net.core.wmem_max = 2097152
net.ipv4.tcp_syncookies = 1
net.ipv4.tcp_max_syn.backlog=1024
```

然后执行 sysctl -p 命令，使上述修改生效。

上面的参数意义解释如下：

- file-max ：这个参数表示进程（比如一个 worker 进程）可以同时打开的最大句柄数，这个参数直接限制最大并发连接数，需根据实际情况配置。
- tcp_tw_reuse：这个参数设置为 1，表示允许将 TIME-WAIT 状态的 socket 重新用于新的 TCP 连接，这对于服务器来说很有意义，因为服务器上总会有大量 TIME-WAIT 状

态的连接。

- ❑ tcp_keepalive_time：这个参数表示当 keepalive 启用时，TCP 发送 keepalive 消息的频度。默认是 2 小时，若将其设置得小一些，可以更快地清理无效的连接。
- ❑ tcp_fin_timeout：这个参数表示当服务器主动关闭连接时，socket 保持在 FIN-WAIT-2 状态的最大时间。
- ❑ tcp_max_tw_buckets：这个参数表示操作系统允许 TIME_WAIT 套接字数量的最大值，如果超过这个数字，TIME_WAIT 套接字将立刻被清除并打印警告信息。该参数默认为 180 000，过多的 TIME_WAIT 套接字会使 Web 服务器变慢。
- ❑ tcp_max_syn_backlog：这个参数表示 TCP 三次握手建立阶段接收 SYN 请求队列的最大长度，默认为 1024，将其设置得大一些可以使出现 Nginx 繁忙来不及 accept 新连接的情况时，Linux 不至于丢失客户端发起的连接请求。
- ❑ ip_local_port_range：这个参数定义了在 UDP 和 TCP 连接中本地（不包括连接的远端）端口的取值范围。
- ❑ net.ipv4.tcp_rmem：这个参数定义了 TCP 接收缓存（用于 TCP 接收滑动窗口）的最小值、默认值、最大值。
- ❑ net.ipv4.tcp_wmem：这个参数定义了 TCP 发送缓存（用于 TCP 发送滑动窗口）的最小值、默认值、最大值。
- ❑ netdev_max_backlog：当网卡接收数据包的速度大于内核处理的速度时，会有一个队列保存这些数据包。这个参数表示该队列的最大值。
- ❑ rmem_default：这个参数表示内核套接字接收缓存区默认的大小。
- ❑ wmem_default：这个参数表示内核套接字发送缓存区默认的大小。
- ❑ rmem_max：这个参数表示内核套接字接收缓存区的最大大小。
- ❑ wmem_max：这个参数表示内核套接字发送缓存区的最大大小。

注意 滑动窗口的大小与套接字缓存区会在一定程度上影响并发连接的数目。每个 TCP 连接都会为维护 TCP 滑动窗口而消耗内存，这个窗口会根据服务器的处理速度收缩或扩张。

参数 wmem_max 的设置，需要平衡物理内存的总大小、Nginx 并发处理的最大连接数量（由 nginx.conf 中的 worker_processes 和 worker_connections 参数决定）而确定。当然，如果仅仅为了提高并发量使服务器不出现 Out Of Memory 问题而去降低滑动窗口大小，那么并不合适，因为滑动窗口过小会影响大数据量的传输速度。rmem_default、wmem_default、rmem_max、wmem_max 这 4 个参数的设置需要根据我们的业务特性以及实际的硬件成本来综合考虑。

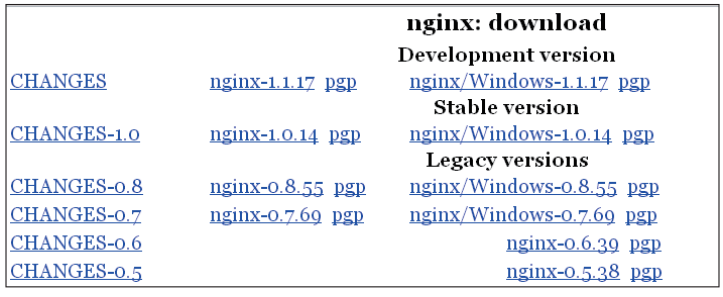
- ❑ tcp_syncookies：该参数与性能无关，用于解决 TCP 的 SYN 攻击。

1.3.5　获取 Nginx 源码

可以在 Nginx 官方网站（http://nginx.org/en/download.html）获取 Nginx 源码包。将下载的 nginx-1.0.14.tar.gz 源码压缩包放置到准备好的 Nginx 源代码目录中，然后解压。例如：

```
tar -zxvf nginx-1.0.14.tar.gz
```

本书编写时的 Nginx 最新稳定版本为 1.0.14（如图 1-2 所示），本书后续部分都将以此版本作为基准。当然，本书将要说明的 Nginx 核心代码一般不会有改动（否则大量第三方模块的功能就无法保证了），即使下载其他版本的 Nginx 源码包也不会影响阅读本书。



nginx: download

Development version

CHANGES	nginx-1.1.17 pgp	nginx/Windows-1.1.17 pgp

Stable version

CHANGES-1.0	nginx-1.0.14 pgp	nginx/Windows-1.0.14 pgp

Legacy versions

CHANGES-0.8	nginx-0.8.55 pgp	nginx/Windows-0.8.55 pgp
CHANGES-0.7	nginx-0.7.69 pgp	nginx/Windows-0.7.69 pgp
CHANGES-0.6		nginx-0.6.39 pgp
CHANGES-0.5		nginx-0.5.38 pgp

图 1-2　Nginx 的不同版本

1.4　编译安装 Nginx

安装 Nginx 最简单的方式是，进入 nginx-1.0.14 目录后执行以下 3 行命令：

```
./configure
make
make install
```

configure 命令做了大量的“幕后”工作，包括检测操作系统内核和已经安装的软件，参数的解析，中间目录的生成以及根据各种参数生成一些 C 源码文件、Makefile 文件等。

make 命令根据 configure 命令生成的 Makefile 文件编译 Nginx 工程，并生成目标文件、最终的二进制文件。

make install 命令根据 configure 执行时的参数将 Nginx 部署到指定的安装目录，包括相关目录的建立和二进制文件、配置文件的复制。

1.5　configure 详解

可以看出，configure 命令至关重要，下文将详细介绍如何使用 configure 命令，并分析

configure 到底是如何工作的，从中我们也可以看出 Nginx 的一些设计思想。

1.5.1 configure 的命令参数

使用 help 命令可以查看 configure 包含的参数。

```
./configure --help
```

这里不一一列出 help 的结果，只是把它的参数分为了四大类型，下面将会详述各类型下所有参数的用法和意义。

1. 路径相关的参数

表 1-2 列出了 Nginx 在编译期、运行期中与路径相关的各种参数。

表 1-2 configure 支持的路径相关参数

参数名称	意 义	默认值
--prefix=PATH	Nginx 安装部署后的根目录	默认为 /usr/local/nginx 目录。注意：这个目标的设置会影响其他参数中的相对目录。例如，如果设置了 --sbin-path=sbin/nginx，那么实际上可执行文件会被放到 /usr/local/nginx/sbin/nginx 中
--sbin-path=PATH	可执行文件的放置路径	<prefix>/sbin/nginx
--conf-path=PATH	配置文件的放置路径	<prefix>/conf/nginx.conf
--error-log-path=PATH	error 日志文件的放置路径。error 日志用于定位问题，可输出多种级别（包括 debug 调试级别）的日志。它的配置非常灵活，可以在 nginx.conf 里配置为不同请求的日志并输出到不同的 log 文件中。这里是默认的 Nginx 核心日志路径	<prefix>/logs/error.log
--pid-path=PATH	pid 文件的存放路径。这个文件里仅以 ASC II 码存放着 Nginx master 的进程 ID，有了这个进程 ID，在使用命令行（例如 nginx -s reload）通过读取 master 进程 ID 向 master 进程发送信号时，才能对运行中的 Nginx 服务产生作用	<prefix>/logs/nginx.pid
--lock-path=PATH	lock 文件的放置路径	<prefix>/logs/nginx.lock
--builddir=DIR	configure 执行时与编译期间产生的临时文件放置的目录，包括产生的 Makefile、C 源文件、目标文件、可执行文件等	<nginx source path>/objs
--with-perl_modules_path=PATH	perl module 放置的路径。只有使用了第三方的 perl module，才需要配置这个路径	无
--with-perl=PATH	perl binary 放置的路径。如果配置的 Nginx 会执行 Perl 脚本，那么就必须要设置此路径	无
--http-log-path=PATH	access 日志放置的位置。每一个 HTTP 请求在结束时都会记录的访问日志	<prefix>/logs/access.log

（续）

参数名称	意　义	默认值
--http-client-body-temp-path=PATH	处理 HTTP 请求时如果请求的包体需要暂时存放到临时磁盘文件中，则把这样的临时文件放置到该路径下	<prefix>/client_body_temp
--http-proxy-temp-path=PATH	Nginx 作为 HTTP 反向代理服务器时，上游服务器产生的 HTTP 包体在需要临时存放到磁盘文件时（详见 12.8 节），这样的临时文件将放到该路径下	<prefix>/proxy_temp
--http-fastcgi-temp-path=PATH	Fastcgi 所使用临时文件的放置目录	<prefix>/fastcgi_temp
--http-uwsgi-temp-path=PATH	uWSGI 所使用临时文件的放置目录	<prefix>/uwsgi_temp
--http-scgi-temp-path=PATH	SCGI 所使用临时文件的放置目录	<prefix>/scgi_temp

2. 编译相关的参数

表 1-3 列出了编译 Nginx 时与编译器相关的参数。

表 1-3　configure 支持的编译相关参数

编译参数	意　义
--with-cc=PATH	C 编译器的路径
--with-cpp=PATH	C 预编译器的路径
--with-cc-opt=OPTIONS	如果希望在 Nginx 编译期间指定加入一些编译选项，如指定宏或者使用 -I 加入某些需要包含的目录，这时可以使用该参数达成目的
--with-ld-opt=OPTIONS	最终的二进制可执行文件是由编译后生成的目标文件与一些第三方库链接生成的，在执行链接操作时可能会需要指定链接参数，--with-ld-opt 就是用于加入链接时的参数。例如，如果我们希望将某个库链接到 Nginx 程序中，需要在这里加入 --with-ld-opt=-llibraryName -LlibraryPath，其中 libraryName 是目标库的名称，libraryPath 则是目标库所在的路径
--with-cpu-opt=CPU	指定 CPU 处理器架构，只能从以下取值中选择：pentium、pentiumpro、pentium3、pentium4、athlon、opteron、sparc32、sparc64、ppc64

3. 依赖软件的相关参数

表 1-4 ～表 1-8 列出了 Nginx 依赖的常用软件支持的参数。

表 1-4　PCRE 的设置参数

PCRE 库的设置参数	意　义
--without-pcre	如果确认 Nginx 不用解析正则表达式，也就是说，nginx.conf 配置文件中不会出现正则表达式，那么可以使用这个参数
--with-pcre	强制使用 PCRE 库
--with-pcre=DIR	指定 PCRE 库的源码位置，在编译 Nginx 时会进入该目录编译 PCRE 源码
--with-pcre-opt=OPTIONS	编译 PCRE 源码时希望加入的编译选项

表 1-5 OpenSSL 的设置参数

OpenSSL 库的设置参数	意 义
--with-openssl=DIR	指定 OpenSSL 库的源码位置，在编译 Nginx 时会进入该目录编译 OpenSSL 源码 注意：如果 Web 服务器支持 HTTPS，也就是 SSL 协议，Nginx 要求必须使用 OpenSSL。可以访问 http://www.openssl.org/ 免费下载
--with-openssl-opt=OPTIONS	编译 OpenSSL 源码时希望加入的编译选项

表 1-6 原子库的设置参数

atomic（原子）库的设置参数	意 义
--with-libatomic	强制使用 atomic 库。atomic 库是 CPU 架构独立的一种原子操作的实现。它支持以下体系架构：x86（包括 i386 和 x86_64）、PPC64、Sparc64（v9 或更高版本）或者安装了 GCC 4.1.0 及更高版本的架构。14.3 节介绍了原子操作在 Nginx 中的实现
--with-libatomic=DIR	atomic 库所在的位置

表 1-7 散列函数库的设置参数

散列函数库的设置参数	意义
--with-MD5=DIR	指定 MD5 库的源码位置，在编译 Nginx 时会进入该目录编译 MD5 源码 注意：Nginx 源码中已经有了 MD5 算法的实现，如果没有特殊需求，那么完全可以使用 Nginx 自身实现的 MD5 算法
--with-MD5-opt=OPTIONS	编译 MD5 源码时希望加入的编译选项
---with-MD5-asm	使用 MD5 的汇编源码
--with-SHA1=DIR	指定 SHA1 库的源码位置，在编译 Nginx 时会进入该目录编译 SHA1 源码。 注意：OpenSSL 中已经有了 SHA1 算法的实现。如果已经安装了 OpenSSL，那么完全可以使用 OpenSSL 实现的 SHA1 算法
--with-SHA1-opt=OPTIONS	编译 SHA1 源码时希望加入的编译选项
--with-SHA1-asm	使用 SHA1 的汇编源码

表 1-8 zlib 库的设置参数

zlib 库的设置参数	意 义
--with-zlib=DIR	指定 zlib 库的源码位置，在编译 Nginx 时会进入该目录编译 zlib 源码。如果使用了 gzip 压缩功能，就需要 zlib 库的支持
--with-zlib-opt=OPTIONS	编译 zlib 源码时希望加入的编译选项
--with-zlib-asm=CPU	指定对特定的 CPU 使用 zlib 库的汇编优化功能，目前仅支持两种架构：pentium 和 pentiumpro

4. 模块相关的参数

除了少量核心代码外，Nginx 完全是由各种功能模块组成的。这些模块会根据配置参数决定自己的行为，因此，正确地使用各个模块非常关键。在 configure 的参数中，我们把它们分为五大类。

❑ 事件模块。
❑ 默认即编译进入 Nginx 的 HTTP 模块。
❑ 默认不会编译进入 Nginx 的 HTTP 模块。
❑ 邮件代理服务器相关的 mail 模块。
❑ 其他模块。

（1）事件模块

表 1-9 中列出了 Nginx 可以选择哪些事件模块编译到产品中。

表 1-9　configure 支持的事件模块参数

编译参数	意　义
--with-rtsig_module	使用 rtsig module 处理事件驱动 默认情况下，Nginx 是不安装 rtsig module 的，即不会把 rtsig module 编译进最终的 Nginx 二进制程序中
--with-select_module	使用 select module 处理事件驱动 select 是 Linux 提供的一种多路复用机制，在 epoll 调用没有诞生前，例如在 Linux 2.4 及其之前的内核中，select 用于支持服务器提供高并发连接 默认情况下，Nginx 是不安装 select module 的，但如果没有找到其他更好的事件模块，该模块将会被安装
--without-select_module	不安装 select module
--with-poll_module	使用 poll module 处理事件驱动 poll 的性能与 select 类似，在大量并发连接下性能都远不如 epoll。默认情况下，Nginx 是不安装 poll module 的
--without-poll_module	不安装 poll module
--with-aio_module	使用 AIO 方式处理事件驱动 注意：这里的 aio module 只能与 FreeBSD 操作系统上的 kqueue 事件处理机制合作，Linux 上无法使用 默认情况下是不安装 aio module 的

（2）默认即编译进入 Nginx 的 HTTP 模块

表 1-10 列出了默认就会编译进 Nginx 的核心 HTTP 模块，以及如何把这些 HTTP 模块从产品中去除。

表 1-10　configure 中默认编译到 Nginx 中的 HTTP 模块参数

默认安装的 HTTP 模块	意　义
--without-http_charset_module	不安装 http charset module。这个模块可以将服务器发出的 HTTP 响应重编码
--without-http_gzip_module	不安装 http gzip module。在服务器发出的 HTTP 响应包中，这个模块可以按照配置文件指定的 content-type 对特定大小的 HTTP 响应包体执行 gzip 压缩
--without-http_ssi_module	不安装 http ssi module。该模块可以在向用户返回的 HTTP 响应包体中加入特定的内容，如 HTML 文件中固定的页头和页尾
--without-http_userid_module	不安装 http userid module。这个模块可以通过 HTTP 请求头部信息里的一些字段认证用户信息，以确定请求是否合法

（续）

默认安装的 HTTP 模块	意 义
--without-http_access_module	不安装 http access module。这个模块可以根据 IP 地址限制能够访问服务器的客户端
--without-http_auth_basic_module	不安装 http auth basic module。这个模块可以提供最简单的用户名 / 密码认证
--without-http_autoindex_module	不安装 http autoindex module。该模块提供简单的目录浏览功能
--without-http_geo_module	不安装 http geo module。这个模块可以定义一些变量，这些变量的值将与客户端 IP 地址关联，这样 Nginx 针对不同的地区的客户端（根据 IP 地址判断）返回不一样的结果，例如不同地区显示不同语言的网页
--without-http_map_module	不安装 http map module。这个模块可以建立一个 key/value 映射表，不同的 key 得到相应的 value，这样可以针对不同的 URL 做特殊处理。例如，返回 302 重定向响应时，可以期望 URL 不同时返回的 Location 字段也不一样
--without-http_split_clients_module	不安装 http split client module。该模块会根据客户端的信息，例如 IP 地址、header 头、cookie 等，来区分处理
--without-http_referer_module	不安装 http referer module。该模块可以根据请求中的 referer 字段来拒绝请求
--without-http_rewrite_module	不安装 http rewrite module。该模块提供 HTTP 请求在 Nginx 服务内部的重定向功能，依赖 PCRE 库
--without-http_proxy_module	不安装 http proxy module。该模块提供基本的 HTTP 反向代理功能
--without-http_fastcgi_module	不安装 http fastcgi module。该模块提供 FastCGI 功能
--without-http_uwsgi_module	不安装 http uwsgi module。该模块提供 uWSGI 功能
--without-http_scgi_module	不安装 http scgi module。该模块提供 SCGI 功能
--without-http_memcached_module	不安装 http memcached module。该模块可以使得 Nginx 直接由上游的 memcached 服务读取数据，并简单地适配成 HTTP 响应返回给客户端
--without-http_limit_zone_module	不安装 http limit zone module。该模块针对某个 IP 地址限制并发连接数。例如，使 Nginx 对一个 IP 地址仅允许一个连接
--without-http_limit_req_module	不安装 http limit req module。该模块针对某个 IP 地址限制并发请求数
--without-http_empty_gif_module	不安装 http empty gif module。该模块可以使得 Nginx 在收到无效请求时，立刻返回内存中的 1×1 像素的 GIF 图片。这种好处在于，对于明显的无效请求不会去试图浪费服务器资源
--without-http_browser_module	不安装 http browser module。该模块会根据 HTTP 请求中的 user-agent 字段（该字段通常由浏览器填写）来识别浏览器
--without-http_upstream_ip_hash_module	不安装 http upstream ip hash module。该模块提供当 Nginx 与后端 server 建立连接时，会根据 IP 做散列运算来决定与后端哪台 server 通信，这样可以实现负载均衡

（3）默认不会编译进入 Nginx 的 HTTP 模块

表 1-11 列出了默认不会编译至 Nginx 中的 HTTP 模块以及把它们加入产品中的方法。

表 1-11　configure 中默认不会编译到 Nginx 中的 HTTP 模块参数

可选的 HTTP 模块	意　义
--with-http_ssl_module	安装 http ssl module。该模块使 Nginx 支持 SSL 协议，提供 HTTPS 服务。 注意：该模块的安装依赖于 OpenSSL 开源软件，即首先应确保已经在之前的参数中配置了 OpenSSL
--with-http_realip_module	安装 http realip module。该模块可以从客户端请求里的 header 信息（如 X-Real-IP 或者 X-Forwarded-For）中获取真正的客户端 IP 地址
--with-http_addition_module	安装 http addtion module。该模块可以在返回客户端的 HTTP 包体头部或者尾部增加内容
--with-http_xslt_module	安装 http xslt module。这个模块可以使 XML 格式的数据在发给客户端前加入 XSL 渲染 注意：这个模块依赖于 libxml2 和 libxslt 库，安装它前首先确保上述两个软件已经安装
--with-http_image_filter_module	安装 http image_filter module。这个模块将符合配置的图片实时压缩为指定大小（width*height）的缩略图再发送给用户，目前支持 JPEG、PNG、GIF 格式。 注意：这个模块依赖于开源的 libgd 库，在安装前确保操作系统已经安装了 libgd
--with-http_geoip_module	安装 http geoip module。该模块可以依据 MaxMind GeoIP 的 IP 地址数据库对客户端的 IP 地址得到实际的地理位置信息 注意：该库依赖于 MaxMind GeoIP 的库文件，可访问 http://geolite.maxmind.com/download/geoip/database/GeoLiteCity.dat.gz 获取
--with-http_sub_module	安装 http sub module。该模块可以在 Nginx 返回客户端的 HTTP 响应包中将指定的字符串替换为自己需要的字符串 例如，在 HTML 的返回中，将 </head> 替换为 </head><script language="javascript" src="$script"></script>
--with-http_dav_module	安装 http dav module。这个模块可以让 Nginx 支持 Webdav 标准，如支持 Webdav 协议中的 PUT、DELETE、COPY、MOVE、MKCOL 等请求
--with-http_flv_module	安装 http flv module。这个模块可以在向客户端返回响应时，对 FLV 格式的视频文件在 header 头做一些处理，使得客户端可以观看、拖动 FLV 视频
--with-http_mp4_module	安装 http mp4 module。该模块使客户端可以观看、拖动 MP4 视频
--with-http_gzip_static_module	安装 http gzip static module。如果采用 gzip 模块把一些文档进行 gzip 格式压缩后再返回给客户端，那么对同一个文件每次都会重新压缩，这是比较消耗服务器 CPU 资源的。gzip static 模块可以在做 gzip 压缩前，先查看相同位置是否有已经做过 gzip 压缩的 .gz 文件，如果有，就直接返回。这样就可以预先在服务器上做好文档的压缩，给 CPU 减负
--with-http_random_index_module	安装 http random index module。该模块在客户端访问某个目录时，随机返回该目录下的任意文件
--with-http_secure_link_module	安装 http secure link module。该模块提供一种验证请求是否有效的机制。例如，它会验证 URL 中需要加入的 token 参数是否属于特定客户端发来的，以及检查时间戳是否过期
--with-http_degradation_module	安装 http degradation module。该模块针对一些特殊的系统调用（如 sbrk）做一些优化，如直接返回 HTTP 响应码为 204 或者 444。目前不支持 Linux 系统
--with-http_stub_status_module	安装 http stub status module。该模块可以让运行中的 Nginx 提供性能统计页面，获取相关的并发连接、请求的信息（14.2.1 节中简单介绍了该模块的原理）
--with-google_perftools_module	安装 google perftools module。该模块提供 Google 的性能测试工具

（4）邮件代理服务器相关的 mail 模块

表 1-12 列出了把邮件模块编译到产品中的参数。

表 1-12 configure 提供的邮件模块参数

可选的 mail 模块	意 义
--with-mail	安装邮件服务器反向代理模块，使 Nginx 可以反向代理 IMAP、POP3、SMTP 等协议。该模块默认不安装
--with-mail_ssl_module	安装 mail ssl module。该模块可以使 IMAP、POP3、SMTP 等协议基于 SSL/TLS 协议之上使用。该模块默认不安装并依赖于 OpenSSL 库
--without-mail_pop3_module	不安装 mail pop3 module。在使用 --with-mail 参数后，pop3 module 是默认安装的，以使 Nginx 支持 POP3 协议
--without-mail_imap_module	不安装 mail imap module。在使用 --with-mail 参数后，imap module 是默认安装的，以使 Nginx 支持 IMAP
--without-mail_smtp_module	不安装 mail smtp module。在使用 --with-mail 参数后，smtp module 是默认安装的，以使 Nginx 支持 SMTP

5. 其他参数

configure 还接收一些其他参数，表 1-13 中列出了相关参数的说明。

表 1-13 configure 提供的其他参数

其他一些参数	意 义
--with-debug	将 Nginx 需要打印 debug 调试级别日志的代码编译进 Nginx。这样可以在 Nginx 运行时通过修改配置文件来使其打印调试日志，这对于研究、定位 Nginx 问题非常有帮助
--add-module=PATH	当在 Nginx 里加入第三方模块时，通过这个参数指定第三方模块的路径。这个参数将在下文如何开发 HTTP 模块时使用到
--without-http	禁用 HTTP 服务器
--without-http-cache	禁用 HTTP 服务器里的缓存 Cache 特性
--with-file-aio	启用文件的异步 I/O 功能来处理磁盘文件，这需要 Linux 内核支持原生的异步 I/O
--with-ipv6	使 Nginx 支持 IPv6
--user=USER	指定 Nginx worker 进程运行时所属的用户 注意：不要将启动 worker 进程的用户设为 root，在 worker 进程出问题时 master 进程要具备停止 / 启动 worker 进程的能力
--group=GROUP	指定 Nginx worker 进程运行时所属的组

1.5.2 configure 执行流程

我们看到 configure 命令支持非常多的参数，读者可能会好奇它在执行时到底做了哪些事情，本节将通过解析 configure 源码来对它有一个感性的认识。configure 由 Shell 脚本编写，中间会调用 <nginx-source>/auto/ 目录下的脚本。这里将只对 configure 脚本本身做分析，对于它所调用的 auto 目录下的其他工具脚本则只做功能性的说明。

configure 脚本的内容如下：

```
#!/bin/sh

# Copyright (C) Igor Sysoev
# Copyright (C) Nginx, Inc.

#auto/options 脚本处理 configure 命令的参数。例如，如果参数是 --help，那么显示支持的所有参数格式。options 脚本会定义后续工作将要用到的变量，然后根据本次参数以及默认值设置这些变量
. auto/options

#auto/init 脚本初始化后续将产生的文件路径。例如，Makefile、ngx_modules.c 等文件默认情况下将会在 <nginx-source>/objs/
. auto/init

#auto/sources 脚本将分析 Nginx 的源码结构，这样才能构造后续的 Makefile 文件
. auto/sources

# 编译过程中所有目标文件生成的路径由—builddir=DIR 参数指定，默认情况下为 <nginx-source>/objs，此时这个目录将会被创建
test -d $NGX_OBJS || mkdir $NGX_OBJS

# 开始准备建立 ngx_auto_headers.h、autoconf.err 等必要的编译文件
echo > $NGX_AUTO_HEADERS_H
echo > $NGX_AUTOCONF_ERR

# 向 objs/ngx_auto_config.h 写入命令行带的参数
echo "#define NGX_CONFIGURE \"$NGX_CONFIGURE\"" > $NGX_AUTO_CONFIG_H

# 判断 DEBUG 标志，如果有，那么在 objs/ngx_auto_config.h 文件中写入 DEBUG 宏
if [ $NGX_DEBUG = YES ]; then
    have=NGX_DEBUG . auto/have
fi

# 现在开始检查操作系统参数是否支持后续编译
if test -z "$NGX_PLATFORM"; then
    echo "checking for OS"

    NGX_SYSTEM=`uname -s 2>/dev/null`
    NGX_RELEASE=`uname -r 2>/dev/null`
    NGX_MACHINE=`uname -m 2>/dev/null`

# 屏幕上输出 OS 名称、内核版本、32 位 /64 位内核
    echo " + $NGX_SYSTEM $NGX_RELEASE $NGX_MACHINE"

    NGX_PLATFORM="$NGX_SYSTEM:$NGX_RELEASE:$NGX_MACHINE";

    case "$NGX_SYSTEM" in
        MINGW32_*)
            NGX_PLATFORM=win32
```

```
        ;;
    esac

else
    echo "building for $NGX_PLATFORM"
    NGX_SYSTEM=$NGX_PLATFORM
fi

# 检查并设置编译器，如 GCC 是否安装、GCC 版本是否支持后续编译 nginx
. auto/cc/conf

# 对非 Windows 操作系统定义一些必要的头文件，并检查其是否存在，以此决定 configure 后续步骤是否可以成功⊖
if [ "$NGX_PLATFORM" != win32 ]; then
    . auto/headers
fi

# 对于当前操作系统，定义一些特定的操作系统相关的方法并检查当前环境是否支持。例如，对于 Linux，在这里使用 sched_setaffinity 设置进程优先级，使用 Linux 特有的 sendfile 系统调用来加速向网络中发送文件块
. auto/os/conf

# 定义类 UNIX 操作系统中通用的头文件和系统调用等，并检查当前环境是否支持
if [ "$NGX_PLATFORM" != win32 ]; then
    . auto/unix
fi

# 最核心的构造运行期 modules 的脚本。它将会生成 ngx_modules.c 文件，这个文件会被编译进 Nginx 中，其中它所做的唯一的事情就是定义了 ngx_modules 数组。ngx_modules 指明 Nginx 运行期间有哪些模块会参与到请求的处理中，包括 HTTP 请求可能会使用哪些模块处理，因此，它对数组元素的顺序非常敏感，也就是说，绝大部分模块在 ngx_modules 数组中的顺序其实是固定的。例如，一个请求必须先执行 ngx_http_gzip_filter_module 模块重新修改 HTTP 响应中的头部后，才能使用 ngx_http_header_filter 模块按照 headers_in 结构体里的成员构造出以 TCP 流形式发送给客户端的 HTTP 响应头部。注意，我们在 --add-module= 参数里加入的第三方模块也在此步骤写入到 ngx_modules.c 文件中了
. auto/modules

#conf 脚本用来检查 Nginx 在链接期间需要链接的第三方静态库、动态库或者目标文件是否存在
. auto/lib/conf

# 处理 Nginx 安装后的路径
case ".$NGX_PREFIX" in
    .)
        NGX_PREFIX=${NGX_PREFIX:-/usr/local/nginx}
        have=NGX_PREFIX value="\"$NGX_PREFIX/\"" . auto/define
    ;;

    .!)
```

⊖ 在 configure 脚本里检查某个特性是否存在时，会生成一个最简单的只包含 main 函数的 C 程序，该程序会包含相应的头文件。然后，通过检查是否可以编译通过来确认特性是否支持，并将结果记录在 objs/autoconf.err 文件中。后续检查头文件、检查特性的脚本都用了类似的方法。

```
            NGX_PREFIX=
        ;;

        *)
            have=NGX_PREFIX value="\"$NGX_PREFIX/\"" . auto/define
        ;;
    esac

    # 处理 Nginx 安装后 conf 文件的路径
    if [ ".$NGX_CONF_PREFIX" != "." ]; then
        have=NGX_CONF_PREFIX value="\"$NGX_CONF_PREFIX/\"" . auto/define
    fi

    # 处理 Nginx 安装后，二进制文件、pid、lock 等其他文件的路径可参见 configure 参数中路径类选项的说明
    have=NGX_SBIN_PATH value="\"$NGX_SBIN_PATH\"" . auto/define
    have=NGX_CONF_PATH value="\"$NGX_CONF_PATH\"" . auto/define
    have=NGX_PID_PATH value="\"$NGX_PID_PATH\"" . auto/define
    have=NGX_LOCK_PATH value="\"$NGX_LOCK_PATH\"" . auto/define
    have=NGX_ERROR_LOG_PATH value="\"$NGX_ERROR_LOG_PATH\"" . auto/define


    have=NGX_HTTP_LOG_PATH value="\"$NGX_HTTP_LOG_PATH\"" . auto/define
    have=NGX_HTTP_CLIENT_TEMP_PATH value="\"$NGX_HTTP_CLIENT_TEMP_PATH\"" . auto/define
    have=NGX_HTTP_PROXY_TEMP_PATH value="\"$NGX_HTTP_PROXY_TEMP_PATH\"" . auto/define
    have=NGX_HTTP_FASTCGI_TEMP_PATH value="\"$NGX_HTTP_FASTCGI_TEMP_PATH\"" . auto/define
    have=NGX_HTTP_UWSGI_TEMP_PATH value="\"$NGX_HTTP_UWSGI_TEMP_PATH\"" . auto/define
    have=NGX_HTTP_SCGI_TEMP_PATH value="\"$NGX_HTTP_SCGI_TEMP_PATH\"" . auto/define

    # 创建编译时使用的 objs/Makefile 文件
    . auto/make

    # 为 objs/Makefile 加入需要连接的第三方静态库、动态库或者目标文件
    . auto/lib/make

    # 为 objs/Makefile 加入 install 功能，当执行 make  install 时将编译生成的必要文件复制到安装路径，
建立必要的目录
    . auto/install

    # 在 ngx_auto_config.h 文件中加入 NGX_SUPPRESS_WARN 宏、NGX_SMP 宏
    . auto/stubs

    # 在 ngx_auto_config.h 文件中指定 NGX_USER 和 NGX_GROUP 宏，如果执行 configure 时没有参数指定，
默认两者皆为 nobody（也就是默认以 nobody 用户运行进程）
    have=NGX_USER value="\"$NGX_USER\"" . auto/define
    have=NGX_GROUP value="\"$NGX_GROUP\"" . auto/define

    # 显示 configure 执行的结果，如果失败，则给出原因
    . auto/summary
```

1.5.3 configure 生成的文件

当 configure 执行成功时会生成 objs 目录，并在该目录下产生以下目录和文件：

```
|---ngx_auto_headers.h
|---autoconf.err
|---ngx_auto_config.h
|---ngx_modules.c
|---src
|    |---core
|    |---event
|    |    |---modules
|    |---os
|    |    |---unix
|    |    |---win32
|    |---http
|    |    |---modules
|    |    |      |---perl
|    |---mail
|    |---misc
|---Makefile
```

上述目录和文件介绍如下。

1）src 目录用于存放编译时产生的目标文件。

2）Makefile 文件用于编译 Nginx 工程以及在加入 install 参数后安装 Nginx。

3）autoconf.err 保存 configure 执行过程中产生的结果。

4）ngx_auto_headers.h 和 ngx_auto_config.h 保存了一些宏，这两个头文件会被 src/core/ngx_config.h 及 src/os/unix/ngx_linux_config.h 文件（可将“linux”替换为其他 UNIX 操作系统）引用。

5）ngx_modules.c 是一个关键文件，我们需要看看它的内部结构。一个默认配置下生成的 ngx_modules.c 文件内容如下：

```
#include <ngx_config.h>
#include <ngx_core.h>

...

ngx_module_t *ngx_modules[] = {
    &ngx_core_module,
    &ngx_errlog_module,
    &ngx_conf_module,
    &ngx_events_module,
    &ngx_event_core_module,
    &ngx_epoll_module,
    &ngx_http_module,
    &ngx_http_core_module,
```

```
    &ngx_http_log_module,
    &ngx_http_upstream_module,
    &ngx_http_static_module,
    &ngx_http_autoindex_module,
    &ngx_http_index_module,
    &ngx_http_auth_basic_module,
    &ngx_http_access_module,
    &ngx_http_limit_zone_module,
    &ngx_http_limit_req_module,
    &ngx_http_geo_module,
    &ngx_http_map_module,
    &ngx_http_split_clients_module,
    &ngx_http_referer_module,
    &ngx_http_rewrite_module,
    &ngx_http_proxy_module,
    &ngx_http_fastcgi_module,
    &ngx_http_uwsgi_module,
    &ngx_http_scgi_module,
    &ngx_http_memcached_module,
    &ngx_http_empty_gif_module,
    &ngx_http_browser_module,
    &ngx_http_upstream_ip_hash_module,
    &ngx_http_write_filter_module,
    &ngx_http_header_filter_module,
    &ngx_http_chunked_filter_module,
    &ngx_http_range_header_filter_module,
    &ngx_http_gzip_filter_module,
    &ngx_http_postpone_filter_module,
    &ngx_http_ssi_filter_module,
    &ngx_http_charset_filter_module,
    &ngx_http_userid_filter_module,
    &ngx_http_headers_filter_module,
    &ngx_http_copy_filter_module,
    &ngx_http_range_body_filter_module,
    &ngx_http_not_modified_filter_module,
    NULL
};
```

ngx_modules.c 文件就是用来定义 ngx_modules 数组的。

ngx_modules 是非常关键的数组，它指明了每个模块在 Nginx 中的优先级，当一个请求同时符合多个模块的处理规则时，将按照它们在 ngx_modules 数组中的顺序选择最靠前的模块优先处理。对于 HTTP 过滤模块而言则是相反的，因为 HTTP 框架在初始化时，会在 ngx_modules 数组中将过滤模块按先后顺序向过滤链表中添加，但每次都是添加到链表的表头，因此，对 HTTP 过滤模块而言，在 ngx_modules 数组中越是靠后的模块反而会首先处理 HTTP 响应（参见第 6 章及第 11 章的 11.9 节）。

因此，ngx_modules 中模块的先后顺序非常重要，不正确的顺序会导致 Nginx 无法工作，这是 auto/modules 脚本执行后的结果。读者可以体会一下上面的 ngx_modules 中同一种类型

下（第 8 章会介绍模块类型，第 10 章、第 11 章将介绍的 HTTP 框架对 HTTP 模块的顺序是最敏感的）各个模块的顺序以及这种顺序带来的意义。

可以看出，在安装过程中，configure 做了大量的幕后工作，我们需要关注在这个过程中 Nginx 做了哪些事情。configure 除了寻找依赖的软件外，还针对不同的 UNIX 操作系统做了许多优化工作。这是 Nginx 跨平台的一种具体实现，也体现了 Nginx 追求高性能的一贯风格。

configure 除了生成 Makefile 外，还生成了 ngx_modules.c 文件，它决定了运行时所有模块的优先级（在编译过程中而不是编码过程中）。对于不需要的模块，既不会加入 ngx_modules 数组，也不会编译进 Nginx 产品中，这也体现了轻量级的概念。

1.6 Nginx 的命令行控制

在 Linux 中，需要使用命令行来控制 Nginx 服务器的启动与停止、重载配置文件、回滚日志文件、平滑升级等行为。默认情况下，Nginx 被安装在目录 /usr/local/nginx/ 中，其二进制文件路径为 /usr/local/nginc/sbin/nginx，配置文件路径为 /usr/local/nginx/conf/nginx.conf。当然，在 configure 执行时是可以指定把它们安装在不同目录的。为了简单起见，本节只说明默认安装情况下的命令行的使用情况，如果读者安装的目录发生了变化，那么替换一下即可。

（1）默认方式启动

直接执行 Nginx 二进制程序。例如：

```
/usr/local/nginx/sbin/nginx
```

这时，会读取默认路径下的配置文件：/usr/local/nginx/conf/nginx.conf。

实际上，在没有显式指定 nginx.conf 配置文件路径时，将打开在 configure 命令执行时使用 --conf-path=PATH 指定的 nginx.conf 文件（参见 1.5.1 节）。

（2）另行指定配置文件的启动方式

使用 -c 参数指定配置文件。例如：

```
/usr/local/nginx/sbin/nginx -c /tmp/nginx.conf
```

这时，会读取 -c 参数后指定的 nginx.conf 配置文件来启动 Nginx。

（3）另行指定安装目录的启动方式

使用 -p 参数指定 Nginx 的安装目录。例如：

```
/usr/local/nginx/sbin/nginx -p /usr/local/nginx/
```

（4）另行指定全局配置项的启动方式

可以通过 -g 参数临时指定一些全局配置项，以使新的配置项生效。例如：

```
/usr/local/nginx/sbin/nginx -g "pid /var/nginx/test.pid;"
```

上面这行命令意味着会把 pid 文件写到 /var/nginx/test.pid 中。

-g 参数的约束条件是指定的配置项不能与默认路径下的 nginx.conf 中的配置项相冲突，否则无法启动。就像上例那样，类似这样的配置项：pid logs/nginx.pid，是不能存在于默认的 nginx.conf 中的。

另一个约束条件是，以 -g 方式启动的 Nginx 服务执行其他命令行时，需要把 -g 参数也带上，否则可能出现配置项不匹配的情形。例如，如果要停止 Nginx 服务，那么需要执行下面代码：

```
/usr/local/nginx/sbin/nginx -g "pid /var/nginx/test.pid;" -s stop
```

如果不带上 -g "pid /var/nginx/test.pid;"，那么找不到 pid 文件，也会出现无法停止服务的情况。

（5）测试配置信息是否有错误

在不启动 Nginx 的情况下，使用 -t 参数仅测试配置文件是否有错误。例如：

```
/usr/local/nginx/sbin/nginx -t
```

执行结果中显示配置是否正确。

（6）在测试配置阶段不输出信息

测试配置选项时，使用 -q 参数可以不把 error 级别以下的信息输出到屏幕。例如：

```
/usr/local/nginx/sbin/nginx -t -q
```

（7）显示版本信息

使用 -v 参数显示 Nginx 的版本信息。例如：

```
/usr/local/nginx/sbin/nginx -v
```

（8）显示编译阶段的参数

使用 -V 参数除了可以显示 Nginx 的版本信息外，还可以显示配置编译阶段的信息，如 GCC 编译器的版本、操作系统的版本、执行 configure 时的参数等。例如：

```
/usr/local/nginx/sbin/nginx -V
```

（9）快速地停止服务

使用 -s stop 可以强制停止 Nginx 服务。-s 参数其实是告诉 Nginx 程序向正在运行的 Nginx 服务发送信号量，Nginx 程序通过 nginx.pid 文件中得到 master 进程的进程 ID，再向运行中的 master 进程发送 TERM 信号来快速地关闭 Nginx 服务。例如：

```
/usr/local/nginx/sbin/nginx -s stop
```

实际上，如果通过 kill 命令直接向 nginx master 进程发送 TERM 或者 INT 信号，效果是

一样的。例如，先通过 ps 命令来查看 nginx master 的进程 ID：

```
:ahf5wapi001:root > ps -ef | grep nginx
root     10800     1  0 02:27 ?        00:00:00 nginx: master process ./nginx
root     10801 10800  0 02:27 ?        00:00:00 nginx: worker process
```

接下来直接通过 kill 命令来发送信号：

```
kill -s SIGTERM 10800
```

或者：

```
kill -s SIGINT 10800
```

上述两条命令的效果与执行 /usr/local/nginx/sbin/nginx -s stop 是完全一样的。

（10）“优雅”地停止服务

如果希望 Nginx 服务可以正常地处理完当前所有请求再停止服务，那么可以使用 -s quit 参数来停止服务。例如：

```
/usr/local/nginx/sbin/nginx -s quit
```

该命令与快速停止 Nginx 服务是有区别的。当快速停止服务时，worker 进程与 master 进程在收到信号后会立刻跳出循环，退出进程。而“优雅”地停止服务时，首先会关闭监听端口，停止接收新的连接，然后把当前正在处理的连接全部处理完，最后再退出进程。

与快速停止服务相似，可以直接发送 QUIT 信号给 master 进程来停止服务，其效果与执行 -s quit 命令是一样的。例如：

```
kill -s SIGQUIT <nginx master pid>
```

如果希望“优雅”地停止某个 worker 进程，那么可以通过向该进程发送 WINCH 信号来停止服务。例如：

```
kill -s SIGWINCH <nginx worker pid>
```

（11）使运行中的 Nginx 重读配置项并生效

使用 -s reload 参数可以使运行中的 Nginx 服务重新加载 nginx.conf 文件。例如：

```
/usr/local/nginx/sbin/nginx -s reload
```

事实上，Nginx 会先检查新的配置项是否有误，如果全部正确就以“优雅”的方式关闭，再重新启动 Nginx 来实现这个目的。类似的，-s 是发送信号，仍然可以用 kill 命令发送 HUP 信号来达到相同的效果。

```
kill -s SIGHUP <nginx master pid>
```

（12）日志文件回滚

使用 -s reopen 参数可以重新打开日志文件，这样可以先把当前日志文件改名或转移到其他目录中进行备份，再重新打开时就会生成新的日志文件。这个功能使得日志文件不至于过大。例如：

```
/usr/local/nginx/sbin/nginx -s reopen
```

当然，这与使用 kill 命令发送 USR1 信号效果相同。

```
kill -s SIGUSR1 <nginx master pid>
```

（13）平滑升级 Nginx

当 Nginx 服务升级到新的版本时，必须要将旧的二进制文件 Nginx 替换掉，通常情况下这是需要重启服务的，但 Nginx 支持不重启服务来完成新版本的平滑升级。

升级时包括以下步骤：

1）通知正在运行的旧版本 Nginx 准备升级。通过向 master 进程发送 USR2 信号可达到目的。例如：

```
kill -s SIGUSR2 <nginx master pid>
```

这时，运行中的 Nginx 会将 pid 文件重命名，如将 /usr/local/nginx/logs/nginx.pid 重命名为 /usr/local/nginx/logs/nginx.pid.oldbin，这样新的 Nginx 才有可能启动成功。

2）启动新版本的 Nginx，可以使用以上介绍过的任意一种启动方法。这时通过 ps 命令可以发现新旧版本的 Nginx 在同时运行。

3）通过 kill 命令向旧版本的 master 进程发送 SIGQUIT 信号，以“优雅”的方式关闭旧版本的 Nginx。随后将只有新版本的 Nginx 服务运行，此时平滑升级完毕。

（14）显示命令行帮助

使用 -h 或者 -? 参数会显示支持的所有命令行参数。

1.7　小结

本章介绍了 Nginx 的特点以及在什么场景下需要使用 Nginx，同时介绍了如何获取 Nginx 以及如何配置、编译、安装运行 Nginx。本章还深入介绍了最为复杂的 configure 过程，这部分内容是学习本书第二部分和第三部分的基础。

第 2 章　Nginx 的配置

Nginx 拥有大量官方发布的模块和第三方模块，这些已有的模块可以帮助我们实现 Web 服务器上很多的功能。使用这些模块时，仅仅需要增加、修改一些配置项即可。因此，本章的目的是熟悉 Nginx 的配置文件，包括配置文件的语法格式、运行所有 Nginx 服务必须具备的基础配置以及使用 HTTP 核心模块配置静态 Web 服务器的方法，最后还会介绍反向代理服务器。

通过本章的学习，读者可以：熟练地配置一个静态 Web 服务器；对影响 Web 服务器性能的各个配置项有深入的理解；对配置语法有全面的了解。通过互联网或其他途径得到任意模块的配置说明，然后可通过修改 nginx.conf 文件来使用这些模块的功能。

2.1　运行中的 Nginx 进程间的关系

在正式提供服务的产品环境下，部署 Nginx 时都是使用一个 master 进程来管理多个 worker 进程，一般情况下，worker 进程的数量与服务器上的 CPU 核心数相等。每一个 worker 进程都是繁忙的，它们在真正地提供互联网服务，master 进程则很“清闲”，只负责监控管理 worker 进程。worker 进程之间通过共享内存、原子操作等一些进程间通信机制来实现负载均衡等功能（第 9 章将会介绍负载均衡机制，第 14 章将会介绍负载均衡锁的实现）。

部署后 Nginx 进程间的关系如图 2-1 所示。

Nginx 是支持单进程（master 进程）提供服务的，那么为什么产品环境下要按照 master-worker 方式配置同时启动多个进程呢？这样做的好处主要有以下两点：

- 由于 master 进程不会对用户请求提供服务，只用于管理真正提供服务的 worker 进程，所以 master 进程可以是唯一的，它仅专注于自己的纯管理工作，为管理员提供命令行服务，包括诸如启动服务、停止服务、重载配置文件、平滑升级程序等。master 进程需要拥有较大的权限，例如，通常会利用 root 用户启动 master 进程。worker 进程的权限要小于或等于 master 进程，这样 master 进程才可以完全地管理 worker 进程。当任意一个 worker 进程出现错误从而导致 coredump 时，master 进程会立刻启动新的 worker 进程继续服务。
- 多个 worker 进程处理互联网请求不但可以提高服务的健壮性（一个 worker 进程出错后，其他 worker 进程仍然可以正常提供服务），最重要的是，这样可以充分利用现

在常见的 SMP 多核架构，从而实现微观上真正的多核并发处理。因此，用一个进程（master 进程）来处理互联网请求肯定是不合适的。另外，为什么要把 worker 进程数量设置得与 CPU 核心数量一致呢？这正是 Nginx 与 Apache 服务器的不同之处。在 Apache 上每个进程在一个时刻只处理一个请求，因此，如果希望 Web 服务器拥有并发处理的请求数更多，就要把 Apache 的进程或线程数设置得更多，通常会达到一台服务器拥有几百个工作进程，这样大量的进程间切换将带来无谓的系统资源消耗。而 Nginx 则不然，一个 worker 进程可以同时处理的请求数只受限于内存大小，而且在架构设计上，不同的 worker 进程之间处理并发请求时几乎没有同步锁的限制，worker 进程通常不会进入睡眠状态，因此，当 Nginx 上的进程数与 CPU 核心数相等时（最好每一个 worker 进程都绑定特定的 CPU 核心），进程间切换的代价是最小的。

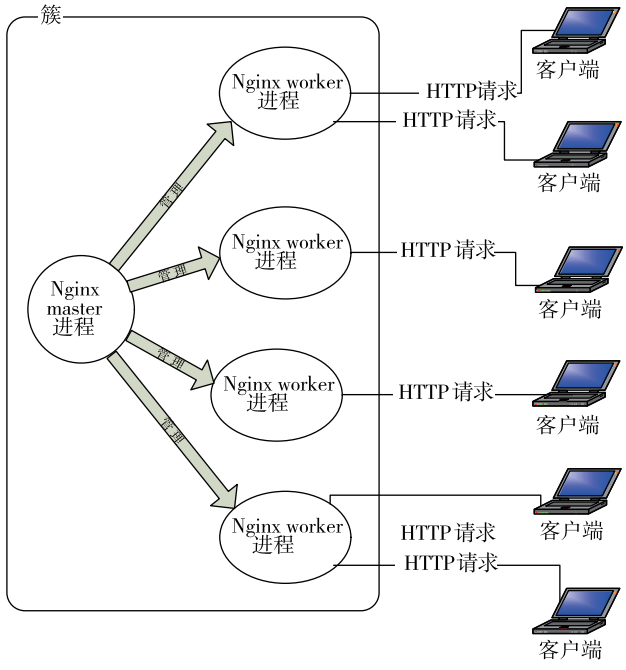


图 2-1　部署后 Nginx 进程间的关系

举例来说，如果产品中的服务器 CPU 核心数为 8，那么就需要配置 8 个 worker 进程（见图 2-2）。

如果对路径部分都使用默认配置，那么 Nginx 运行目录为 /usr/local/nginx，其目录结构如下。

```
|---sbin
|    |---nginx
```

```
|---conf
|     |---koi-win
|     |---koi-utf
|     |---win-utf
|     |---mime.types
|     |---mime.types.default
|     |---fastcgi_params
|     |---fastcgi_params.default
|     |---fastcgi.conf
|     |---fastcgi.conf.default
|     |---uwsgi_params
|     |---uwsgi_params.default
|     |---scgi_params
|     |---scgi_params.default
|     |---nginx.conf
|     |---nginx.conf.default
|---logs
|     |---error.log
|     |---access.log
|     |---nginx.pid
|---html
|     |---50x.html
|     |---index.html
|---client_body_temp
|---proxy_temp
|---fastcgi_temp
|---uwsgi_temp
|---scgi_temp
```

```
top - 01:52:31 up 28 days, 14:34,  2 users,  load average: 0.00, 0.00, 0.00
Tasks:   9 total,   0 running,   9 sleeping,   0 stopped,   0 zombie
Cpu0  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :  0.3%us,  0.0%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu3  :  0.3%us,  0.0%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu4  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu5  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu6  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu7  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   2059504k total,  2045548k used,    13956k free,   127652k buffers
Swap:  4008176k total,     1344k used,  4006832k free,  1518872k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 6984 root      17   0  276m 4832 3792 S  0.0  0.2   0:27.15 nginx: master process
 8394 nobody    15   0  279m 4836  600 S  0.0  0.2   0:00.01 nginx: worker process
 8395 nobody    15   0  279m 4836  600 S  0.0  0.2   0:00.01 nginx: worker process
 8396 nobody    17   0  279m 4824  588 S  0.0  0.2   0:00.01 nginx: worker process
 8397 nobody    15   0  279m 4836  600 S  0.0  0.2   0:00.00 nginx: worker process
 8398 nobody    15   0  279m 4836  600 S  0.0  0.2   0:00.01 nginx: worker process
 8399 nobody    16   0  279m 4836  600 S  0.0  0.2   0:00.00 nginx: worker process
 8400 nobody    15   0  279m 4836  600 S  0.0  0.2   0:00.01 nginx: worker process
 8401 nobody    15   0  279m 4836  600 S  0.0  0.2   0:00.00 nginx: worker process
```

图 2-2　worker 进程的数量尽量与 CPU 核心数相等

2.2　Nginx 配置的通用语法

Nginx 的配置文件其实是一个普通的文本文件。下面来看一个简单的例子。

```
user  nobody;

worker_processes  8;
error_log  /var/log/nginx/error.log error;

#pid        logs/nginx.pid;

events {
    use epoll;
    worker_connections  50000;
}

http {
    include       mime.types;
    default_type  application/octet-stream;

    log_format  main  '$remote_addr [$time_local] "$request" '
                      '$status $bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';

    access_log  logs/access.log  main buffer=32k;

   ...
}
```

在这段简短的配置代码中，每一行配置项的语法格式都将在 2.2.2 节介绍，出现的 events 和 http 块配置项将在 2.2.1 节介绍，以 # 符号开头的注释将在 2.2.3 节介绍，类似“buffer=32k”这样的配置项的单位将在 2.2.4 节介绍。

2.2.1　块配置项

块配置项由一个块配置项名和一对大括号组成。具体示例如下：

```
events {
...
}

http {
    upstream backend {
           server 127.0.0.1:8080;
    }

    gzip on;
    server {
```

```
            ...
            location /webstatic {
                  gzip off;
            }
      }
}
```

上面代码段中的 events、http、server、location、upstream 等都是块配置项，块配置项之后是否如“location /webstatic {...}”那样在后面加上参数，取决于解析这个块配置项的模块，不能一概而论，但块配置项一定会用大括号把一系列所属的配置项全包含进来，表示大括号内的配置项同时生效。所有的事件类配置都要在 events 块中，http、server 等配置也遵循这个规定。

块配置项可以嵌套。内层块直接继承外层块，例如，上例中，server 块里的任意配置都是基于 http 块里的已有配置的。当内外层块中的配置发生冲突时，究竟是以内层块还是外层块的配置为准，取决于解析这个配置项的模块，第 4 章将会介绍 http 块内配置项冲突的处理方法。例如，上例在 http 模块中已经打开了“gzip on;”，但其下的 location/webstatic 又把 gzip 关闭了：gzip off;，最终，在 /webstatic 的处理模块中，gzip 模块是按照 gzip off 来处理请求的。

2.2.2 配置项的语法格式

从上文的示例可以看出，最基本的配置项语法格式如下：

```
配置项名 配置项值 1 配置项值 2 … ;
```

下面解释一下配置项的构成部分。

首先，在行首的是配置项名，这些配置项名必须是 Nginx 的某一个模块想要处理的，否则 Nginx 会认为配置文件出现了非法的配置项名。配置项名输入结束后，将以空格作为分隔符。

其次是配置项值，它可以是数字或字符串（当然也包括正则表达式）。针对一个配置项，既可以只有一个值，也可以包含多个值，配置项值之间仍然由空格符来分隔。当然，一个配置项对应的值究竟有多少个，取决于解析这个配置项的模块。我们必须根据某个 Nginx 模块对一个配置项的约定来更改配置项，第 4 章将会介绍模块是如何约定一个配置项的格式。

最后，每行配置的结尾需要加上分号。

注意 如果配置项值中包括语法符号，比如空格符，那么需要使用单引号或双引号括住配置项值，否则 Nginx 会报语法错误。例如：

```
log_format  main  '$remote_addr - $remote_user [$time_local] "$request" ';
```

2.2.3　配置项的注释

如果有一个配置项暂时需要注释掉，那么可以加“#”注释掉这一行配置。例如：

```
#pid        logs/nginx.pid;
```

2.2.4　配置项的单位

大部分模块遵循一些通用的规定，如指定空间大小时不用每次都定义到字节、指定时间时不用精确到毫秒。

当指定空间大小时，可以使用的单位包括：

- K 或者 k 千字节（KiloByte，KB）。
- M 或者 m 兆字节（MegaByte，MB）。

例如：

```
gzip_buffers     4 8k;
client_max_body_size 64M;
```

当指定时间时，可以使用的单位包括：

- ms（毫秒），s（秒），m（分钟），h（小时），d（天），w（周，包含 7 天），M（月，包含 30 天），y（年，包含 365 天）。

例如：

```
expires    10y;
proxy_read_timeout600;
client_body_timeout       2m;
```

注意　配置项后的值究竟是否可以使用这些单位，取决于解析该配置项的模块。如果这个模块使用了 Nginx 框架提供的相应解析配置项方法，那么配置项值才可以携带单位。第 4 章中详细描述了 Nginx 框架提供的 14 种预设解析方法，其中一些方法将可以解析以上列出的单位。

2.2.5　在配置中使用变量

有些模块允许在配置项中使用变量，如在日志记录部分，具体示例如下。

```
log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';
```

其中，remote_addr 是一个变量，使用它的时候前面要加上 $ 符号。需要注意的是，这种变量只有少数模块支持，并不是通用的。

许多模块在解析请求时都会提供多个变量（如本章后面提到的 http core module、http proxy module、http upstream module 等），以使其他模块的配置可以即时使用。我们在学习某个模块提供的配置说明时可以关注它是否提供变量。

提示 在执行 configure 命令时，我们已经把许多模块编译进 Nginx 中，但是否启用这些模块，一般取决于配置文件中相应的配置项。换句话说，每个 Nginx 模块都有自己感兴趣的配置项，大部分模块都必须在 nginx.conf 中读取某个配置项后才会在运行时启用。例如，只有当配置 http {...} 这个配置项时，ngx_http_module 模块才会在 Nginx 中启用，其他依赖 ngx_http_module 的模块也才能正常使用。

2.3 Nginx 服务的基本配置

Nginx 在运行时，至少必须加载几个核心模块和一个事件类模块。这些模块运行时所支持的配置项称为基本配置——所有其他模块执行时都依赖的配置项。

下面详述基本配置项的用法。由于配置项较多，所以把它们按照用户使用时的预期功能分成了以下 4 类：

- ❑ 用于调试、定位问题的配置项。
- ❑ 正常运行的必备配置项。
- ❑ 优化性能的配置项。
- ❑ 事件类配置项（有些事件类配置项归纳到优化性能类，这是因为它们虽然也属于 events {} 块，但作用是优化性能）。

有这么一些配置项，即使没有显式地进行配置，它们也会有默认的值，如 daemon，即使在 nginx.conf 中没有对它进行配置，也相当于打开了这个功能，这点需要注意。对于这样的配置项，作者会在下面相应的配置项描述上加入一行“默认：”来进行说明。

2.3.1 用于调试进程和定位问题的配置项

先来看一下用于调试进程、定位问题的配置项，如下所示。

（1）是否以守护进程方式运行 Nginx

语法：daemon on | off;

默认：daemon on;

守护进程（daemon）是脱离终端并且在后台运行的进程。它脱离终端是为了避免进程执行过程中的信息在任何终端上显示，这样一来，进程也不会被任何终端所产生的信息所打断。Nginx 毫无疑问是一个需要以守护进程方式运行的服务，因此，默认都是以这种方式运

行的。

不过 Nginx 还是提供了关闭守护进程的模式，之所以提供这种模式，是为了方便跟踪调试 Nginx，毕竟用 gdb 调试进程时最烦琐的就是如何继续跟进 fork 出的子进程了。这在第三部分研究 Nginx 架构时很有用。

（2）是否以 master/worker 方式工作

语法：master_process on | off;

默认：master_process on;

可以看到，在如图 2-1 所示的产品环境中，是以一个 master 进程管理多个 worker 进程的方式运行的，几乎所有的产品环境下，Nginx 都以这种方式工作。

与 daemon 配置相同，提供 master_process 配置也是为了方便跟踪调试 Nginx。如果用 off 关闭了 master_process 方式，就不会 fork 出 worker 子进程来处理请求，而是用 master 进程自身来处理请求。

（3）error 日志的设置

语法：error_log /path/file level;

默认：error_log logs/error.log error;

error 日志是定位 Nginx 问题的最佳工具，我们可以根据自己的需求妥善设置 error 日志的路径和级别。

/path/file 参数可以是一个具体的文件，例如，默认情况下是 logs/error.log 文件，最好将它放到一个磁盘空间足够大的位置；/path/file 也可以是 /dev/null，这样就不会输出任何日志了，这也是关闭 error 日志的唯一手段；/path/file 也可以是 stderr，这样日志会输出到标准错误文件中。

level 是日志的输出级别，取值范围是 debug、info、notice、warn、error、crit、alert、emerg，从左至右级别依次增大。当设定为一个级别时，大于或等于该级别的日志都会被输出到 /path/file 文件中，小于该级别的日志则不会输出。例如，当设定为 error 级别时，error、crit、alert、emerg 级别的日志都会输出。

如果设定的日志级别是 debug，则会输出所有的日志，这样数据量会很大，需要预先确保 /path/file 所在磁盘有足够的磁盘空间。

注意　如果日志级别设定到 debug，必须在 configure 时加入 --with-debug 配置项。

（4）是否处理几个特殊的调试点

语法：debug_points [stop | abort]

这个配置项也是用来帮助用户跟踪调试 Nginx 的。它接受两个参数：stop 和 abort。Nginx 在一些关键的错误逻辑中（Nginx 1.0.14 版本中有 8 处）设置了调试点。如果设置了 debug_points 为 stop，那么 Nginx 的代码执行到这些调试点时就会发出 SIGSTOP 信号以用

于调试。如果 debug_points 设置为 abort，则会产生一个 coredump 文件，可以使用 gdb 来查看 Nginx 当时的各种信息。

通常不会使用这个配置项。

（5）仅对指定的客户端输出 debug 级别的日志

语法：debug_connection [IP | CIDR]

这个配置项实际上属于事件类配置，因此，它必须放在 events {...} 中才有效。它的值可以是 IP 地址或 CIDR 地址，例如：

```
events {
    debug_connection 10.224.66.14;
    debug_connection 10.224.57.0/24;
}
```

这样，仅仅来自以上 IP 地址的请求才会输出 debug 级别的日志，其他请求仍然沿用 error_log 中配置的日志级别。

上面这个配置对修复 Bug 很有用，特别是定位高并发请求下才会发生的问题。

注意 使用 debug_connection 前，需确保在执行 configure 时已经加入了 --with-debug 参数，否则不会生效。

（6）限制 coredump 核心转储文件的大小

语法：worker_rlimit_core size;

在 Linux 系统中，当进程发生错误或收到信号而终止时，系统会将进程执行时的内存内容（核心映像）写入一个文件（core 文件），以作为调试之用，这就是所谓的核心转储（core dumps）。当 Nginx 进程出现一些非法操作（如内存越界）导致进程直接被操作系统强制结束时，会生成核心转储 core 文件，可以从 core 文件获取当时的堆栈、寄存器等信息，从而帮助我们定位问题。但这种 core 文件中的许多信息不一定是用户需要的，如果不加以限制，那么可能一个 core 文件会达到几 GB，这样随便 coredumps 几次就会把磁盘占满，引发严重问题。通过 worker_rlimit_core 配置可以限制 core 文件的大小，从而有效帮助用户定位问题。

（7）指定 coredump 文件生成目录

语法：working_directory path;

worker 进程的工作目录。这个配置项的唯一用途就是设置 coredump 文件所放置的目录，协助定位问题。因此，需确保 worker 进程有权限向 working_directory 指定的目录中写入文件。

2.3.2 正常运行的配置项

下面是正常运行的配置项的相关介绍。

（1）定义环境变量

语法：env VAR|VAR=VALUE

这个配置项可以让用户直接设置操作系统上的环境变量。例如：

```
env TESTPATH=/tmp/;
```

（2）嵌入其他配置文件

语法：include /path/file;

include 配置项可以将其他配置文件嵌入到当前的 nginx.conf 文件中，它的参数既可以是绝对路径，也可以是相对路径（相对于 Nginx 的配置目录，即 nginx.conf 所在的目录），例如：

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

可以看到，参数的值可以是一个明确的文件名，也可以是含有通配符 * 的文件名，同时可以一次嵌入多个配置文件。

（3）pid 文件的路径

语法：pid path/file;

默认：pid logs/nginx.pid;

保存 master 进程 ID 的 pid 文件存放路径。默认与 configure 执行时的参数“--pid-path”所指定的路径是相同的，也可以随时修改，但应确保 Nginx 有权在相应的目标中创建 pid 文件，该文件直接影响 Nginx 是否可以运行。

（4）Nginx worker 进程运行的用户及用户组

语法：user username [groupname];

默认：user nobody nobody;

user 用于设置 master 进程启动后，fork 出的 worker 进程运行在哪个用户和用户组下。当按照“user username;”设置时，用户组名与用户名相同。

若用户在 configure 命令执行时使用了参数 --user=username 和 --group=groupname，此时 nginx.conf 将使用参数中指定的用户和用户组。

（5）指定 Nginx worker 进程可以打开的最大句柄描述符个数

语法：worker_rlimit_nofile limit;

设置一个 worker 进程可以打开的最大文件句柄数。

（6）限制信号队列

语法：worker_rlimit_sigpending limit;

设置每个用户发往 Nginx 的信号队列的大小。也就是说，当某个用户的信号队列满了，这个用户再发送的信号量会被丢掉。

2.3.3 优化性能的配置项

下面是优化性能的配置项的相关介绍。

（1）Nginx worker 进程个数

语法：worker_processes number;

默认：worker_processes 1;

在 master/worker 运行方式下，定义 worker 进程的个数。

worker 进程的数量会直接影响性能。那么，用户配置多少个 worker 进程才好呢？这实际上与业务需求有关。

每个 worker 进程都是单线程的进程，它们会调用各个模块以实现多种多样的功能。如果这些模块确认不会出现阻塞式的调用，那么，有多少 CPU 内核就应该配置多少个进程；反之，如果有可能出现阻塞式调用，那么需要配置稍多一些的 worker 进程。

例如，如果业务方面会致使用户请求大量读取本地磁盘上的静态资源文件，而且服务器上的内存较小，以至于大部分的请求访问静态资源文件时都必须读取磁盘（磁头的寻址是缓慢的），而不是内存中的磁盘缓存，那么磁盘 I/O 调用可能会阻塞住 worker 进程少量时间，进而导致服务整体性能下降。

多 worker 进程可以充分利用多核系统架构，但若 worker 进程的数量多于 CPU 内核数，那么会增大进程间切换带来的消耗（Linux 是抢占式内核）。一般情况下，用户要配置与 CPU 内核数相等的 worker 进程，并且使用下面的 worker_cpu_affinity 配置来绑定 CPU 内核。

（2）绑定 Nginx worker 进程到指定的 CPU 内核

语法：worker_cpu_affinity cpumask [cpumask...]

为什么要绑定 worker 进程到指定的 CPU 内核呢？假定每一个 worker 进程都是非常繁忙的，如果多个 worker 进程都在抢同一个 CPU，那么这就会出现同步问题。反之，如果每一个 worker 进程都独享一个 CPU，就在内核的调度策略上实现了完全的并发。

例如，如果有 4 颗 CPU 内核，就可以进行如下配置：

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

注意 worker_cpu_affinity 配置仅对 Linux 操作系统有效。Linux 操作系统使用 sched_setaffinity() 系统调用实现这个功能。

（3）SSL 硬件加速

语法：ssl_engine device;

如果服务器上有 SSL 硬件加速设备，那么就可以进行配置以加快 SSL 协议的处理速度。用户可以使用 OpenSSL 提供的命令来查看是否有 SSL 硬件加速设备：

```
openssl engine -t
```

（4）系统调用 gettimeofday 的执行频率

语法：timer_resolution t;

默认情况下，每次内核的事件调用（如 epoll、select、poll、kqueue 等）返回时，都会执行一次 gettimeofday，实现用内核的时钟来更新 Nginx 中的缓存时钟。在早期的 Linux 内核中，gettimeofday 的执行代价不小，因为中间有一次内核态到用户态的内存复制。当需要降低 gettimeofday 的调用频率时，可以使用 timer_resolution 配置。例如，“timer_resolution 100ms；”表示至少每 100ms 才调用一次 gettimeofday。

但在目前的大多数内核中，如 x86-64 体系架构，gettimeofday 只是一次 vsyscall，仅仅对共享内存页中的数据做访问，并不是通常的系统调用，代价并不大，一般不必使用这个配置。而且，如果希望日志文件中每行打印的时间更准确，也可以使用它。

（5）Nginx worker 进程优先级设置

语法：worker_priority nice;

默认：worker_priority 0;

该配置项用于设置 Nginx worker 进程的 nice 优先级。

在 Linux 或其他类 UNIX 操作系统中，当许多进程都处于可执行状态时，将按照所有进程的优先级来决定本次内核选择哪一个进程执行。进程所分配的 CPU 时间片大小也与进程优先级相关，优先级越高，进程分配到的时间片也就越大（例如，在默认配置下，最小的时间片只有 5ms，最大的时间片则有 800ms）。这样，优先级高的进程会占有更多的系统资源。

优先级由静态优先级和内核根据进程执行情况所做的动态调整（目前只有 ±5 的调整）共同决定。nice 值是进程的静态优先级，它的取值范围是 –20 ～ +19，–20 是最高优先级，+19 是最低优先级。因此，如果用户希望 Nginx 占有更多的系统资源，那么可以把 nice 值配置得更小一些，但不建议比内核进程的 nice 值（通常为 –5）还要小。

2.3.4　事件类配置项

下面是事件类配置项的相关介绍。

（1）是否打开 accept 锁

语法：accept_mutex [on | off]

默认：accept_mutext on;

accept_mutex 是 Nginx 的负载均衡锁，本书会在第 9 章事件处理框架中详述 Nginx 是如何实现负载均衡的。这里，读者仅需要知道 accept_mutex 这把锁可以让多个 worker 进程轮流地、序列化地与新的客户端建立 TCP 连接。当某一个 worker 进程建立的连接数量达到 worker_connections 配置的最大连接数的 7/8 时，会大大地减小该 worker 进程试图建立新 TCP 连接的机会，以此实现所有 worker 进程之上处理的客户端请求数尽量接近。

accept 锁默认是打开的，如果关闭它，那么建立 TCP 连接的耗时会更短，但 worker 进程之间的负载会非常不均衡，因此不建议关闭它。

（2）lock 文件的路径

语法：lock_file path/file;

默认：lock_file logs/nginx.lock;

accept 锁可能需要这个 lock 文件，如果 accept 锁关闭，lock_file 配置完全不生效。如果打开了 accept 锁，并且由于编译程序、操作系统架构等因素导致 Nginx 不支持原子锁，这时才会用文件锁实现 accept 锁（14.8.1 节将会介绍文件锁的用法），这样 lock_file 指定的 lock 文件才会生效。

注意 在基于 i386、AMD64、Sparc64、PPC64 体系架构的操作系统上，若使用 GCC、Intel C++ 、SunPro C++ 编译器来编译 Nginx，则可以肯定这时的 Nginx 是支持原子锁的，因为 Nginx 会利用 CPU 的特性并用汇编语言来实现它（可以参考 14.3 节 x86 架构下原子操作的实现）。这时的 lock_file 配置是没有意义的。

（3）使用 accept 锁后到真正建立连接之间的延迟时间

语法：accept_mutex_delay Nms;

默认：accept_mutex_delay 500ms;

在使用 accept 锁后，同一时间只有一个 worker 进程能够取到 accept 锁。这个 accept 锁不是阻塞锁，如果取不到会立刻返回。如果有一个 worker 进程试图取 accept 锁而没有取到，它至少要等 accept_mutex_delay 定义的时间间隔后才能再次试图取锁。

（4）批量建立新连接

语法：multi_accept [on | off];

默认：multi_accept off;

当事件模型通知有新连接时，尽可能地对本次调度中客户端发起的所有 TCP 请求都建立连接。

（5）选择事件模型

语法：use [kqueue | rtsig | epoll | /dev/poll | select | poll | eventport];

默认：Nginx 会自动使用最适合的事件模型。

对于 Linux 操作系统来说，可供选择的事件驱动模型有 poll、select、epoll 三种。epoll 当然是性能最高的一种，在 9.6 节会解释 epoll 为什么可以处理大并发连接。

（6）每个 worker 的最大连接数

语法：worker_connections number;

定义每个 worker 进程可以同时处理的最大连接数。

2.4　用 HTTP 核心模块配置一个静态 Web 服务器

静态 Web 服务器的主要功能由 ngx_http_core_module 模块（HTTP 框架的主要成员）实现，当然，一个完整的静态 Web 服务器还有许多功能是由其他的 HTTP 模块实现的。本节主要讨论如何配置一个包含基本功能的静态 Web 服务器，文中会完整地说明 ngx_http_core_module 模块提供的配置项及变量的用法，但不会过多说明其他 HTTP 模块的配置项。在阅读完本节内容后，读者应当可以通过简单的查询相关模块（如 ngx_http_gzip_filter_module、ngx_http_image_filter_module 等）的配置项说明，方便地在 nginx.conf 配置文件中加入新的配置项，从而实现更多的 Web 服务器功能。

除了 2.3 节提到的基本配置项外，一个典型的静态 Web 服务器还会包含多个 server 块和 location 块，例如：

```
http {
    gzip on;

    upstream {
           …
    }
    …
    server {
           listen localhost:80;
           …
           location /webstatic {
                  if … {
                         …
                  }
                  root /opt/webresource;
                  …
           }
           location ~* .(jpg|jpeg|png|jpe|gif)$ {
                  …
           }
    }
    server {
           …
    }
}
```

所有的 HTTP 配置项都必须直属于 http 块、server 块、location 块、upstream 块或 if 块等（HTTP 配置项自然必须全部在 http{} 块之内，这里的“直属于”是指配置项直接所属的大括号对应的配置块），同时，在描述每个配置项的功能时，会说明它可以在上述的哪个块中存在，因为有些配置项可以任意地出现在某一个块中，而有些配置项只能出现在特定的块中，在第 4 章介绍自定义配置项的读取时，相信读者就会体会到这种设计思路。

Nginx 为配置一个完整的静态 Web 服务器提供了非常多的功能，下面会把这些配置项分为以下 8 类进行详述：虚拟主机与请求的分发、文件路径的定义、内存及磁盘资源的分配、网络连接的设置、MIME 类型的设置、对客户端请求的限制、文件操作的优化、对客户端请求的特殊处理。这种划分只是为了帮助大家从功能上理解这些配置项。

在这之后会列出 ngx_http_core_module 模块提供的变量，以及简单说明它们的意义。

2.4.1 虚拟主机与请求的分发

由于 IP 地址的数量有限，因此经常存在多个主机域名对应着同一个 IP 地址的情况，这时在 nginx.conf 中就可以按照 server_name（对应用户请求中的主机域名）并通过 server 块来定义虚拟主机，每个 server 块就是一个虚拟主机，它只处理与之相对应的主机域名请求。这样，一台服务器上的 Nginx 就能以不同的方式处理访问不同主机域名的 HTTP 请求了。

（1）监听端口

语法：listen address:port [default(deprecated in 0.8.21) | default_server | [backlog=num | rcvbuf=size | sndbuf=size | accept_filter=filter | deferred | bind | ipv6only=[on|off] | ssl]];

默认：listen 80;

配置块：server

listen 参数决定 Nginx 服务如何监听端口。在 listen 后可以只加 IP 地址、端口或主机名，非常灵活，例如：

```
listen 127.0.0.1:8000;
listen 127.0.0.1; # 注意：不加端口时，默认监听 80 端口
listen 8000;
listen *:8000;
listen localhost:8000;
```

如果服务器使用 IPv6 地址，那么可以这样使用：

```
listen [::]:8000;
listen [fe80::1];
listen [:::a8c9:1234]:80;
```

在地址和端口后，还可以加上其他参数，例如：

```
listen  443 default_server ssl;
listen  127.0.0.1 default_server accept_filter=dataready backlog=1024;
```

下面说明 listen 可用参数的意义。

- ❑ default：将所在的 server 块作为整个 Web 服务的默认 server 块。如果没有设置这个参数，那么将会以在 nginx.conf 中找到的第一个 server 块作为默认 server 块。为什么需要默认虚拟主机呢？当一个请求无法匹配配置文件中的所有主机域名时，就会选用默

认的虚拟主机（在 11.3 节介绍默认主机的使用）。

- default_server：同上。
- backlog=num：表示 TCP 中 backlog 队列的大小。默认为 –1，表示不予设置。在 TCP 建立三次握手过程中，进程还没有开始处理监听句柄，这时 backlog 队列将会放置这些新连接。可如果 backlog 队列已满，还有新的客户端试图通过三次握手建立 TCP 连接，这时客户端将会建立连接失败。
- rcvbuf=size：设置监听句柄的 SO_RCVBUF 参数。
- sndbuf=size：设置监听句柄的 SO_SNDBUF 参数。
- accept_filter：设置 accept 过滤器，只对 FreeBSD 操作系统有用。
- deferred：在设置该参数后，若用户发起建立连接请求，并且完成了 TCP 的三次握手，内核也不会为了这次的连接调度 worker 进程来处理，只有用户真的发送请求数据时（内核已经在网卡中收到请求数据包），内核才会唤醒 worker 进程处理这个连接。这个参数适用于大并发的情况下，它减轻了 worker 进程的负担。当请求数据来临时，worker 进程才会开始处理这个连接。只有确认上面所说的应用场景符合自己的业务需求时，才可以使用 deferred 配置。
- bind：绑定当前端口 / 地址对，如 127.0.0.1:8000。只有同时对一个端口监听多个地址时才会生效。
- ssl：在当前监听的端口上建立的连接必须基于 SSL 协议。

（2）主机名称

语法：server_name name [...];

默认：server_name "";

配置块：server

server_name 后可以跟多个主机名称，如 server_name www.testweb.com、download.testweb.com;。

在开始处理一个 HTTP 请求时，Nginx 会取出 header 头中的 Host，与每个 server 中的 server_name 进行匹配，以此决定到底由哪一个 server 块来处理这个请求。有可能一个 Host 与多个 server 块中的 server_name 都匹配，这时就会根据匹配优先级来选择实际处理的 server 块。server_name 与 Host 的匹配优先级如下：

1）首先选择所有字符串完全匹配的 server_name，如 www.testweb.com。

2）其次选择通配符在前面的 server_name，如 *.testweb.com。

3）再次选择通配符在后面的 server_name，如 www.testweb.*。

4）最后选择使用正则表达式才匹配的 server_name，如 ~^\.testweb\.com$。

实际上，这个规则正是 7.7 节中介绍的带通配符散列表的实现依据，同时，在 10.4 节也介绍了虚拟主机配置的管理。如果 Host 与所有的 server_name 都不匹配，这时将会按下列顺

序选择处理的 server 块。

1）优先选择在 listen 配置项后加入 [default | default_server] 的 server 块。

2）找到匹配 listen 端口的第一个 server 块。

如果 server_name 后跟着空字符串（如 server_name "";），那么表示匹配没有 Host 这个 HTTP 头部的请求。

注意 Nginx 正是使用 server_name 配置项针对特定 Host 域名的请求提供不同的服务，以此实现虚拟主机功能。

（3）server_names_hash_bucket_size

语法：server_names _hash_bucket_size size;

默认：server_names _hash_bucket_size 32|64|128;

配置块：http、server、location

为了提高快速寻找到相应 server name 的能力，Nginx 使用散列表来存储 server name。server_names _hash_bucket_size 设置了每个散列桶占用的内存大小。

（4）server_names _hash_max_size

语法：server_names _hash_max_size size;

默认：server_names _hash_max_size 512;

配置块：http、server、location

server_names _hash_max_size 会影响散列表的冲突率。server_names _hash_max_size 越大，消耗的内存就越多，但散列 key 的冲突率则会降低，检索速度也更快。server_names _hash_max_size 越小，消耗的内存就越小，但散列 key 的冲突率可能增高。

（5）重定向主机名称的处理

语法：server_name_in_redirect on | off;

默认：server_name_in_redirect on;

配置块：http、server 或者 location

该配置需要配合 server_name 使用。在使用 on 打开时，表示在重定向请求时会使用 server_name 里配置的第一个主机名代替原先请求中的 Host 头部，而使用 off 关闭时，表示在重定向请求时使用请求本身的 Host 头部。

（6）location

语法：location [=|~|~*|^~|@] /uri/ { ... }

配置块：server

location 会尝试根据用户请求中的 URI 来匹配上面的 /uri 表达式，如果可以匹配，就选择 location {} 块中的配置来处理用户请求。当然，匹配方式是多样的，下面介绍 location 的匹配规则。

1）= 表示把 URI 作为字符串，以便与参数中的 uri 做完全匹配。例如：

```
location  = / {
  # 只有当用户请求是 / 时，才会使用该 location 下的配置
   ...
}
```

2）~ 表示匹配 URI 时是字母大小写敏感的。

3）~* 表示匹配 URI 时忽略字母大小写问题。

4）^~ 表示匹配 URI 时只需要其前半部分与 uri 参数匹配即可。例如：

```
location ^~ /images/ {
  # 以 /images/ 开始的请求都会匹配上
  ...
}
```

5）@ 表示仅用于 Nginx 服务内部请求之间的重定向，带有 @ 的 location 不直接处理用户请求。

当然，在 uri 参数里是可以用正则表达式的，例如：

```
location ~* \.(gif|jpg|jpeg)$ {
  # 匹配以 .gif、.jpg、.jpeg 结尾的请求
   ...
}
```

注意，location 是有顺序的，当一个请求有可能匹配多个 location 时，实际上这个请求会被第一个 location 处理。

在以上各种匹配方式中，都只能表达为“如果匹配 ... 则 ...”。如果需要表达“如果不匹配 ... 则 ...”，就很难直接做到。有一种解决方法是在最后一个 location 中使用 / 作为参数，它会匹配所有的 HTTP 请求，这样就可以表示如果不能匹配前面的所有 location，则由“/”这个 location 处理。例如：

```
location  / {
  # / 可以匹配所有请求
   ...
}
```

2.4.2　文件路径的定义

下面介绍一下文件路径的定义配置项。

（1）以 root 方式设置资源路径

语法：root path;

默认：root html;

配置块：http、server、location、if

例如，定义资源文件相对于 HTTP 请求的根目录。

```
location /download/ {
   root /opt/web/html/;
}
```

在上面的配置中，如果有一个请求的 URI 是 /download/index/test.html，那么 Web 服务器将会返回服务器上 /opt/web/html/download/index/test.html 文件的内容。

（2）以 alias 方式设置资源路径

语法：alias path;

配置块：location

alias 也是用来设置文件资源路径的，它与 root 的不同点主要在于如何解读紧跟 location 后面的 uri 参数，这将会致使 alias 与 root 以不同的方式将用户请求映射到真正的磁盘文件上。例如，如果有一个请求的 URI 是 /conf/nginx.conf，而用户实际想访问的文件在 /usr/local/nginx/conf/nginx.conf，那么想要使用 alias 来进行设置的话，可以采用如下方式：

```
location /conf {
            alias /usr/local/nginx/conf/;
}
```

如果用 root 设置，那么语句如下所示：

```
location /conf {
            root /usr/local/nginx/;
}
```

使用 alias 时，在 URI 向实际文件路径的映射过程中，已经把 location 后配置的 /conf 这部分字符串丢弃掉，因此，/conf/nginx.conf 请求将根据 alias path 映射为 path/nginx.conf。root 则不然，它会根据完整的 URI 请求来映射，因此，/conf/nginx.conf 请求会根据 root path 映射为 path/conf/nginx.conf。这也是 root 可以放置到 http、server、location 或 if 块中，而 alias 只能放置到 location 块中的原因。

alias 后面还可以添加正则表达式，例如：

```
location ~ ^/test/(\w+)\.(\w+)$ {
    alias /usr/local/nginx/$2/$1.$2;
}
```

这样，请求在访问 /test/nginx.conf 时，Nginx 会返回 /usr/local/nginx/conf/nginx.conf 文件中的内容。

（3）访问首页

语法：index file ...;

默认：index index.html;

配置块：http、server、location

有时，访问站点时的 URI 是 /，这时一般是返回网站的首页，而这与 root 和 alias 都不同。这里用 ngx_http_index_module 模块提供的 index 配置实现。index 后可以跟多个文件参数，Nginx 将会按照顺序来访问这些文件，例如：

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

接收到请求后，Nginx 首先会尝试访问 path/index.php 文件，如果可以访问，就直接返回文件内容结束请求，否则再试图返回 path/html/index.php 文件的内容，依此类推。

（4）根据 HTTP 返回码重定向页面

语法：error_page code [code...] [= | =answer-code] uri | @named_location

配置块：http、server、location、if

当对于某个请求返回错误码时，如果匹配上了 error_page 中设置的 code，则重定向到新的 URI 中。例如：

```
error_page   404          /404.html;
error_page   502 503 504  /50x.html;
error_page   403          http://example.com/forbidden.html;
error_page   404          = @fetch;
```

注意，虽然重定向了 URI，但返回的 HTTP 错误码还是与原来的相同。用户可以通过“=”来更改返回的错误码，例如：

```
error_page 404 =200 /empty.gif;
error_page 404 =403 /forbidden.gif;
```

也可以不指定确切的返回错误码，而是由重定向后实际处理的真实结果来决定，这时，只要把“=”后面的错误码去掉即可，例如：

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

如果不想修改 URI，只是想让这样的请求重定向到另一个 location 中进行处理，那么可以这样设置：

```
location / (
    error_page 404 @fallback;
)

location @fallback (
    proxy_pass http://backend;
)
```

这样，返回 404 的请求会被反向代理到 http://backend 上游服务器中处理。

（5）是否允许递归使用 error_page

语法：recursive_error_pages [on | off];

默认：recursive_error_pages off;

配置块：http、server、location

确定是否允许递归地定义 error_page。

（6）try_files

语法：try_files path1 [path2] uri;

配置块：server、location

try_files 后要跟若干路径，如 path1 path2...，而且最后必须要有 uri 参数，意义如下：尝试按照顺序访问每一个 path，如果可以有效地读取，就直接向用户返回这个 path 对应的文件结束请求，否则继续向下访问。如果所有的 path 都找不到有效的文件，就重定向到最后的参数 uri 上。因此，最后这个参数 uri 必须存在，而且它应该是可以有效重定向的。例如：

```
try_files /system/maintenance.html $uri $uri/index.html $uri.html @other;
location @other {
  proxy_pass http://backend;
}
```

上面这段代码表示如果前面的路径，如 /system/maintenance.html 等，都找不到，就会反向代理到 http://backend 服务上。还可以用指定错误码的方式与 error_page 配合使用，例如：

```
location / {
  try_files $uri $uri/ /error.php?c=404 =404;
}
```

2.4.3 内存及磁盘资源的分配

下面介绍处理请求时内存、磁盘资源分配的配置项。

（1）HTTP 包体只存储到磁盘文件中

语法：client_body_in_file_only on | clean | off;

默认：client_body_in_file_only off;

配置块：http、server、location

当值为非 off 时，用户请求中的 HTTP 包体一律存储到磁盘文件中，即使只有 0 字节也会存储为文件。当请求结束时，如果配置为 on，则这个文件不会被删除（该配置一般用于调试、定位问题），但如果配置为 clean，则会删除该文件。

（2）HTTP 包体尽量写入到一个内存 buffer 中

语法：client_body_in_single_buffer on | off;

默认：client_body_in_single_buffer off;

配置块：http、server、location

用户请求中的 HTTP 包体一律存储到内存 buffer 中。当然，如果 HTTP 包体的大小超过了下面 client_body_buffer_size 设置的值，包体还是会写入到磁盘文件中。

（3）存储 HTTP 头部的内存 buffer 大小

语法：client_header_buffer_size size;

默认：client_header_buffer_size 1k;

配置块：http、server

上面配置项定义了正常情况下 Nginx 接收用户请求中 HTTP header 部分（包括 HTTP 行和 HTTP 头部）时分配的内存 buffer 大小。有时，请求中的 HTTP header 部分可能会超过这个大小，这时 large_client_header_buffers 定义的 buffer 将会生效。

（4）存储超大 HTTP 头部的内存 buffer 大小

语法：large_client_header_buffers number size;

默认：large_client_header_buffers 4 8k;

配置块：http、server

large_client_header_buffers 定义了 Nginx 接收一个超大 HTTP 头部请求的 buffer 个数和每个 buffer 的大小。如果 HTTP 请求行（如 GET /index HTTP/1.1）的大小超过上面的单个 buffer，则返回 "Request URI too large" (414)。请求中一般会有许多 header，每一个 header 的大小也不能超过单个 buffer 的大小，否则会返回 "Bad request" (400)。当然，请求行和请求头部的总和也不可以超过 buffer 个数 *buffer 大小。

（5）存储 HTTP 包体的内存 buffer 大小

语法：client_body_buffer_size size;

默认：client_body_buffer_size 8k/16k;

配置块：http、server、location

上面配置项定义了 Nginx 接收 HTTP 包体的内存缓冲区大小。也就是说，HTTP 包体会先接收到指定的这块缓存中，之后才决定是否写入磁盘。

注意　如果用户请求中含有 HTTP 头部 Content-Length，并且其标识的长度小于定义的 buffer 大小，那么 Nginx 会自动降低本次请求所使用的内存 buffer，以降低内存消耗。

（6）HTTP 包体的临时存放目录

语法：client_body_temp_path dir-path [level1 [level2 [level3]]]

默认：client_body_temp_path client_body_temp;

配置块：http、server、location

上面配置项定义 HTTP 包体存放的临时目录。在接收 HTTP 包体时，如果包体的大小大于 client_body_buffer_size，则会以一个递增的整数命名并存放到 client_body_temp_path 指

定的目录中。后面跟着的 level1、level2、level3，是为了防止一个目录下的文件数量太多，从而导致性能下降，因此使用了 level 参数，这样可以按照临时文件名最多再加三层目录。例如：

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

如果新上传的 HTTP 包体使用 00000123456 作为临时文件名，就会被存放在这个目录中。

```
/opt/nginx/client_temp/6/45/00000123456
```

（7）connection_pool_size

语法：connection_pool_size size;

默认：connection_pool_size 256;

配置块：http、server

Nginx 对于每个建立成功的 TCP 连接会预先分配一个内存池，上面的 size 配置项将指定这个内存池的初始大小（即 ngx_connection_t 结构体中的 pool 内存池初始大小，9.8.1 节将介绍这个内存池是何时分配的），用于减少内核对于小块内存的分配次数。需慎重设置，因为更大的 size 会使服务器消耗的内存增多，而更小的 size 则会引发更多的内存分配次数。

（8）request_pool_size

语法：request_pool_size size;

默认：request_pool_size 4k;

配置块：http、server

Nginx 开始处理 HTTP 请求时，将会为每个请求都分配一个内存池，size 配置项将指定这个内存池的初始大小（即 ngx_http_request_t 结构体中的 pool 内存池初始大小，11.3 节将介绍这个内存池是何时分配的），用于减少内核对于小块内存的分配次数。TCP 连接关闭时会销毁 connection_pool_size 指定的连接内存池，HTTP 请求结束时会销毁 request_pool_size 指定的 HTTP 请求内存池，但它们的创建、销毁时间并不一致，因为一个 TCP 连接可能被复用于多个 HTTP 请求。

2.4.4 网络连接的设置

下面介绍网络连接的设置配置项。

（1）读取 HTTP 头部的超时时间

语法：client_header_timeout time（默认单位：秒）;

默认：client_header_timeout 60;

配置块：http、server、location

客户端与服务器建立连接后将开始接收 HTTP 头部，在这个过程中，如果在一个时间间隔（超时时间）内没有读取到客户端发来的字节，则认为超时，并向客户端返回 408 ("Request

timed out") 响应。

（2）读取 HTTP 包体的超时时间

语法：client_body_timeout time（默认单位：秒）；

默认：client_body_timeout 60;

配置块：http、server、location

此配置项与 client_header_timeout 相似，只是这个超时时间只在读取 HTTP 包体时才有效。

（3）发送响应的超时时间

语法：send_timeout time;

默认：send_timeout 60;

配置块：http、server、location

这个超时时间是发送响应的超时时间，即 Nginx 服务器向客户端发送了数据包，但客户端一直没有去接收这个数据包。如果某个连接超过 send_timeout 定义的超时时间，那么 Nginx 将会关闭这个连接。

（4）reset_timeout_connection

语法：reset_timeout_connection on | off;

默认：reset_timeout_connection off;

配置块：http、server、location

连接超时后将通过向客户端发送 RST 包来直接重置连接。这个选项打开后，Nginx 会在某个连接超时后，不是使用正常情形下的四次握手关闭 TCP 连接，而是直接向用户发送 RST 重置包，不再等待用户的应答，直接释放 Nginx 服务器上关于这个套接字使用的所有缓存（如 TCP 滑动窗口）。相比正常的关闭方式，它使得服务器避免产生许多处于 FIN_WAIT_1、FIN_WAIT_2、TIME_WAIT 状态的 TCP 连接。

注意，使用 RST 重置包关闭连接会带来一些问题，默认情况下不会开启。

（5）lingering_close

语法：lingering_close off | on | always;

默认：lingering_close on;

配置块：http、server、location

该配置控制 Nginx 关闭用户连接的方式。always 表示关闭用户连接前必须无条件地处理连接上所有用户发送的数据。off 表示关闭连接时完全不管连接上是否已经有准备就绪的来自用户的数据。on 是中间值，一般情况下在关闭连接前都会处理连接上的用户发送的数据，除了有些情况下在业务上认定这之后的数据是不必要的。

（6）lingering_time

语法：lingering_time time;

默认：lingering_time 30s;

配置块：http、server、location

lingering_close 启用后，这个配置项对于上传大文件很有用。上文讲过，当用户请求的 Content-Length 大于 max_client_body_size 配置时，Nginx 服务会立刻向用户发送 413（Request entity too large）响应。但是，很多客户端可能不管 413 返回值，仍然持续不断地上传 HTTP body，这时，经过了 lingering_time 设置的时间后，Nginx 将不管用户是否仍在上传，都会把连接关闭掉。

（7）lingering_timeout

语法：lingering_timeout time;

默认：lingering_timeout 5s;

配置块：http、server、location

lingering_close 生效后，在关闭连接前，会检测是否有用户发送的数据到达服务器，如果超过 lingering_timeout 时间后还没有数据可读，就直接关闭连接；否则，必须在读取完连接缓冲区上的数据并丢弃掉后才会关闭连接。

（8）对某些浏览器禁用 keepalive 功能

语法：keepalive_disable [msie6 | safari | none]...

默认：keepalive_disable msie6 safari

配置块：http、server、location

HTTP 请求中的 keepalive 功能是为了让多个请求复用一个 HTTP 长连接，这个功能对服务器的性能提高是很有帮助的。但有些浏览器，如 IE 6 和 Safari，它们对于使用 keepalive 功能的 POST 请求处理有功能性问题。因此，针对 IE 6 及其早期版本、Safari 浏览器默认是禁用 keepalive 功能的。

（9）keepalive 超时时间

语法：keepalive_timeout time（默认单位：秒）;

默认：keepalive_timeout 75;

配置块：http、server、location

一个 keepalive 连接在闲置超过一定时间后（默认的是 75 秒），服务器和浏览器都会去关闭这个连接。当然，keepalive_timeout 配置项是用来约束 Nginx 服务器的，Nginx 也会按照规范把这个时间传给浏览器，但每个浏览器对待 keepalive 的策略有可能是不同的。

（10）一个 keepalive 长连接上允许承载的请求最大数

语法：keepalive_requests n;

默认：keepalive_requests 100;

配置块：http、server、location

一个 keepalive 连接上默认最多只能发送 100 个请求。

（11）tcp_nodelay

语法：tcp_nodelay on | off;

默认：tcp_nodelay on;

配置块：http、server、location

确定对 keepalive 连接是否使用 TCP_NODELAY 选项。

（12）tcp_nopush

语法：tcp_nopush on | off;

默认：tcp_nopush off;

配置块：http、server、location

在打开 sendfile 选项时，确定是否开启 FreeBSD 系统上的 TCP_NOPUSH 或 Linux 系统上的 TCP_CORK 功能。打开 tcp_nopush 后，将会在发送响应时把整个响应包头放到一个 TCP 包中发送。

2.4.5　MIME 类型的设置

下面是 MIME 类型的设置配置项。

❑ MIME type 与文件扩展的映射

语法：type {...};

配置块：http、server、location

定义 MIME type 到文件扩展名的映射。多个扩展名可以映射到同一个 MIME type。例如：

```
types {
    text/html    html;
    text/html    conf;
    image/gif    gif;
    image/jpeg   jpg;
}
```

❑ 默认 MIME type

语法：default_type MIME-type;

默认：default_type text/plain;

配置块：http、server、location

当找不到相应的 MIME type 与文件扩展名之间的映射时，使用默认的 MIME type 作为 HTTP header 中的 Content-Type。

❑ types_hash_bucket_size

语法：types_hash_bucket_size size;

默认：types_hash_bucket_size 32|64|128;

配置块：http、server、location

为了快速寻找到相应 MIME type，Nginx 使用散列表来存储 MIME type 与文件扩展名。types_hash_bucket_size 设置了每个散列桶占用的内存大小。

❑ types_hash_max_size

语法：types_hash_max_size size;

默认：types_hash_max_size 1024;

配置块：http、server、location

types_hash_max_size 影响散列表的冲突率。types_hash_max_size 越大，就会消耗更多的内存，但散列 key 的冲突率会降低，检索速度就更快。types_hash_max_size 越小，消耗的内存就越小，但散列 key 的冲突率可能上升。

2.4.6 对客户端请求的限制

下面介绍对客户端请求的限制的配置项。

（1）按 HTTP 方法名限制用户请求

语法：limit_except method ... {...}

配置块：location

Nginx 通过 limit_except 后面指定的方法名来限制用户请求。方法名可取值包括：GET、HEAD、POST、PUT、DELETE、MKCOL、COPY、MOVE、OPTIONS、PROPFIND、PROPPATCH、LOCK、UNLOCK 或者 PATCH。例如：

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

注意，允许 GET 方法就意味着也允许 HEAD 方法。因此，上面这段代码表示的是禁止 GET 方法和 HEAD 方法，但其他 HTTP 方法是允许的。

（2）HTTP 请求包体的最大值

语法：client_max_body_size size;

默认：client_max_body_size 1m;

配置块：http、server、location

浏览器在发送含有较大 HTTP 包体的请求时，其头部会有一个 Content-Length 字段，client_max_body_size 是用来限制 Content-Length 所示值的大小的。因此，这个限制包体的配置非常有用处，因为不用等 Nginx 接收完所有的 HTTP 包体——这有可能消耗很长时间——就可以告诉用户请求过大不被接受。例如，用户试图上传一个 10GB 的文件，Nginx 在收完包头后，发现 Content-Length 超过 client_max_body_size 定义的值，就直接发送 413 ("Request Entity Too Large") 响应给客户端。

（3）对请求的限速

语法：limit_rate speed;

默认：limit_rate 0;

配置块：http、server、location、if

此配置是对客户端请求限制每秒传输的字节数。speed 可以使用 2.2.4 节中提到的多种单位，默认参数为 0，表示不限速。

针对不同的客户端，可以用 $ limit_rate 参数执行不同的限速策略。例如：

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

（4）limit_rate_after

语法：limit_rate_after time;

默认：limit_rate_after 1m;

配置块：http、server、location、if

此配置表示 Nginx 向客户端发送的响应长度超过 limit_rate_after 后才开始限速。例如：

```
limit_rate_after 1m;
limit_rate 100k;
```

11.9.2 节将从源码上介绍 limit_rate_after 与 limit_rate 的区别，以及 HTTP 框架是如何使用它们来限制发送响应速度的。

2.4.7　文件操作的优化

下面介绍文件操作的优化配置项。

（1）sendfile 系统调用

语法：sendfile on | off;

默认：sendfile off;

配置块：http、server、location

可以启用 Linux 上的 sendfile 系统调用来发送文件，它减少了内核态与用户态之间的两次内存复制，这样就会从磁盘中读取文件后直接在内核态发送到网卡设备，提高了发送文件的效率。

（2）AIO 系统调用

语法：aio on | off;

默认：aio off;

配置块：http、server、location

此配置项表示是否在 FreeBSD 或 Linux 系统上启用内核级别的异步文件 I/O 功能。注意，它与 sendfile 功能是互斥的。

（3）directio

语法：directio size | off;

默认：directio off;

配置块：http、server、location

此配置项在 FreeBSD 和 Linux 系统上使用 O_DIRECT 选项去读取文件，缓冲区大小为 size，通常对大文件的读取速度有优化作用。注意，它与 sendfile 功能是互斥的。

（4）directio_alignment

语法：directio_alignment size;

默认：directio_alignment 512;

配置块：http、server、location

它与 directio 配合使用，指定以 directio 方式读取文件时的对齐方式。一般情况下，512B 已经足够了，但针对一些高性能文件系统，如 Linux 下的 XFS 文件系统，可能需要设置到 4KB 作为对齐方式。

（5）打开文件缓存

语法：open_file_cache max = N [inactive = time] | off;

默认：open_file_cache off;

配置块：http、server、location

文件缓存会在内存中存储以下 3 种信息：

- 文件句柄、文件大小和上次修改时间。
- 已经打开过的目录结构。
- 没有找到的或者没有权限操作的文件信息。

这样，通过读取缓存就减少了对磁盘的操作。

该配置项后面跟 3 种参数。

- max：表示在内存中存储元素的最大个数。当达到最大限制数量后，将采用 LRU（Least Recently Used）算法从缓存中淘汰最近最少使用的元素。
- inactive：表示在 inactive 指定的时间段内没有被访问过的元素将会被淘汰。默认时间为 60 秒。
- off：关闭缓存功能。

例如：

```
open_file_cache max=1000 inactive=20s;
```

（6）是否缓存打开文件错误的信息

语法：open_file_cache_errors on | off;

默认：open_file_cache_errors off;

配置块：http、server、location

此配置项表示是否在文件缓存中缓存打开文件时出现的找不到路径、没有权限等错误信息。

（7）不被淘汰的最小访问次数

语法：open_file_cache_min_uses number;

默认：open_file_cache_min_uses 1;

配置块：http、server、location

它与 open_file_cache 中的 inactive 参数配合使用。如果在 inactive 指定的时间段内，访问次数超过了 open_file_cache_min_uses 指定的最小次数，那么将不会被淘汰出缓存。

（8）检验缓存中元素有效性的频率

语法：open_file_cache_valid time;

默认：open_file_cache_valid 60s;

配置块：http、server、location

默认为每 60 秒检查一次缓存中的元素是否仍有效。

2.4.8　对客户端请求的特殊处理

下面介绍对客户端请求的特殊处理的配置项。

（1）忽略不合法的 HTTP 头部

语法：ignore_invalid_headers on | off;

默认：ignore_invalid_headers on;

配置块：http、server

如果将其设置为 off，那么当出现不合法的 HTTP 头部时，Nginx 会拒绝服务，并直接向用户发送 400（Bad Request）错误。如果将其设置为 on，则会忽略此 HTTP 头部。

（2）HTTP 头部是否允许下画线

语法：underscores_in_headers on | off;

默认：underscores_in_headers off;

配置块：http、server

默认为 off，表示 HTTP 头部的名称中不允许带“_”（下画线）。

（3）对 If-Modified-Since 头部的处理策略

语法：if_modified_since [off|exact|before];

默认：if_modified_since exact;

配置块：http、server、location

出于性能考虑，Web 浏览器一般会在客户端本地缓存一些文件，并存储当时获取的时间。这样，下次向 Web 服务器获取缓存过的资源时，就可以用 If-Modified-Since 头部把上次获取的时间捎带上，而 if_modified_since 将根据后面的参数决定如何处理 If-Modified-Since 头部。

相关参数说明如下。

- off：表示忽略用户请求中的 If-Modified-Since 头部。这时，如果获取一个文件，那么会正常地返回文件内容。HTTP 响应码通常是 200。
- exact：将 If-Modified-Since 头部包含的时间与将要返回的文件上次修改的时间做精确比较，如果没有匹配上，则返回 200 和文件的实际内容，如果匹配上，则表示浏览器缓存的文件内容已经是最新的了，没有必要再返回文件从而浪费时间与带宽了，这时会返回 304 Not Modified，浏览器收到后会直接读取自己的本地缓存。
- before ：是比 exact 更宽松的比较。只要文件的上次修改时间等于或者早于用户请求中的 If-Modified-Since 头部的时间，就会向客户端返回 304 Not Modified。

（4）文件未找到时是否记录到 error 日志

语法：log_not_found on | off;

默认：log_not_found on;

配置块：http、server、location

此配置项表示当处理用户请求且需要访问文件时，如果没有找到文件，是否将错误日志记录到 error.log 文件中。这仅用于定位问题。

（5）merge_slashes

语法：merge_slashes on | off;

默认：merge_slashes on;

配置块：http、server、location

此配置项表示是否合并相邻的“/”，例如，//test///a.txt，在配置为 on 时，会将其匹配为 location /test/a.txt；如果配置为 off，则不会匹配，URI 将仍然是 //test///a.txt。

（6）DNS 解析地址

语法：resolver address ...;

配置块：http、server、location

设置 DNS 名字解析服务器的地址，例如：

```
resolver 127.0.0.1 192.0.2.1;
```

（7）DNS 解析的超时时间

语法：resolver_timeout time;

默认：resolver_timeout 30s;

配置块：http、server、location

此配置项表示 DNS 解析的超时时间。

（8）返回错误页面时是否在 Server 中注明 Nginx 版本

语法：server_tokens on | off;

默认：server_tokens on;

配置块：http、server、location

表示处理请求出错时是否在响应的 Server 头部中标明 Nginx 版本，这是为了方便定位问题。

2.4.9　ngx_http_core_module 模块提供的变量

在记录 access_log 访问日志文件时，可以使用 ngx_http_core_module 模块处理请求时所产生的丰富的变量，当然，这些变量还可以用于其他 HTTP 模块。例如，当 URI 中的某个参数满足设定的条件时，有些 HTTP 模块的配置项可以使用类似 $arg_PARAMETER 这样的变量。又如，若想把每个请求中的限速信息记录到 access 日志文件中，则可以使用 $limit_rate 变量。

表 2-1 列出了 ngx_http_core_module 模块提供的这些变量。

表 2-1　ngx_http_core_module 模块提供的变量

参 数 名	意　义
$arg_PARAMETER	HTTP 请求中某个参数的值，如 /index.html?size=100，可以用 $arg_size 取得 100 这个值
$args	HTTP 请求中的完整参数。例如，在请求 /index.html?_w=120&_h=120 中，$args 表示字符串 _w=120&_h=120
$binary_remote_addr	二进制格式的客户端地址。例如：\x0A\xE0B\x0E
$body_bytes_sent	表示在向客户端发送的 http 响应中，包体部分的字节数
$content_length	表示客户端请求头部中的 Content-Length 字段
$content_type	表示客户端请求头部中的 Content-Type 字段
$cookie_COOKIE	表示在客户端请求头部中的 cookie 字段
$document_root	表示当前请求所使用的 root 配置项的值
$uri	表示当前请求的 URI，不带任何参数
$document_uri	与 $uri 含义相同
$request_uri	表示客户端发来的原始请求 URI，带完整的参数。$uri 和 $document_uri 未必是用户的原始请求，在内部重定向后可能是重定向后的 URI，而 $request_uri 永远不会改变，始终是客户端的原始 URI
$host	表示客户端请求头部中的 Host 字段。如果 Host 字段不存在，则以实际处理的 server（虚拟主机）名称代替。如果 Host 字段中带有端口，如 IP:PORT，那么 $host 是去掉端口的，它的值为 IP。$host 是全小写的。这些特性与 http_HEADER 中的 http_host 不同，http_host 只是“忠实”地取出 Host 头部对应的值

（续）

参数名	意 义
$hostname	表示 Nginx 所在机器的名称，与 gethostbyname 调用返回的值相同
$http_HEADER	表示当前 HTTP 请求中相应头部的值。HEADER 名称全小写。例如，用 $http_host 表示请求中 Host 头部对应的值
$sent_http_HEADER	表示返回客户端的 HTTP 响应中相应头部的值。HEADER 名称全小写。例如，用 $sent_http_content_type 表示响应中 Content-Type 头部对应的值
$is_args	表示请求中的 URI 是否带参数，如果带参数，$is_args 值为 ?，如果不带参数，则是空字符串
$limit_rate	表示当前连接的限速是多少，0 表示无限速
$nginx_version	表示当前 Nginx 的版本号，如 1.0.14
$query_string	请求 URI 中的参数，与 $args 相同，然而 $query_string 是只读的不会改变
$remote_addr	表示客户端的地址
$remote_port	表示客户端连接使用的端口
$remote_user	表示使用 Auth Basic Module 时定义的用户名
$request_filename	表示用户请求中的 URI 经过 root 或 alias 转换后的文件路径
$request_body	表示 HTTP 请求中的包体，该参数只在 proxy_pass 或 fastcgi_pass 中有意义
$request_body_file	表示 HTTP 请求中的包体存储的临时文件名
$request_completion	当请求已经全部完成时，其值为“ok”。若没有完成，就要返回客户端，则其值为空字符串；或者在断点续传等情况下使用 HTTP range 访问的并不是文件的最后一块，那么其值也是空字符串
$request_method	表示 HTTP 请求的方法名，如 GET、PUT、POST 等
$scheme	表示 HTTP scheme，如在请求 https://nginx.com/ 中表示 https
$server_addr	表示服务器地址
$server_name	表示服务器名称
$server_port	表示服务器端口
$server_protocol	表示服务器向客户端发送响应的协议，如 HTTP/1.1 或 HTTP/1.0

2.5 用 HTTP proxy module 配置一个反向代理服务器

反向代理（reverse proxy）方式是指用代理服务器来接受 Internet 上的连接请求，然后将请求转发给内部网络中的上游服务器，并将从上游服务器上得到的结果返回给 Internet 上请求连接的客户端，此时代理服务器对外的表现就是一个 Web 服务器。充当反向代理服务器也是 Nginx 的一种常见用法（反向代理服务器必须能够处理大量并发请求），本节将介绍 Nginx 作为 HTTP 反向代理服务器的基本用法。

由于 Nginx 具有“强悍”的高并发高负载能力，因此一般会作为前端的服务器直接向客户端提供静态文件服务。但也有一些复杂、多变的业务不适合放到 Nginx 服务器上，这时会用 Apache、Tomcat 等服务器来处理。于是，Nginx 通常会被配置为既是静态 Web 服务器也是反向代理服务器（如图 2-3 所示），不适合 Nginx 处理的请求就会直接转发到上游服务器中处理。

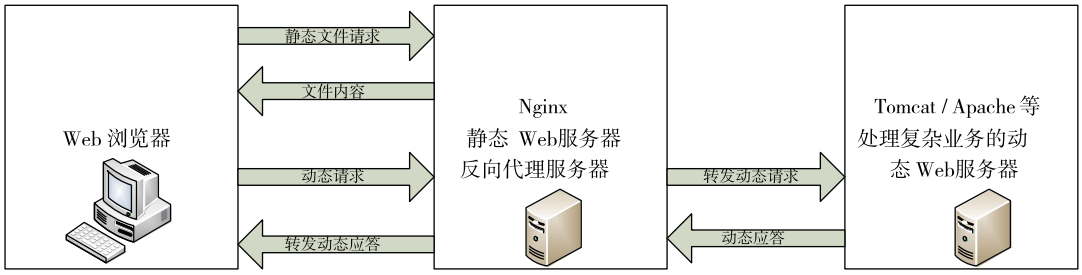


图 2-3　作为静态 Web 服务器与反向代理服务器的 Nginx

与 Squid 等其他反向代理服务器相比，Nginx 的反向代理功能有自己的特点，如图 2-4 所示。

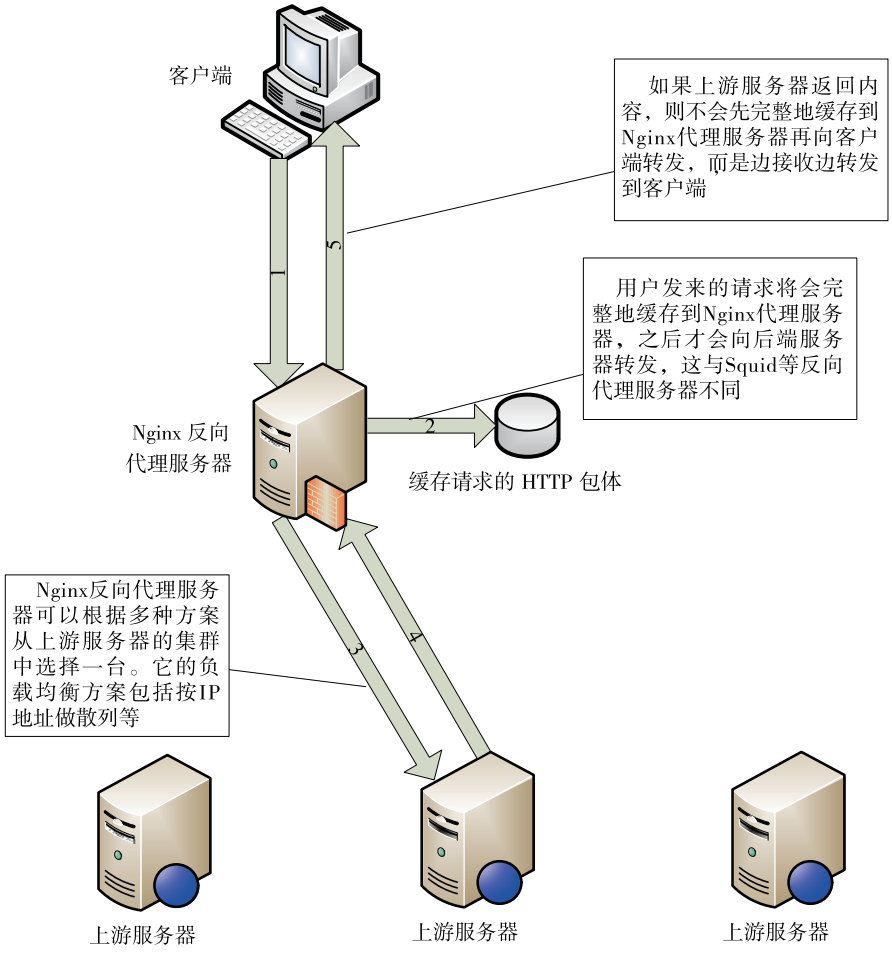


图 2-4　Nginx 作为反向代理服务器时转发请求的流程

当客户端发来 HTTP 请求时，Nginx 并不会立刻转发到上游服务器，而是先把用户的请求（包括 HTTP 包体）完整地接收到 Nginx 所在服务器的硬盘或者内存中，然后再向上游服务器发起连接，把缓存的客户端请求转发到上游服务器。而 Squid 等代理服务器则采用一边接收客户端请求，一边转发到上游服务器的方式。

Nginx 的这种工作方式有什么优缺点呢？很明显，缺点是延长了一个请求的处理时间，并增加了用于缓存请求内容的内存和磁盘空间。而优点则是降低了上游服务器的负载，尽量把压力放在 Nginx 服务器上。

Nginx 的这种工作方式为什么会降低上游服务器的负载呢？通常，客户端与代理服务器之间的网络环境会比较复杂，多半是“走”公网，网速平均下来可能较慢，因此，一个请求可能要持续很久才能完成。而代理服务器与上游服务器之间一般是“走”内网，或者有专线连接，传输速度较快。Squid 等反向代理服务器在与客户端建立连接且还没有开始接收 HTTP 包体时，就已经向上游服务器建立了连接。例如，某个请求要上传一个 1GB 的文件，那么每次 Squid 在收到一个 TCP 分包（如 2KB）时，就会即时地向上游服务器转发。在接收客户端完整 HTTP 包体的漫长过程中，上游服务器始终要维持这个连接，这直接对上游服务器的并发处理能力提出了挑战。

Nginx 则不然，它在接收到完整的客户端请求（如 1GB 的文件）后，才会与上游服务器建立连接转发请求，由于是内网，所以这个转发过程会执行得很快。这样，一个客户端请求占用上游服务器的连接时间就会非常短，也就是说，Nginx 的这种反向代理方案主要是为了降低上游服务器的并发压力。

Nginx 将上游服务器的响应转发到客户端有许多种方法，第 12 章将介绍其中常见的两种方式。

2.5.1 负载均衡的基本配置

作为代理服务器，一般都需要向上游服务器的集群转发请求。这里的负载均衡是指选择一种策略，尽量把请求平均地分布到每一台上游服务器上。下面介绍负载均衡的配置项。

（1）upstream 块

语法：upstream name {...}

配置块：http

upstream 块定义了一个上游服务器的集群，便于反向代理中的 proxy_pass 使用。例如：

```
upstream backend {
  server backend1.example.com;
  server backend2.example.com;
    server backend3.example.com;
}
```

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

（2）server

语法：server name [parameters];

配置块：upstream

server 配置项指定了一台上游服务器的名字，这个名字可以是域名、IP 地址端口、UNIX 句柄等，在其后还可以跟下列参数。

- weight=number：设置向这台上游服务器转发的权重，默认为 1。
- max_fails=number ：该选项与 fail_timeout 配合使用，指在 fail_timeout 时间段内，如果向当前的上游服务器转发失败次数超过 number，则认为在当前的 fail_timeout 时间段内这台上游服务器不可用。max_fails 默认为 1，如果设置为 0，则表示不检查失败次数。
- fail_timeout=time ：fail_timeout 表示该时间段内转发失败多少次后就认为上游服务器暂时不可用，用于优化反向代理功能。它与向上游服务器建立连接的超时时间、读取上游服务器的响应超时时间等完全无关。fail_timeout 默认为 10 秒。
- down：表示所在的上游服务器永久下线，只在使用 ip_hash 配置项时才有用。
- backup ：在使用 ip_hash 配置项时它是无效的。它表示所在的上游服务器只是备份服务器，只有在所有的非备份上游服务器都失效后，才会向所在的上游服务器转发请求。

例如：

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

（3）ip_hash

语法：ip_hash;

配置块：upstream

在有些场景下，我们可能会希望来自某一个用户的请求始终落到固定的一台上游服务器中。例如，假设上游服务器会缓存一些信息，如果同一个用户的请求任意地转发到集群中的任一台上游服务器中，那么每一台上游服务器都有可能会缓存同一份信息，这既会造成资源的浪费，也会难以有效地管理缓存信息。ip_hash 就是用以解决上述问题的，它首先根据客户端的 IP 地址计算出一个 key，将 key 按照 upstream 集群里的上游服务器数量进行取模，然后以取模后的结果把请求转发到相应的上游服务器中。这样就确保了同一个客户端的请求只

会转发到指定的上游服务器中。

ip_hash 与 weight（权重）配置不可同时使用。如果 upstream 集群中有一台上游服务器暂时不可用，不能直接删除该配置，而是要 down 参数标识，确保转发策略的一贯性。例如：

```
upstream backend {
  ip_hash;
  server   backend1.example.com;
  server   backend2.example.com;
  server   backend3.example.com  down;
  server   backend4.example.com;
}
```

（4）记录日志时支持的变量

如果需要将负载均衡时的一些信息记录到 access_log 日志中，那么在定义日志格式时可以使用负载均衡功能提供的变量，见表 2-2。

表 2-2 访问上游服务器时可以使用的变量

变 量 名	意 义
$upstream_addr	处理请求的上游服务器地址
$upstream_cache_status	表示是否命中缓存，取值范围：MISS、EXPIRED、UPDATING、STALE、HIT
$upstream_status	上游服务器返回的响应中的 HTTP 响应码
$upstream_response_time	上游服务器的响应时间，精度到毫秒
$upstream_http_$HEADER	HTTP 的头部，如 upstream_http_host

例如，可以在定义 access_log 访问日志格式时使用表 2-2 中的变量。

```
log_format timing '$remote_addr - $remote_user [$time_local]  $request '
  'upstream_response_time $upstream_response_time '
  'msec $msec request_time $request_time';

log_format up_head '$remote_addr - $remote_user [$time_local]  $request '
  'upstream_http_content_type $upstream_http_content_type';
```

2.5.2 反向代理的基本配置

下面介绍反向代理的基本配置项。

（1）proxy_pass

语法：proxy_pass URL;

配置块：location、if

此配置项将当前请求反向代理到 URL 参数指定的服务器上，URL 可以是主机名或 IP 地址加端口的形式，例如：

```
proxy_pass http://localhost:8000/uri/;
```

也可以是 UNIX 句柄：

```
proxy_pass http://unix:/path/to/backend.socket:/uri/;
```

还可以如上节负载均衡中所示，直接使用 upstream 块，例如：

```
upstream backend {
  ...
}

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

用户可以把 HTTP 转换成更安全的 HTTPS，例如：

```
proxy_pass https://192.168.0.1;
```

默认情况下反向代理是不会转发请求中的 Host 头部的。如果需要转发，那么必须加上配置：

```
proxy_set_header Host $host;
```

（2）proxy_method

语法：proxy_method method;

配置块：http、server、location

此配置项表示转发时的协议方法名。例如设置为：

```
proxy_method POST;
```

那么客户端发来的 GET 请求在转发时方法名也会改为 POST。

（3）proxy_hide_header

语法：proxy_hide_header the_header;

配置块：http、server、location

Nginx 会将上游服务器的响应转发给客户端，但默认不会转发以下 HTTP 头部字段：Date、Server、X-Pad 和 X-Accel-*。使用 proxy_hide_header 后可以任意地指定哪些 HTTP 头部字段不能被转发。例如：

```
proxy_hide_header Cache-Control;
proxy_hide_header MicrosoftOfficeWebServer;
```

（4）proxy_pass_header

语法：proxy_pass_header the_header;

配置块：http、server、location

与 proxy_hide_header 功能相反，proxy_pass_header 会将原来禁止转发的 header 设置为允许转发。例如：

```
proxy_pass_header X-Accel-Redirect;
```

（5）proxy_pass_request_body

语法：proxy_pass_request_body on | off;

默认：proxy_pass_request_body on;

配置块：http、server、location

作用为确定是否向上游服务器发送 HTTP 包体部分。

（6）proxy_pass_request_headers

语法：proxy_pass_request_headers on | off;

默认：proxy_pass_request_headers on;

配置块：http、server、location

作用为确定是否转发 HTTP 头部。

（7）proxy_redirect

语法：proxy_redirect [default|off|redirect replacement];

默认：proxy_redirect default;

配置块：http、server、location

当上游服务器返回的响应是重定向或刷新请求（如 HTTP 响应码是 301 或者 302）时，proxy_redirect 可以重设 HTTP 头部的 location 或 refresh 字段。例如，如果上游服务器发出的响应是 302 重定向请求，location 字段的 URI 是 http://localhost:8000/two/some/uri/，那么在下面的配置情况下，实际转发给客户端的 location 是 http://frontend/one/some/uri/。

```
proxy_redirect http://localhost:8000/two/ http://frontend/one/;
```

这里还可以使用 ngx-http-core-module 提供的变量来设置新的 location 字段。例如：

```
proxy_redirect   http://localhost:8000/    http://$host:$server_port/;
```

也可以省略 replacement 参数中的主机名部分，这时会用虚拟主机名称来填充。例如：

```
proxy_redirect http://localhost:8000/two/ /one/;
```

使用 off 参数时，将使 location 或者 refresh 字段维持不变。例如：

```
proxy_redirect off;
```

使用默认的 default 参数时，会按照 proxy_pass 配置项和所属的 location 配置项重组发往客户端的 location 头部。例如，下面两种配置效果是一样的：

```
location /one/ {
  proxy_pass       http://upstream:port/two/;
  proxy_redirect   default;
}

location /one/ {
  proxy_pass       http://upstream:port/two/;
  proxy_redirect   http://upstream:port/two/   /one/;
}
```

（8）proxy_next_upstream

语法：proxy_next_upstream [error | timeout | invalid_header | http_500 | http_502 | http_503 | http_504 | http_404 | off];

默认：proxy_next_upstream error timeout;

配置块：http、server、location

此配置项表示当向一台上游服务器转发请求出现错误时，继续换一台上游服务器处理这个请求。前面已经说过，上游服务器一旦开始发送应答，Nginx 反向代理服务器会立刻把应答包转发给客户端。因此，一旦 Nginx 开始向客户端发送响应包，之后的过程中若出现错误也是不允许换下一台上游服务器继续处理的。这很好理解，这样才可以更好地保证客户端只收到来自一个上游服务器的应答。proxy_next_upstream 的参数用来说明在哪些情况下会继续选择下一台上游服务器转发请求。

- ❑ error：当向上游服务器发起连接、发送请求、读取响应时出错。
- ❑ timeout：发送请求或读取响应时发生超时。
- ❑ invalid_header：上游服务器发送的响应是不合法的。
- ❑ http_500：上游服务器返回的 HTTP 响应码是 500。
- ❑ http_502：上游服务器返回的 HTTP 响应码是 502。
- ❑ http_503：上游服务器返回的 HTTP 响应码是 503。
- ❑ http_504：上游服务器返回的 HTTP 响应码是 504。
- ❑ http_404：上游服务器返回的 HTTP 响应码是 404。
- ❑ off：关闭 proxy_next_upstream 功能—出错就选择另一台上游服务器再次转发。

Nginx 的反向代理模块还提供了很多种配置，如设置连接的超时时间、临时文件如何存储，以及最重要的如何缓存上游服务器响应等功能。这些配置可以通过阅读 ngx_http_proxy_module 模块的说明了解，只有深入地理解，才能实现一个高性能的反向代理服务器。本节只是介绍反向代理服务器的基本功能，在第 12 章中我们将会深入地探索 upstream 机制，到那时，读者也许会发现 ngx_http_proxy_module 模块只是使用 upstream 机制实现了反向代理功能而已。

2.6 小结

Nginx 由少量的核心框架代码和许多模块组成，每个模块都有它独特的功能。因此，读者可以通过查看每个模块实现了什么功能，来了解 Nginx 可以帮我们做些什么。

Nginx 的 Wiki 网站（http://wiki.nginx.org/Modules）上列出了官方提供的所有模块及配置项，仔细观察就会发现，这些配置项的语法与本章的内容都是很相近的，读者只需要弄清楚模块说明中每个配置项的意义即可。另外，网页 http://wiki.nginx.org/3rdPartyModules 中列出了 Wiki 上已知的几十个第三方模块，同时读者还可以从搜索引擎上搜索到更多的第三方模块。了解每个模块的配置项用法，并在 Nginx 中使用这些模块，可以让 Nginx 做到更多。

随着对本书的学习，读者会对 Nginx 模块的设计思路有深入的了解，也会渐渐熟悉如何编写一个模块。如果某个模块的实现与你的想法有出入，可以更改这个模块的源码，实现你期望的业务功能。如果所有的模块都没有你想要的功能，不妨自己重写一个定制的模块，也可以申请发布到 Nginx 网站上供大家分享。

第二部分

如何编写 HTTP 模块

本部分内容

第 3 章　开发一个简单的 HTTP 模块

第 4 章　配置、error 日志和请求上下文

第 5 章　访问第三方服务

第 6 章　开发一个简单的 HTTP 过滤模块

第 7 章　Nginx 提供的高级数据结构

第 3 章　开发一个简单的 HTTP 模块

当通过开发 HTTP 模块来实现产品功能时，是可以完全享用 Nginx 的优秀设计所带来的、与官方模块相同的高并发特性的。不过，如何开发一个充满异步调用、无阻塞的 HTTP 模块呢？首先，需要把程序嵌入到 Nginx 中，也就是说，最终编译出的二进制程序 Nginx 要包含我们的代码（见 3.3 节）；其次，这个全新的 HTTP 模块要能介入到 HTTP 请求的处理流程中（具体参见 3.1 节、3.4 节、3.5 节）。满足上述两个前提后，我们的模块才能开始处理 HTTP 请求，但在开始处理请求前还需要先了解一些 Nginx 框架定义的数据结构（见 3.2 节），这是后面必须要用到的；正式处理请求时，还要可以获得 Nginx 框架接收、解析后的用户请求信息（见 3.6 节）；业务执行完毕后，则要考虑发送响应给用户（见 3.7 节），包括将磁盘中的文件以 HTTP 包体的形式发送给用户（见 3.8 节）。

本章最后会讨论如何用 C++ 语言来编写 HTTP 模块，这虽然不是 Nginx 官方倡导的方式，但 C++ 向前兼容 C 语言，使用 C++ 语言开发的模块还是可以很容易地嵌入到 Nginx 中。本章不会深入探讨 HTTP 模块与 Nginx 的各个核心模块是如何配合工作的，而且这部分提到的每个接口将只涉及用法而不涉及实现原理，在第 3 部分我们才会进一步阐述本章提到的许多接口是如何实现异步访问的。

3.1　如何调用 HTTP 模块

在开发 HTTP 模块前，首先需要了解典型的 HTTP 模块是如何介入 Nginx 处理用户请求流程的。图 3-1 是一个简化的时序图，这里省略了许多异步调用，忽略了多个不同的 HTTP 处理阶段，仅标识了在一个典型请求的处理过程中主要模块被调用的流程，以此帮助读者理解 HTTP 模块如何处理用户请求。完整的流程将在第 11 章中详细介绍。

从图 3-1 中看到，worker 进程会在一个 for 循环语句里反复调用事件模块检测网络事件。当事件模块检测到某个客户端发起的 TCP 请求时（接收到 SYN 包），将会为它建立 TCP 连接，成功建立连接后根据 nginx.conf 文件中的配置会交由 HTTP 框架处理。HTTP 框架会试图接收完整的 HTTP 头部，并在接收到完整的 HTTP 头部后将请求分发到具体的 HTTP 模块中处理。这种分发策略是多样化的，其中最常见的是根据请求的 URI 和 nginx.conf 里 location 配置项的匹配度来决定如何分发（本章的例子正是应用这种分发策略，在第 10 章中会介绍其他分发策略）。HTTP 模块在处理请求的结束时，大多会向客户端发送响应，此时会自动地依次调用所有的 HTTP 过滤模块，每个过滤模块可以根据配置文件决定自己的行为。

例如，gzip 过滤模块根据配置文件中的 gzip on|off 来决定是否压缩响应。HTTP 处理模块在返回时会将控制权交还给 HTTP 框架，如果在返回前设置了 subrequest，那么 HTTP 框架还会继续异步地调用适合的 HTTP 模块处理子请求。

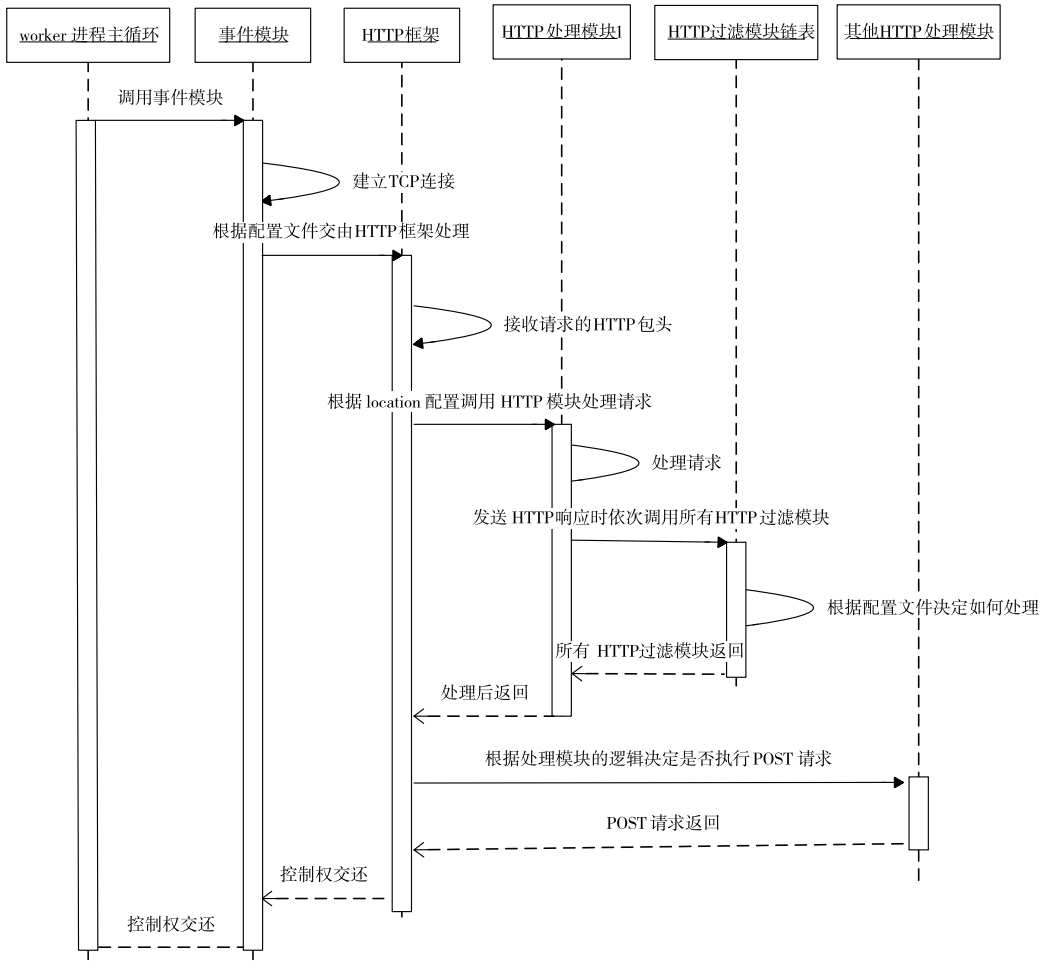


图 3-1　Nginx HTTP 模块调用的简化流程

开发 HTTP 模块时，首先要注意的就是 HTTP 框架到具体的 HTTP 模块间数据流的传递，以及开发的 HTTP 模块如何与诸多的过滤模块协同工作（第 10 章、第 11 章会详细介绍 HTTP 框架）。下面正式进入 HTTP 模块的开发环节。

3.2 准备工作

Nginx 模块需要使用 C（或者 C++）语言编写代码来实现，每个模块都要有自己的名字。按照 Nginx 约定俗成的命名规则，我们把第一个 HTTP 模块命名为 ngx_http_mytest_module。由于第一个模块非常简单，一个 C 源文件就可以完成，所以这里按照官方惯例，将唯一的源代码文件命名为 ngx_http_mytest_module.c。

实际上，我们还需要定义一个名称，以便在编译前的 configure 命令执行时显示是否执行成功（即 configure 脚本执行时的 ngx_addon_name 变量）。为方便理解，仍然使用同一个模块名来表示，如 ngx_http_mytest_module。

为了让 HTTP 模块正常工作，首先需要把它编译进 Nginx（3.3 节会探讨编译新增模块的两种方式）。其次需要设定模块如何在运行中生效，比如在图 3-1 描述的典型方式中，配置文件中的 location 块决定了匹配某种 URI 的请求将会由相应的 HTTP 模块处理，因此，运行时 HTTP 框架会在接收完毕 HTTP 请求的头部后，将请求的 URI 与配置文件中的所有 location 进行匹配（事实上会优先匹配虚拟主机，第 11 章会详细说明该流程），匹配后再根据 location {} 内的配置项选择 HTTP 模块来调用。这是一种最典型的 HTTP 模块调用方式。3.4 节将解释 HTTP 模块定义嵌入方式时用到的数据结构，3.5 节将定义我们的第一个 HTTP 模块，3.6 节中介绍如何使用上述模块调用方式来处理请求。

既然有典型的调用方式，自然也有非典型的调用方式，比如 ngx_http_access_module 模块，它是根据 IP 地址决定某个客户端是否可以访问服务的，因此，这个模块需要在 NGX_HTTP_ACCESS_PHASE 阶段（在第 10 章中会详述 HTTP 框架定义的 11 个阶段）生效，它会比本章介绍的 mytest 模块更早地介入请求的处理中，同时它的流程与图 3-1 中的不同，它可以对所有请求产生作用。也就是说，任何 HTTP 请求都会调用 ngx_http_access_module 模块处理，只是该模块会根据它感兴趣的配置项及所在的配置块来决定行为方式，这与 mytest 模块不同，在 mytest 模块中，只有在配置了 location /uri {mytest;} 后，HTTP 框架才会在某个请求匹配了 /uri 后调用它处理请求。如果某个匹配了 URI 请求的 location 中没有配置 mytest 配置项，mytest 模块依然是不会被调用的。

为了做到跨平台，Nginx 定义、封装了一些基本的数据结构。由于 Nginx 对内存分配比较“吝啬”（只有保证低内存消耗，才可能实现十万甚至百万级别的同时并发连接数），所以这些 Nginx 数据结构天生都是尽可能少占用内存。下面介绍本章中将要用到的 Nginx 定义的几个基本数据结构和方法，在第 7 章还会介绍一些复杂的容器，读者可以从中体会到如何才能有效地利用内存。

3.2.1 整型的封装

Nginx 使用 ngx_int_t 封装有符号整型，使用 ngx_uint_t 封装无符号整型。Nginx 各模块

的变量定义都是如此使用的，建议读者沿用 Nginx 的习惯，以此替代 int 和 unsinged int。

在 Linux 平台下，Nginx 对 ngx_int_t 和 ngx_uint_t 的定义如下：

```
typedef intptr_t        ngx_int_t;
typedef uintptr_t       ngx_uint_t;
```

3.2.2　ngx_str_t 数据结构

在 Nginx 的领域中，ngx_str_t 结构就是字符串。ngx_str_t 的定义如下：

```
typedef struct {
    size_t      len;
    u_char     *data;
} ngx_str_t;
```

ngx_str_t 只有两个成员，其中 data 指针指向字符串起始地址，len 表示字符串的有效长度。注意，ngx_str_t 的 data 成员指向的并不是普通的字符串，因为这段字符串未必会以 '\0' 作为结尾，所以使用时必须根据长度 len 来使用 data 成员。例如，在 3.7.2 节中，我们会看到 r->method_name 就是一个 ngx_str_t 类型的变量，比较 method_name 时必须如下这样使用：

```
if (0 == ngx_strncmp(
               r->method_name.data,
               "PUT",
               r->method_name.len)
       )
{...}
```

这里，ngx_strncmp 其实就是 strncmp 函数，为了跨平台 Nginx 习惯性地对其进行了名称上的封装，下面看一下它的定义：

```
#define ngx_strncmp(s1, s2, n)  strncmp((const char *) s1, (const char *) s2, n)
```

任何试图将 ngx_str_t 的 data 成员当做字符串来使用的情况，都可能导致内存越界！Nginx 使用 ngx_str_t 可以有效地降低内存使用量。例如，用户请求“GET /test?a=1 http/1.1\r\n”存储到内存地址 0x1d0b0110 上，这时只需要把 r->method_name 设置为 {len = 3, data = 0x1d0b0110} 就可以表示方法名“GET”，而不需要单独为 method_name 再分配内存冗余的存储字符串。

3.2.3　ngx_list_t 数据结构

ngx_list_t 是 Nginx 封装的链表容器，它在 Nginx 中使用得很频繁，例如 HTTP 的头部就是用 ngx_list_t 来存储的。当然，C 语言封装的链表没有 C++ 或 Java 等面向对象语言那么容易理解。先看一下 ngx_list_t 相关成员的定义：

```
typedef struct ngx_list_part_s  ngx_list_part_t;
```

```
struct ngx_list_part_s {
    void             *elts;
    ngx_uint_t        nelts;
    ngx_list_part_t  *next;
};

typedef struct {
    ngx_list_part_t  *last;
    ngx_list_part_t   part;
    size_t            size;
    ngx_uint_t        nalloc;
    ngx_pool_t       *pool;
} ngx_list_t;
```

ngx_list_t 描述整个链表，而 ngx_list_part_t 只描述链表的一个元素。这里要注意的是，ngx_list_t 不是一个单纯的链表，为了便于理解，我们姑且称它为存储数组的链表，什么意思呢？抽象地说，就是每个链表元素 ngx_list_part_t 又是一个数组，拥有连续的内存，它既依赖于 ngx_list_t 里的 size 和 nalloc 来表示数组的容量，同时又依靠每个 ngx_list_part_t 成员中的 nelts 来表示数组当前已使用了多少容量。因此，ngx_list_t 是一个链表容器，而链表中的元素又是一个数组。事实上，ngx_list_part_t 数组中的元素才是用户想要存储的东西，ngx_list_t 链表能够容纳的元素数量由 ngx_list_part_t 数组元素的个数与每个数组所能容纳的元素相乘得到。

这样设计有什么好处呢？

- ❑ 链表中存储的元素是灵活的，它可以是任何一种数据结构。
- ❑ 链表元素需要占用的内存由 ngx_list_t 管理，它已经通过数组分配好了。
- ❑ 小块的内存使用链表访问效率是低下的，使用数组通过偏移量来直接访问内存则要高效得多。

下面详述每个成员的意义。

（1）ngx_list_t

- ❑ part：链表的首个数组元素。
- ❑ last：指向链表的最后一个数组元素。
- ❑ size：前面讲过，链表中的每个 ngx_list_part_t 元素都是一个数组。因为数组存储的是某种类型的数据结构，且 ngx_list_t 是非常灵活的数据结构，所以它不会限制存储什么样的数据，只是通过 size 限制每一个数组元素的占用的空间大小，也就是用户要存储的一个数据所占用的字节数必须小于或等于 size。
- ❑ nalloc：链表的数组元素一旦分配后是不可更改的。nalloc 表示每个 ngx_list_part_t 数组的容量，即最多可存储多少个数据。
- ❑ pool：链表中管理内存分配的内存池对象。用户要存放的数据占用的内存都是由 pool 分配的，下文中会详细介绍。

（2）ngx_list_part_t

❑ elts：指向数组的起始地址。

❑ nelts：表示数组中已经使用了多少个元素。当然，nelts 必须小于 ngx_list_t 结构体中的 nalloc。

❑ next：下一个链表元素 ngx_list_part_t 的地址。

事实上，ngx_list_t 中的所有数据都是由 ngx_pool_t 类型的 pool 内存池分配的，它们通常都是连续的内存（在由一个 pool 内存池分配的情况下）。下面以图 3-2 为例来看一下 ngx_list_t 的内存分布情况。

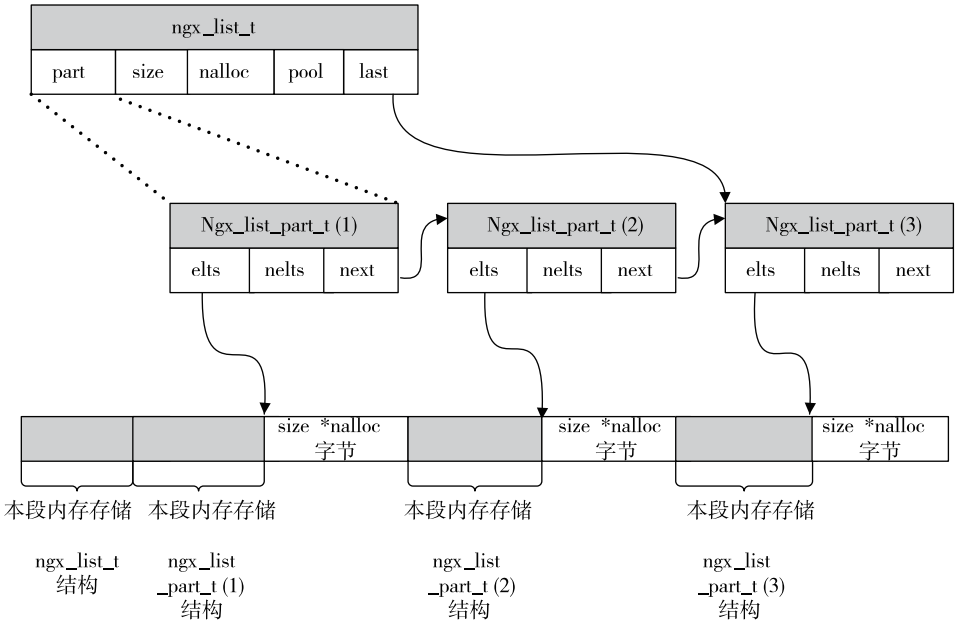


图 3-2　ngx_list_t 的内存分布

图 3-2 中是由 3 个 ngx_list_part_t 数组元素组成的 ngx_list_t 链表可能拥有的一种内存分布结构，读者可以从这种较为常见的内存分布中看到 ngx_list_t 链表的用法。这里，pool 内存池为其分配了连续的内存，最前端内存存储的是 ngx_list_t 结构中的成员，紧接着是第一个 ngx_list_part_t 结构占用的内存，然后是 ngx_list_part_t 结构指向的数组，它们一共占用 size*nalloc 字节，表示数组中拥有 nalloc 个大小为 size 的元素。其后面是第 2 个 ngx_list_part_t 结构以及它所指向的数组，依此类推。

对于链表，Nginx 提供的接口包括：ngx_list_create 接口用于创建新的链表，ngx_list_init 接口用于初始化一个已有的链表，ngx_list_push 接口用于添加新的元素，如下所示：

```
ngx_list_t *ngx_list_create(ngx_pool_t *pool, ngx_uint_t n, size_t size);
```

```
static ngx_inline ngx_int_t
ngx_list_init(ngx_list_t *list, ngx_pool_t *pool, ngx_uint_t n, size_t size);


void *ngx_list_push(ngx_list_t *list);
```

调用 ngx_list_create 创建元素时，pool 参数是内存池对象（参见 3.7.2 节），size 是每个元素的大小，n 是每个链表数组可容纳元素的个数（相当于 ngx_list_t 结构中的 nalloc 成员）。ngx_list_create 返回新创建的链表地址，如果创建失败，则返回 NULL 空指针。ngx_list_create 被调用后至少会创建一个数组（不会创建空链表），其中包含 n 个大小为 size 字节的连续内存块，也就是 ngx_list_t 结构中的 part 成员。

下面看一个简单的例子，我们首先建立一个链表，它存储的元素是 ngx_str_t，其中每个链表数组中存储 4 个元素，代码如下所示：

```
ngx_list_t* testlist = ngx_list_create(r->pool, 4,sizeof(ngx_str_t));
if (testlist == NULL) {
    return NGX_ERROR;
}
```

ngx_list_init 的使用方法与 ngx_list_create 非常类似，需要注意的是，这时链表数据结构已经创建好了，若 ngx_list_init 返回 NGX_OK，则表示初始化成功，若返回 NGX_ERROR，则表示失败。

调用 ngx_list_push 表示添加新的元素，传入的参数是 ngx_list_t 链表。正常情况下，返回的是新分配的元素首地址。如果返回 NULL 空指针，则表示添加失败。在使用它时通常先调用 ngx_list_push 得到返回的元素地址，再对返回的地址进行赋值。例如：

```
ngx_str_t* str = ngx_list_push(testlist);
if (str == NULL) {
   return NGX_ERROR;
}

str->len= sizeof("Hello world");
str->value = "Hello world";
```

遍历链表时 Nginx 没有提供相应的接口，实际上也不需要。我们可以用以下方法遍历链表中的元素：

```
//part 用于指向链表中的每一个 ngx_list_part_t 数组
ngx_list_part_t* part = &testlist.part;

// 根据链表中的数据类型，把数组里的 elts 转化为该类型使用
ngx_str_t* str = part->elts;

//i 表示元素在链表的每个 ngx_list_part_t 数组里的序号
for (i = 0; /* void */; i++) {
```

```
        if (i >= part->nelts) {
                if (part->next == NULL) {
                        // 如果某个 ngx_list_part_t 数组的 next 指针为空，
                        // 则说明已经遍历完链表
                        break;
                }

                // 访问下一个 ngx_list_part_t
                part = part->next;
                header = part->elts;

                // 将 i 序号置为 0，准备重新访问下一个数组
                i = 0;
        }

        // 这里可以很方便地取到当前遍历到的链表元素
        printf("list element: %*s\n",str[i].len, str[i].data);
}
```

3.2.4　ngx_table_elt_t 数据结构

ngx_table_elt_t 数据结构如下所示：

```
typedef struct {
    ngx_uint_t        hash;
    ngx_str_t         key;
    ngx_str_t         value;
    u_char           *lowcase_key;
} ngx_table_elt_t;
```

可以看到，ngx_table_elt_t 就是一个 key/value 对，ngx_str_t 类型的 key、value 成员分别存储的是名字、值字符串。hash 成员表明 ngx_table_elt_t 也可以是某个散列表数据结构（ngx_hash_t 类型）中的成员。ngx_uint_t 类型的 hash 成员可以在 ngx_hash_t 中更快地找到相同 key 的 ngx_table_elt_t 数据。lowcase_key 指向的是全小写的 key 字符串。

显而易见，ngx_table_elt_t 是为 HTTP 头部“量身订制”的，其中 key 存储头部名称（如 Content-Length），value 存储对应的值（如“1024”），lowcase_key 是为了忽略 HTTP 头部名称的大小写（例如，有些客户端发来的 HTTP 请求头部是 content-length，Nginx 希望它与大小写敏感的 Content-Length 做相同处理，有了全小写的 lowcase_key 成员后就可以快速达成目的了），hash 用于快速检索头部（它的用法在 3.6.3 节中进行详述）。

3.2.5　ngx_buf_t 数据结构

缓冲区 ngx_buf_t 是 Nginx 处理大数据的关键数据结构，它既应用于内存数据也应用于磁盘数据。下面主要介绍 ngx_buf_t 结构体本身，而描述磁盘文件的 ngx_file_t 结构体则在 3.8.1 节中说明。下面来看一下相关代码：

```
typedef struct ngx_buf_s  ngx_buf_t;
typedef void *            ngx_buf_tag_t;
struct ngx_buf_s {
    /*pos 通常是用来告诉使用者本次应该从 pos 这个位置开始处理内存中的数据，这样设置是因为同一个
ngx_buf_t 可能被多次反复处理。当然，pos 的含义是由使用它的模块定义的 */
    u_char          *pos;
    /*last 通常表示有效的内容到此为止，注意，pos 与 last 之间的内存是希望 nginx 处理的内容 */
    u_char          *last;
    /* 处理文件时，file_pos 与 file_last 的含义与处理内存时的 pos 与 last 相同，file_pos 表示将要
处理的文件位置，file_last 表示截止的文件位置 */
    off_t            file_pos;
    off_t            file_last;

    // 如果 ngx_buf_t 缓冲区用于内存，那么 start 指向这段内存的起始地址
    u_char          *start;
    // 与 start 成员对应，指向缓冲区内存的末尾
    u_char          *end;
    /* 表示当前缓冲区的类型，例如由哪个模块使用就指向这个模块 ngx_module_t 变量的地址 */
    ngx_buf_tag_t    tag;
    // 引用的文件
    ngx_file_t      *file;
    /* 当前缓冲区的影子缓冲区，该成员很少用到，仅仅在 12.8 节描述的使用缓冲区转发上游服务器的响应
时才使用了 shadow 成员，这是因为 Nginx 太节约内存了，分配一块内存并使用 ngx_buf_t 表示接收到的上游服务
器响应后，在向下游客户端转发时可能会把这块内存存储到文件中，也可能直接向下游发送，此时 Nginx 绝不会重新
复制一份内存用于新的目的，而是再次建立一个 ngx_buf_t 结构体指向原内存，这样多个 ngx_buf_t 结构体指向
了同一块内存，它们之间的关系就通过 shadow 成员来引用。这种设计过于复杂，通常不建议使用 */
    ngx_buf_t       *shadow;

    // 临时内存标志位，为 1 时表示数据在内存中且这段内存可以修改
    unsigned         temporary:1;

    // 标志位，为 1 时表示数据在内存中且这段内存不可以被修改
    unsigned         memory:1;

    // 标志位，为 1 时表示这段内存是用 mmap 系统调用映射过来的，不可以被修改
    unsigned         mmap:1;

    // 标志位，为 1 时表示可回收
    unsigned         recycled:1;
    // 标志位，为 1 时表示这段缓冲区处理的是文件而不是内存
    unsigned         in_file:1;
    // 标志位，为 1 时表示需要执行 flush 操作
    unsigned         flush:1;
    /* 标志位，对于操作这块缓冲区时是否使用同步方式，需谨慎考虑，这可能会阻塞 Nginx 进程，Nginx
中所有操作几乎都是异步的，这是它支持高并发的关键。有些框架代码在 sync 为 1 时可能会有阻塞的方式进行 I/O
操作，它的意义视使用它的 Nginx 模块而定 */
    unsigned         sync:1;
    /* 标志位，表示是否是最后一块缓冲区，因为 ngx_buf_t 可以由 ngx_chain_t 链表串联起来，因此，
当 last_buf 为 1 时，表示当前是最后一块待处理的缓冲区 */
    unsigned         last_buf:1;
```

```
    // 标志位，表示是否是 ngx_chain_t 中的最后一块缓冲区
    unsigned         last_in_chain:1;
    /* 标志位，表示是否是最后一个影子缓冲区，与 shadow 域配合使用。通常不建议使用它 */
    unsigned         last_shadow:1;
    // 标志位，表示当前缓冲区是否属于临时文件
    unsigned         temp_file:1;
};
```

关于使用 ngx_buf_t 的案例参见 3.7.2 节。ngx_buf_t 是一种基本数据结构，本质上它提供的仅仅是一些指针成员和标志位。对于 HTTP 模块来说，需要注意 HTTP 框架、事件框架是如何设置和使用 pos、last 等指针以及如何处理这些标志位的，上述说明只是最常见的用法。（如果我们自定义一个 ngx_buf_t 结构体，不应当受限于上述用法，而应该根据业务需求自行定义。例如，在 13.7 节中用一个 ngx_buf_t 缓冲区转发上下游 TCP 流时，pos 会指向将要发送到下游的 TCP 流起始地址，而 last 会指向预备接收上游 TCP 流的缓冲区起始地址。）

3.2.6　ngx_chain_t 数据结构

ngx_chain_t 是与 ngx_buf_t 配合使用的链表数据结构，下面看一下它的定义：

```
typedef struct ngx_chain_s       ngx_chain_t;
struct ngx_chain_s {
    ngx_buf_t    *buf;
    ngx_chain_t  *next;
};
```

buf 指向当前的 ngx_buf_t 缓冲区，next 则用来指向下一个 ngx_chain_t。如果这是最后一个 ngx_chain_t，则需要把 next 置为 NULL。

在向用户发送 HTTP 包体时，就要传入 ngx_chain_t 链表对象，注意，如果是最后一个 ngx_chain_t，那么必须将 next 置为 NULL，否则永远不会发送成功，而且这个请求将一直不会结束（Nginx 框架的要求）。

3.3　如何将自己的 HTTP 模块编译进 Nginx

Nginx 提供了一种简单的方式将第三方的模块编译到 Nginx 中。首先把源代码文件全部放到一个目录下，同时在该目录中编写一个文件用于通知 Nginx 如何编译本模块，这个文件名必须为 config。它的格式将在 3.3.1 节中说明。

这样，只要在 configure 脚本执行时加入参数 --add-module=PATH（PATH 就是上面我们给定的源代码、config 文件的保存目录），就可以在执行正常编译安装流程时完成 Nginx 编译工作。

有时，Nginx 提供的这种方式可能无法满足我们的需求，其实，在执行完 configure 脚

本后 Nginx 会生成 objs/Makefile 和 objs/ngx_modules.c 文件，完全可以自己去修改这两个文件，这是一种更强大也复杂得多的方法，我们将在 3.3.3 节中说明如何直接修改它们。

3.3.1 config 文件的写法

config 文件其实是一个可执行的 Shell 脚本。如果只想开发一个 HTTP 模块，那么 config 文件中需要定义以下 3 个变量：

- ngx_addon_name：仅在 configure 执行时使用，一般设置为模块名称。
- HTTP_MODULES：保存所有的 HTTP 模块名称，每个 HTTP 模块间由空格符相连。在重新设置 HTTP_MODULES 变量时，不要直接覆盖它，因为 configure 调用到自定义的 config 脚本前，已经将各个 HTTP 模块设置到 HTTP_MODULES 变量中了，因此，要像如下这样设置：

```
"$HTTP_MODULES ngx_http_mytest_module"
```

- NGX_ADDON_SRCS：用于指定新增模块的源代码，多个待编译的源代码间以空格符相连。注意，在设置 NGX_ADDON_SRCS 时可以使用 $ngx_addon_dir 变量，它等价于 configure 执行时 --add-module=PATH 的 PATH 参数。

因此，对于 mytest 模块，可以这样编写 config 文件：

```
ngx_addon_name=ngx_http_mytest_module
HTTP_MODULES="$HTTP_MODULES ngx_http_mytest_module"
NGX_ADDON_SRCS="$NGX_ADDON_SRCS $ngx_addon_dir/ngx_http_mytest_module.c"
```

注意 以上 3 个变量并不是唯一可以在 config 文件中自定义的部分。如果我们不是开发 HTTP 模块，而是开发一个 HTTP 过滤模块，那么就要用 HTTP_FILTER_MODULES 替代上面的 HTTP_MODULES 变量。事实上，包括 $CORE_MODULES、$EVENT_MODULES、$HTTP_MODULES、$HTTP_FILTER_MODULES、$HTTP_HEADERS_FILTER_MODULE 等模块变量都可以重定义，它们分别对应着 Nginx 的核心模块、事件模块、HTTP 模块、HTTP 过滤模块、HTTP 头部过滤模块。除了 NGX_ADDON_SRCS 变量，或许还有一个变量我们会用到，即 $NGX_ADDON_DEPS 变量，它指定了模块依赖的路径，同样可以在 config 中设置。

3.3.2 利用 configure 脚本将定制的模块加入到 Nginx 中

在 1.6 节提到的 configure 执行流程中，其中有两行脚本负责将第三方模块加入到 Nginx 中，如下所示。

```
. auto/modules
```

```
. auto/make
```

下面完整地解释一下 configure 脚本是如何与 3.3.1 节中提到的 config 文件配合起来把定制的第三方模块加入到 Nginx 中的。

在执行 configure --add-module=PATH 命令时，PATH 就是第三方模块所在的路径。在 configure 中，通过 auto/options 脚本设置了 NGX_ADDONS 变量：

```
--add-module=*)                  NGX_ADDONS="$NGX_ADDONS $value" ;;
```

在 configure 命令执行到 auto/modules 脚本时，将在生成的 ngx_modules.c 文件中加入定制的第三方模块。

```
if test -n "$NGX_ADDONS"; then

    echo configuring additional modules

    for ngx_addon_dir in $NGX_ADDONS
    do
        echo "adding module in $ngx_addon_dir"

        if test -f $ngx_addon_dir/config; then
            # 在这里执行自定义的 config 脚本
            . $ngx_addon_dir/config

            echo " + $ngx_addon_name was configured"

        else
            echo "$0: error: no $ngx_addon_dir/config was found"
            exit 1
        fi
    done
fi
```

可以看到，$NGX_ADDONS 可以包含多个目录，对于每个目录，如果其中存在 config 文件就会执行，也就是说，在 config 中重新定义的变量都会生效。之后，auto/modules 脚本开始创建 ngx_modules.c 文件，这个文件的关键点就是定义了 ngx_module_t *ngx_modules[] 数组，这个数组存储了 Nginx 中的所有模块。Nginx 在初始化、处理请求时，都会循环访问 ngx_modules 数组，确定该用哪一个模块来处理。下面来看一下 auto/modules 是如何生成数组的，代码如下所示：

```
modules="$CORE_MODULES $EVENT_MODULES"

if [ $USE_OPENSSL = YES ]; then
    modules="$modules $OPENSSL_MODULE"
    CORE_DEPS="$CORE_DEPS $OPENSSL_DEPS"
    CORE_SRCS="$CORE_SRCS $OPENSSL_SRCS"
```

```
fi

if [ $HTTP = YES ]; then
    modules="$modules $HTTP_MODULES $HTTP_FILTER_MODULES \
             $HTTP_HEADERS_FILTER_MODULE \
             $HTTP_AUX_FILTER_MODULES \
             $HTTP_COPY_FILTER_MODULE \
             $HTTP_RANGE_BODY_FILTER_MODULE \
             $HTTP_NOT_MODIFIED_FILTER_MODULE"

    NGX_ADDON_DEPS="$NGX_ADDON_DEPS \$(HTTP_DEPS)"
fi
```

首先，auto/modules 会按顺序生成 modules 变量。注意，这里的 $HTTP_MODULES 等已经在 config 文件中重定义了。这时，modules 变量是包含所有模块的。然后，开始生成 ngx_modules.c 文件：

```
cat << END                                        > $NGX_MODULES_C

#include <ngx_config.h>
#include <ngx_core.h>

$NGX_PRAGMA

END

for mod in $modules
do
    echo "extern ngx_module_t  $mod;"           >> $NGX_MODULES_C
done

echo                                              >> $NGX_MODULES_C
echo 'ngx_module_t *ngx_modules[] = {'            >> $NGX_MODULES_C

for mod in $modules
do
    # 向 ngx_modules 数组里添加 Nginx 模块
    echo "    &$mod,"                             >> $NGX_MODULES_C
done

cat << END                                        >> $NGX_MODULES_C
    NULL
};

END
```

这样就已经确定了 Nginx 在运行时会调用自定义的模块，而 auto/make 脚本负责把相关模块编译进 Nginx。

在 Makefile 中生成编译第三方模块的源代码如下：

```
if test -n "$NGX_ADDON_SRCS"; then

    ngx_cc="\$(CC) $ngx_compile_opt \$(CFLAGS) $ngx_use_pch \$(ALL_INCS)"

    for ngx_src in $NGX_ADDON_SRCS
    do
        ngx_obj="addon/`basename \`dirname $ngx_src\``"

        ngx_obj=`echo $ngx_obj/\`basename $ngx_src\` \
            | sed -e "s/\//$ngx_regex_dirsep/g"`

        ngx_obj=`echo $ngx_obj \
            | sed -e
              "s#^\(.*\.\)cpp\\$#$ngx_objs_dir\1$ngx_objext#g" \
                  -e
              "s#^\(.*\.\)cc\\$#$ngx_objs_dir\1$ngx_objext#g" \
                  -e
              "s#^\(.*\.\)c\\$#$ngx_objs_dir\1$ngx_objext#g" \
                  -e
              "s#^\(.*\.\)S\\$#$ngx_objs_dir\1$ngx_objext#g"`

        ngx_src=`echo $ngx_src | sed -e "s/\//$ngx_regex_dirsep/g"`

        cat << END                                              >> $NGX_MAKEFILE

$ngx_obj:  \$(ADDON_DEPS)$ngx_cont$ngx_src
   $ngx_cc$ngx_tab$ngx_objout$ngx_obj$ngx_tab$ngx_src$NGX_AUX

END
     done

fi
```

下面这段代码用于将各个模块的目标文件设置到 ngx_obj 变量中，紧接着会生成 Makefile 里的链接代码，并将所有的目标文件、库文件链接成二进制程序。

```
for ngx_src in $NGX_ADDON_SRCS
do
    ngx_obj="addon/`basename \`dirname $ngx_src\``"

    test -d $NGX_OBJS/$ngx_obj || mkdir -p $NGX_OBJS/$ngx_obj

    ngx_obj=`echo $ngx_obj/\`basename $ngx_src\` \
        | sed -e "s/\//$ngx_regex_dirsep/g"`

    ngx_all_srcs="$ngx_all_srcs $ngx_obj"
done
```

```
...

cat << END                                                    >> $NGX_MAKEFILE

$NGX_OBJS${ngx_dirsep}nginx${ngx_binext}:
    $ngx_deps$ngx_spacer \$(LINK)
    ${ngx_long_start}${ngx_binout}$NGX_OBJS${ngx_dirsep}nginx$ngx_long_cont$ngx
_objs$ngx_libs$ngx_link
    $ngx_rcc
${ngx_long_end}
END
```

综上可知，第三方模块就是这样嵌入到 Nginx 程序中的。

3.3.3 直接修改 Makefile 文件

3.3.2 节中介绍的方法毫无疑问是最方便的，因为大量的工作已由 Nginx 中的 configure 脚本帮我们做好了。在使用其他第三方模块时，一般也推荐使用该方法。

我们有时可能需要更灵活的方式，比如重新决定 ngx_module_t *ngx_modules[] 数组中各个模块的顺序，或者在编译源代码时需要加入一些独特的编译选项，那么可以在执行完 configure 后，对生成的 objs/ngx_modules.c 和 objs/Makefile 文件直接进行修改。

在修改 objs/ngx_modules.c 时，首先要添加新增的第三方模块的声明，如下所示。

```
extern ngx_module_t  ngx_http_mytest_module;
```

其次，在合适的地方将模块加入到 ngx_modules 数组中。

```
ngx_module_t *ngx_modules[] = {
  ...
  &ngx_http_upstream_ip_hash_module,
  &ngx_http_mytest_module,
  &ngx_http_write_filter_module,
   ...
   NULL
};
```

注意，模块的顺序很重要。如果同时有两个模块表示对同一个请求感兴趣，那么只有顺序在前的模块会被调用。

修改 objs/Makefile 时需要增加编译源代码的部分，例如：

```
objs/addon/httpmodule/ngx_http_mytest_module.o: $(ADDON_DEPS) \
        ../sample/httpmodule//ngx_http_mytest_module.c
        $(CC) -c $(CFLAGS)  $(ALL_INCS) \
                -o objs/addon/httpmodule/ngx_http_mytest_module.o \
                ../sample/httpmodule//ngx_http_mytest_module.c
```

还需要把目标文件链接到 Nginx 中，例如：

```
objs/nginx:     objs/src/core/nginx.o \
...
     objs/addon/httpmodule/ngx_http_mytest_module.o \
  objs/ngx_modules.o

    $(LINK) -o objs/nginx \
    objs/src/core/nginx.o \
    ...
    objs/addon/httpmodule/ngx_http_mytest_module.o \
  objs/ngx_modules.o \
  -lpthread -lcrypt -lpcre -lcrypto -lcrypto -lz
```

请慎用这种直接修改 Makefile 和 ngx_modules.c 的方法，不正确的修改可能导致 Nginx 工作不正常。

3.4　HTTP 模块的数据结构

定义 HTTP 模块方式很简单，例如：

```
ngx_module_t ngx_http_mytest_module;
```

其中，ngx_module_t 是一个 Nginx 模块的数据结构（详见 8.2 节）。下面来分析一下 Nginx 模块中所有的成员，如下所示：

```
typedef struct ngx_module_s      ngx_module_t;
struct ngx_module_s {
    /* 下面的 ctx_index、index、spare0、spare1、spare2、spare3、version 变量不需要在定义时赋值，
可以用 Nginx 准备好的宏 NGX_MODULE_V1 来定义，它已经定义好了这 7 个值。
#define NGX_MODULE_V1           0, 0, 0, 0, 0, 0, 1

    对于一类模块（由下面的 type 成员决定类别）而言，ctx_index 表示当前模块在这类模块中的序号。这
个成员常常是由管理这类模块的一个 Nginx 核心模块设置的，对于所有的 HTTP 模块而言，ctx_index 是由核心模
块 ngx_http_module 设置的。ctx_index 非常重要，Nginx 的模块化设计非常依赖于各个模块的顺序，它们既用
于表达优先级，也用于表明每个模块的位置，借以帮助 Nginx 框架快速获得某个模块的数据（HTTP 框架设置 ctx_
index 的过程参见 10.7 节）*/
        ngx_uint_t             ctx_index;

     /*index 表示当前模块在 ngx_modules 数组中的序号。注意，ctx_index 表示的是当前模块在一类模
块中的序号，而 index 表示当前模块在所有模块中的序号，它同样关键。Nginx 启动时会根据 ngx_modules 数组
设置各模块的 index 值。例如：
    ngx_max_module = 0;
    for (i = 0; ngx_modules[i]; i++) {
        ngx_modules[i]->index = ngx_max_module++;
    }
    */
        ngx_uint_t             index;
```

```
    //spare 系列的保留变量，暂未使用
    ngx_uint_t             spare0;
    ngx_uint_t             spare1;
    ngx_uint_t             spare2;
    ngx_uint_t             spare3;
    // 模块的版本，便于将来的扩展。目前只有一种，默认为 1
    ngx_uint_t             version;

     /*ctx 用于指向一类模块的上下文结构体，为什么需要 ctx 呢？因为前面说过，Nginx 模块有许多种类，
不同类模块之间的功能差别很大。例如，事件类型的模块主要处理 I/O 事件相关的功能，HTTP 类型的模块主要处理
HTTP 应用层的功能。这样，每个模块都有了自己的特性，而 ctx 将会指向特定类型模块的公共接口。例如，在 HTTP
模块中，ctx 需要指向 ngx_http_module_t 结构体 */
    void                  *ctx;

    //commands 将处理 nginx.conf 中的配置项，详见第 4 章
    ngx_command_t         *commands;

     /*type 表示该模块的类型，它与 ctx 指针是紧密相关的。在官方 Nginx 中，它的取值范围是以下 5 种：
NGX_HTTP_MODULE、NGX_CORE_MODULE、NGX_CONF_MODULE、NGX_EVENT_MODULE、NGX_MAIL_MODULE。这
5 种模块间的关系参考图 8-2。实际上，还可以自定义新的模块类型 */
    ngx_uint_t             type;

     /* 在 Nginx 的启动、停止过程中，以下 7 个函数指针表示有 7 个执行点会分别调用这 7 种方法（参见
8.4 节～ 8.6 节）。对于任一个方法而言，如果不需要 Nginx 在某个时刻执行它，那么简单地把它设为 NULL 空指针
即可 */

     /* 虽然从字面上理解应当在 master 进程启动时回调 init_master，但到目前为止，框架代码从来不会
调用它，因此，可将 init_master 设为 NULL */
    ngx_int_t             (*init_master)(ngx_log_t *log);
     /*init_module 回调方法在初始化所有模块时被调用。在 master/worker 模式下，这个阶段将在启动
worker 子进程前完成 */
    ngx_int_t             (*init_module)(ngx_cycle_t *cycle);
  /* init_process 回调方法在正常服务前被调用。在 master/worker 模式下，多个 worker 子进程已经产
生，在每个 worker 进程的初始化过程会调用所有模块的 init_process 函数 */
    ngx_int_t             (*init_process)(ngx_cycle_t *cycle);
  /* 由于 Nginx 暂不支持多线程模式，所以 init_thread 在框架代码中没有被调用过，设为 NULL*/
    ngx_int_t             (*init_thread)(ngx_cycle_t *cycle);
  // 同上，exit_thread 也不支持，设为 NULL
    void                  (*exit_thread)(ngx_cycle_t *cycle);
  /* exit_process 回调方法在服务停止前调用。在 master/worker 模式下，worker 进程会在退出前调用它 */
    void                  (*exit_process)(ngx_cycle_t *cycle);
  // exit_master 回调方法将在 master 进程退出前被调用
    void                  (*exit_master)(ngx_cycle_t *cycle);

     /* 以下 8 个 spare_hook 变量也是保留字段，目前没有使用，但可用 Nginx 提供的 NGX_MODULE_V1_
PADDING 宏来填充。看一下该宏的定义：#define NGX_MODULE_V1_PADDING  0, 0, 0, 0, 0, 0, 0, 0*/
    uintptr_t             spare_hook0;
    uintptr_t             spare_hook1;
    uintptr_t             spare_hook2;
```

```
    uintptr_t             spare_hook3;
    uintptr_t             spare_hook4;
    uintptr_t             spare_hook5;
    uintptr_t             spare_hook6;
    uintptr_t             spare_hook7;
};
```

定义一个 HTTP 模块时，务必把 type 字段设为 NGX_HTTP_MODULE。

对于下列回调方法：init_module、init_process、exit_process、exit_master，调用它们的是 Nginx 的框架代码。换句话说，这 4 个回调方法与 HTTP 框架无关，即使 nginx.conf 中没有配置 http {...} 这种开启 HTTP 功能的配置项，这些回调方法仍然会被调用。因此，通常开发 HTTP 模块时都把它们设为 NULL 空指针。这样，当 Nginx 不作为 Web 服务器使用时，不会执行 HTTP 模块的任何代码。

定义 HTTP 模块时，最重要的是要设置 ctx 和 commands 这两个成员。对于 HTTP 类型的模块来说，ngx_module_t 中的 ctx 指针必须指向 ngx_http_module_t 接口（HTTP 框架的要求）。下面先来分析 ngx_http_module_t 结构体的成员。

HTTP 框架在读取、重载配置文件时定义了由 ngx_http_module_t 接口描述的 8 个阶段，HTTP 框架在启动过程中会在每个阶段中调用 ngx_http_module_t 中相应的方法。当然，如果 ngx_http_module_t 中的某个回调方法设为 NULL 空指针，那么 HTTP 框架是不会调用它的。

```
typedef struct {
    // 解析配置文件前调用
    ngx_int_t   (*preconfiguration)(ngx_conf_t *cf);
    // 完成配置文件的解析后调用
    ngx_int_t   (*postconfiguration)(ngx_conf_t *cf);

     /* 当需要创建数据结构用于存储 main 级别（直属于 http{...} 块的配置项）的全局配置项时，可以通过 create_main_conf 回调方法创建存储全局配置项的结构体 */
    void        *(*create_main_conf)(ngx_conf_t *cf);
    // 常用于初始化 main 级别配置项
    char        *(*init_main_conf)(ngx_conf_t *cf, void *conf);

      /* 当需要创建数据结构用于存储 srv 级别（直属于虚拟主机 server{...} 块的配置项）的配置项时，可以通过实现 create_srv_conf 回调方法创建存储 srv 级别配置项的结构体 */
    void        *(*create_srv_conf)(ngx_conf_t *cf);
    // merge_srv_conf 回调方法主要用于合并 main 级别和 srv 级别下的同名配置项
    char        *(*merge_srv_conf)(ngx_conf_t *cf, void *prev, void *conf);

     /* 当需要创建数据结构用于存储 loc 级别（直属于 location{...} 块的配置项）的配置项时，可以实现 create_loc_conf 回调方法 */
    void        *(*create_loc_conf)(ngx_conf_t *cf);
    // merge_loc_conf 回调方法主要用于合并 srv 级别和 loc 级别下的同名配置项
    char        *(*merge_loc_conf)(ngx_conf_t *cf, void *prev, void *conf);
} ngx_http_module_t;
```

不过，这 8 个阶段的调用顺序与上述定义的顺序是不同的。在 Nginx 启动过程中，

HTTP 框架调用这些回调方法的实际顺序有可能是这样的（与 nginx.conf 配置项有关）：

1）create_main_conf

2）create_srv_conf

3）create_loc_conf

4）preconfiguration

5）init_main_conf

6）merge_srv_conf

7）merge_loc_conf

8）postconfiguration

commands 数组用于定义模块的配置文件参数，每一个数组元素都是 ngx_command_t 类型，数组的结尾用 ngx_null_command 表示。Nginx 在解析配置文件中的一个配置项时首先会遍历所有的模块，对于每一个模块而言，即通过遍历 commands 数组进行，另外，在数组中检查到 ngx_null_command 时，会停止使用当前模块解析该配置项。每一个 ngx_command_t 结构体定义了自己感兴趣的一个配置项：

```
typedef struct ngx_command_s     ngx_command_t;
struct ngx_command_s {
    // 配置项名称，如 "gzip"
    ngx_str_t             name;
     /* 配置项类型，type 将指定配置项可以出现的位置。例如，出现在 server{} 或 location{} 中，以
及它可以携带的参数个数 */
    ngx_uint_t            type;
    // 出现了 name 中指定的配置项后，将会调用 set 方法处理配置项的参数
    char               *(*set)(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
    // 在配置文件中的偏移量
    ngx_uint_t            conf;
     /* 通常用于使用预设的解析方法解析配置项，这是配置模块的一个优秀设计。它需要与 conf 配合使用，
在第 4 章中详细介绍 */
    ngx_uint_t            offset;
    // 配置项读取后的处理方法，必须是 ngx_conf_post_t 结构的指针
    void                 *post;
};
```

ngx_null_command 只是一个空的 ngx_command_t，如下所示：

```
#define ngx_null_command  { ngx_null_string, 0, NULL, 0, 0, NULL }
```

3.5 定义自己的 HTTP 模块

上文中我们了解了定义 HTTP 模块时需要定义哪些成员以及实现哪些方法，但在定义 HTTP 模块前，首先需要确定自定义的模块应当在什么样的场景下开始处理用户请求，也就是说，先要弄清楚我们的模块是如何介入到 Nginx 处理用户请求的流程中的。从 2.4 节中

的 HTTP 配置项意义可知，一个 HTTP 请求会被许多个配置项控制，实际上这是因为一个 HTTP 请求可以被许多个 HTTP 模块同时处理。这样一来，肯定会有一个先后问题，也就是说，谁先处理请求谁的“权力”就更大。例如，ngx_http_access_module 模块的 deny 选项一旦得到满足后，Nginx 就会决定拒绝来自某个 IP 的请求，后面的诸如 root 这种访问静态文件的处理方式是得不到执行的。另外，由于同一个配置项可以从属于许多个 server、location 配置块，那么这个配置项将会针对不同的请求起作用。因此，现在面临的问题是，我们希望自己的模块在哪个时刻开始处理请求？是希望自己的模块对到达 Nginx 的所有请求都起作用，还是希望只对某一类请求（如 URI 匹配了 location 后表达式的请求）起作用？

Nginx 的 HTTP 框架定义了非常多的用法，我们有很大的自由来定义自己的模块如何介入 HTTP 请求的处理，但本章只想说明最简单、最常见的 HTTP 模块应当如何编写，因此，我们这样定义第一个 HTTP 模块介入 Nginx 的方式：

1）不希望模块对所有的 HTTP 请求起作用。

2）在 nginx.conf 文件中的 http{}、server{} 或者 location{} 块内定义 mytest 配置项，如果一个用户请求通过主机域名、URI 等匹配上了相应的配置块，而这个配置块下又具有 mytest 配置项，那么希望 mytest 模块开始处理请求。

在这种介入方式下，模块处理请求的顺序是固定的，即必须在 HTTP 框架定义的 NGX_HTTP_CONTENT_PHASE 阶段开始处理请求，具体内容下文详述。

下面开始按照这种方式定义 mytest 模块。首先，定义 mytest 配置项的处理。从上文中关于 ngx_command_t 结构的说明来看，只需要定义一个 ngx_command_t 数组，并设置在出现 mytest 配置后的解析方法由 ngx_http_mytest“担当”，如下所示：

```
    static ngx_command_t  ngx_http_mytest_commands[] = {

        { ngx_string("mytest"),
    NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_HTTP_LMT_CONF|NGX_
CONF_NOARGS,
          ngx_http_mytest,
          NGX_HTTP_LOC_CONF_OFFSET,
          0,
          NULL },

          ngx_null_command
    };
```

其中，ngx_http_mytest 是 ngx_command_t 结构体中的 set 成员（完整定义为 char *(*set)(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);），当在某个配置块中出现 mytest 配置项时，Nginx 将会调用 ngx_http_mytest 方法。下面看一下如何实现 ngx_http_mytest 方法。

```
    static char *
    ngx_http_mytest(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
```

```
    {
        ngx_http_core_loc_conf_t  *clcf;

         /* 首先找到 mytest 配置项所属的配置块，clcf 看上去像是 location 块内的数据结构，其实不然，
它可以是 main、srv 或者 loc 级别配置项，也就是说，在每个 http{} 和 server{} 内也都有一个 ngx_http_
core_loc_conf_t 结构体 */
        clcf = ngx_http_conf_get_module_loc_conf(cf, ngx_http_core_module);

        /*HTTP 框架在处理用户请求进行到 NGX_HTTP_CONTENT_PHASE 阶段时，如果请求的主机域名、URI 与
mytest 配置项所在的配置块相匹配，就将调用我们实现的 ngx_http_mytest_handler 方法处理这个请求 */
        clcf->handler = ngx_http_mytest_handler;

        return NGX_CONF_OK;
    }
```

当 Nginx 接收完 HTTP 请求的头部信息时，就会调用 HTTP 框架处理请求，另外在 11.6 节描述的 NGX_HTTP_CONTENT_PHASE 阶段将有可能调用 mytest 模块处理请求。在 ngx_http_mytest 方法中，我们定义了请求的处理方法为 ngx_http_mytest_handler，举个例子来说，如果用户的请求 URI 是 /test/example，而在配置文件中有这样的 location 块：

```
Location /test {
    mytest;
}
```

那么，HTTP 框架在 NGX_HTTP_CONTENT_PHASE 阶段就会调用到我们实现的 ngx_http_mytest_handler 方法来处理这个用户请求。事实上，HTTP 框架共定义了 11 个阶段（第三方 HTTP 模块只能介入其中的 7 个阶段处理请求，详见 10.6 节），本章只关注 NGX_HTTP_CONTENT_PHASE 处理阶段，多数 HTTP 模块都在此阶段实现相关功能。下面简单说明一下这 11 个阶段。

```
typedef enum {
    // 在接收到完整的 HTTP 头部后处理的 HTTP 阶段
    NGX_HTTP_POST_READ_PHASE = 0,

    /* 在还没有查询到 URI 匹配的 location 前，这时 rewrite 重写 URL 也作为一个独立的 HTTP 阶段 */
    NGX_HTTP_SERVER_REWRITE_PHASE,

    /* 根据 URI 寻找匹配的 location，这个阶段通常由 ngx_http_core_module 模块实现，不建议其他
HTTP 模块重新定义这一阶段的行为 */
    NGX_HTTP_FIND_CONFIG_PHASE,

    /* 在 NGX_HTTP_FIND_CONFIG_PHASE 阶段之后重写 URL 的意义与 NGX_HTTP_SERVER_REWRITE_
PHASE 阶段显然是不同的，因为这两者会导致查找到不同的 location 块（location 是与 URI 进行匹配的）*/
    NGX_HTTP_REWRITE_PHASE,

    /* 这一阶段是用于在 rewrite 重写 URL 后重新跳到 NGX_HTTP_FIND_CONFIG_PHASE 阶段，找到与新的
URI 匹配的 location。所以，这一阶段是无法由第三方 HTTP 模块处理的，而仅由 ngx_http_core_module 模块使用 */
```

```
        NGX_HTTP_POST_REWRITE_PHASE,

        // 处理 NGX_HTTP_ACCESS_PHASE 阶段前，HTTP 模块可以介入的处理阶段
        NGX_HTTP_PREACCESS_PHASE,

        /* 这个阶段用于让 HTTP 模块判断是否允许这个请求访问 Nginx 服务器
        NGX_HTTP_ACCESS_PHASE,

        /* 当 NGX_HTTP_ACCESS_PHASE 阶段中 HTTP 模块的 handler 处理方法返回不允许访问的错误码时
(实际是 NGX_HTTP_FORBIDDEN 或者 NGX_HTTP_UNAUTHORIZED)，这个阶段将负责构造拒绝服务的用户响应。所
以，这个阶段实际上用于给 NGX_HTTP_ACCESS_PHASE 阶段收尾 */
        NGX_HTTP_POST_ACCESS_PHASE,

        /* 这个阶段完全是为了 try_files 配置项而设立的。当 HTTP 请求访问静态文件资源时，try_files 配置
项可以使这个请求顺序地访问多个静态文件资源，如果某一次访问失败，则继续访问 try_files 中指定的下一个静态
资源。另外，这个功能完全是在 NGX_HTTP_TRY_FILES_PHASE 阶段中实现的 */
        NGX_HTTP_TRY_FILES_PHASE,

        // 用于处理 HTTP 请求内容的阶段，这是大部分 HTTP 模块最喜欢介入的阶段
        NGX_HTTP_CONTENT_PHASE,

        /* 处理完请求后记录日志的阶段。例如，ngx_http_log_module 模块就在这个阶段中加入了一个
handler 处理方法，使得每个 HTTP 请求处理完毕后会记录 access_log 日志 */
        NGX_HTTP_LOG_PHASE
    } ngx_http_phases;
```

当然，用户可以在以上 11 个阶段中任意选择一个阶段让 mytest 模块介入，但这需要学习完第 10 章、第 11 章的内容，完全熟悉了 HTTP 框架的处理流程后才可以做到。

暂且不管如何实现处理请求的 ngx_http_mytest_handler 方法，如果没有什么工作是必须在 HTTP 框架初始化时完成的，那就不必实现 ngx_http_module_t 的 8 个回调方法，可以像下面这样定义 ngx_http_module_t 接口。

```
static ngx_http_module_t  ngx_http_mytest_module_ctx = {
    NULL,                          /* preconfiguration */
    NULL,                          /* postconfiguration */

    NULL,                          /* create main configuration */
    NULL,                          /* init main configuration */

    NULL,                          /* create server configuration */
    NULL,                          /* merge server configuration */

    NULL,                          /* create location configuration */
    NULL                           /* merge location configuration */
};
```

最后，定义 mytest 模块：

```
ngx_module_t  ngx_http_mytest_module = {
```

```
    NGX_MODULE_V1,
    &ngx_http_mytest_module_ctx,           /* module context */
    ngx_http_mytest_commands,              /* module directives */
    NGX_HTTP_MODULE,                       /* module type */
    NULL,                                  /* init master */
    NULL,                                  /* init module */
    NULL,                                  /* init process */
    NULL,                                  /* init thread */
    NULL,                                  /* exit thread */
    NULL,                                  /* exit process */
    NULL,                                  /* exit master */
    NGX_MODULE_V1_PADDING
};
```

这样，mytest 模块在编译时将会被加入到 ngx_modules 全局数组中。Nginx 在启动时，会调用所有模块的初始化回调方法，当然，这个例子中我们没有实现它们（也没有实现 HTTP 框架初始化时会调用的 ngx_http_module_t 中的 8 个方法）。

3.6 处理用户请求

本节介绍如何处理一个实际的 HTTP 请求。回顾一下上文，在出现 mytest 配置项时，ngx_http_mytest 方法会被调用，这时将 ngx_http_core_loc_conf_t 结构的 handler 成员指定为 ngx_http_mytest_handler，另外，HTTP 框架在接收完 HTTP 请求的头部后，会调用 handler 指向的方法。下面看一下 handler 成员的原型 ngx_http_handler_pt：

```
typedef ngx_int_t (*ngx_http_handler_pt)(ngx_http_request_t *r);
```

从上面这段代码可以看出，实际处理请求的方法 ngx_http_mytest_handler 将接收一个 ngx_http_request_t 类型的参数 r，返回一个 ngx_int_t（参见 3.2.1 节）类型的结果。下面先探讨一下 ngx_http_mytest_handler 方法可以返回什么，再看一下参数 r 包含了哪些 Nginx 已经解析完的用户请求信息。

3.6.1 处理方法的返回值

这个返回值可以是 HTTP 中响应包的返回码，其中包括了 HTTP 框架已经在 /src/http/ngx_http_request.h 文件中定义好的宏，如下所示。

```
#define NGX_HTTP_OK                        200
#define NGX_HTTP_CREATED                   201
#define NGX_HTTP_ACCEPTED                  202
#define NGX_HTTP_NO_CONTENT                204
#define NGX_HTTP_PARTIAL_CONTENT           206

#define NGX_HTTP_SPECIAL_RESPONSE          300
```

```
#define NGX_HTTP_MOVED_PERMANENTLY         301
#define NGX_HTTP_MOVED_TEMPORARILY         302
#define NGX_HTTP_SEE_OTHER                 303
#define NGX_HTTP_NOT_MODIFIED              304
#define NGX_HTTP_TEMPORARY_REDIRECT        307

#define NGX_HTTP_BAD_REQUEST               400
#define NGX_HTTP_UNAUTHORIZED              401
#define NGX_HTTP_FORBIDDEN                 403
#define NGX_HTTP_NOT_FOUND                 404
#define NGX_HTTP_NOT_ALLOWED               405
#define NGX_HTTP_REQUEST_TIME_OUT          408
#define NGX_HTTP_CONFLICT                  409
#define NGX_HTTP_LENGTH_REQUIRED           411
#define NGX_HTTP_PRECONDITION_FAILED       412
#define NGX_HTTP_REQUEST_ENTITY_TOO_LARGE  413
#define NGX_HTTP_REQUEST_URI_TOO_LARGE     414
#define NGX_HTTP_UNSUPPORTED_MEDIA_TYPE    415
#define NGX_HTTP_RANGE_NOT_SATISFIABLE     416

/* The special code to close connection without any response */
#define NGX_HTTP_CLOSE                     444
#define NGX_HTTP_NGINX_CODES               494
#define NGX_HTTP_REQUEST_HEADER_TOO_LARGE  494
#define NGX_HTTPS_CERT_ERROR               495
#define NGX_HTTPS_NO_CERT                  496

#define NGX_HTTP_TO_HTTPS                  497
#define NGX_HTTP_CLIENT_CLOSED_REQUEST     499


#define NGX_HTTP_INTERNAL_SERVER_ERROR     500
#define NGX_HTTP_NOT_IMPLEMENTED           501
#define NGX_HTTP_BAD_GATEWAY               502
#define NGX_HTTP_SERVICE_UNAVAILABLE       503
#define NGX_HTTP_GATEWAY_TIME_OUT          504
#define NGX_HTTP_INSUFFICIENT_STORAGE      507
```

注意　以上返回值除了 RFC2616 规范中定义的返回码外，还有 Nginx 自身定义的 HTTP 返回码。例如，NGX_HTTP_CLOSE 就是用于要求 HTTP 框架直接关闭用户连接的。

在 ngx_http_mytest_handler 的返回值中，如果是正常的 HTTP 返回码，Nginx 就会按照规范构造合法的响应包发送给用户。例如，假设对于 PUT 方法暂不支持，那么，在处理方法中发现方法名是 PUT 时，返回 NGX_HTTP_NOT_ALLOWED，这样 Nginx 也就会构造类似下面的响应包给用户。

```
http/1.1 405 Not Allowed
Server: nginx/1.0.14
```

```
Date: Sat, 28 Apr 2012 06:07:17 GMT
Content-Type: text/html
Content-Length: 173
Connection: keep-alive

<html>
<head><title>405 Not Allowed</title></head>
<body bgcolor="white">
<center><h1>405 Not Allowed</h1></center>
<hr><center>nginx/1.0.14</center>
</body>
</html>
```

在处理方法中除了返回 HTTP 响应码外，还可以返回 Nginx 全局定义的几个错误码，包括：

```
#define  NGX_OK          0
#define  NGX_ERROR      -1
#define  NGX_AGAIN      -2
#define  NGX_BUSY       -3
#define  NGX_DONE       -4
#define  NGX_DECLINED   -5
#define  NGX_ABORT      -6
```

这些错误码对于 Nginx 自身提供的大部分方法来说都是通用的。所以，当我们最后调用 ngx_http_output_filter（参见 3.7 节）向用户发送响应包时，可以将 ngx_http_output_filter 的返回值作为 ngx_http_mytest_handler 方法的返回值使用。例如：

```
static ngx_int_t ngx_http_mytest_handler(ngx_http_request_t *r)
{
    ...

    ngx_int_t rc = ngx_http_send_header(r);
    if (rc == NGX_ERROR || rc > NGX_OK || r->header_only) {
        return rc;
    }

    return ngx_http_output_filter(r, &out);
}
```

当然，直接返回以上 7 个通用值也是可以的。在不同的场景下，这 7 个通用返回值代表的含义不尽相同。在 mytest 的例子中，HTTP 框架在 NGX_HTTP_CONTENT_PHASE 阶段调用 ngx_http_mytest_handler 后，会将 ngx_http_mytest_handler 的返回值作为参数传给 ngx_http_finalize_request 方法，如下所示。

```
    if (r->content_handler) {
        r->write_event_handler = ngx_http_request_empty_handler;
        ngx_http_finalize_request(r, r->content_handler(r));
```

```
    return NGX_OK;
}
```

上面的 r->content_handler 会指向 ngx_http_mytest_handler 处理方法。也就是说，事实上 ngx_http_finalize_request 决定了 ngx_http_mytest_handler 如何起作用。本章不探讨 ngx_http_finalize_request 的实现（详见 11.10 节），只简单地说明一下 4 个通用返回码，另外，在 11.10 节中介绍这 4 个返回码引发的 Nginx 一系列动作。

- NGX_OK ：表示成功。Nginx 将会继续执行该请求的后续动作（如执行 subrequest 或撤销这个请求）。
- NGX_DECLINED ：继续在 NGX_HTTP_CONTENT_PHASE 阶段寻找下一个对于该请求感兴趣的 HTTP 模块来再次处理这个请求。
- NGX_DONE ：表示到此为止，同时 HTTP 框架将暂时不再继续执行这个请求的后续部分。事实上，这时会检查连接的类型，如果是 keepalive 类型的用户请求，就会保持住 HTTP 连接，然后把控制权交给 Nginx。这个返回码很有用，考虑以下场景 ：在一个请求中我们必须访问一个耗时极长的操作（比如某个网络调用），这样会阻塞住 Nginx，又因为我们没有把控制权交还给 Nginx，而是在 ngx_http_mytest_handler 中让 Nginx worker 进程休眠了（如等待网络的回包），所以，这就会导致 Nginx 出现性能问题，该进程上的其他用户请求也得不到响应。可如果我们把这个耗时极长的操作分为上下两个部分（就像 Linux 内核中对中断处理的划分），上半部分和下半部分都是无阻塞的（耗时很少的操作），这样，在 ngx_http_mytest_handler 进入时调用上半部分，然后返回 NGX_DONE，把控制交还给 Nginx，从而让 Nginx 继续处理其他请求。在下半部分被触发时（这里不探讨具体的实现方式，事实上使用 upstream 方式做反向代理时用的就是这种思想），再回调下半部分处理方法，这样就可以保证 Nginx 的高性能特性了。如果需要彻底了解 NGX_DONE 的意义，那么必须学习第 11 章内容，其中还涉及请求的引用计数内容。
- NGX_ERROR：表示错误。这时会调用 ngx_http_terminate_request 终止请求。如果还有 POST 子请求，那么将会在执行完 POST 请求后再终止本次请求。

3.6.2　获取 URI 和参数

请求的所有信息（如方法、URI、协议版本号和头部等）都可以在传入的 ngx_http_request_t 类型参数 r 中取得。ngx_http_request_t 结构体的内容很多，本节不会探讨 ngx_http_request_t 中所有成员的意义（ngx_http_request_t 结构体中的许多成员只有 HTTP 框架才感兴趣，在 11.3.1 节会更详细的说明），只介绍一下获取 URI 和参数的方法，这非常简单，因为 Nginx 提供了多种方法得到这些信息。下面先介绍相关成员的定义。

```
typedef struct ngx_http_request_s     ngx_http_request_t;
```

```
struct ngx_http_request_s {
    ...
    ngx_uint_t                        method;
    ngx_uint_t                        http_version;

    ngx_str_t                         request_line;
    ngx_str_t                         uri;
    ngx_str_t                         args;
    ngx_str_t                         exten;
    ngx_str_t                         unparsed_uri;

    ngx_str_t                         method_name;
    ngx_str_t                         http_protocol;

    u_char                           *uri_start;
    u_char                           *uri_end;
    u_char                           *uri_ext;
    u_char                           *args_start;
    u_char                           *request_start;
    u_char                           *request_end;
    u_char                           *method_end;
    u_char                           *schema_start;
    u_char                           *schema_end;
    ...
};
```

在对一个用户请求行进行解析时，可以得到下列 4 类信息。

（1）方法名

method 的类型是 ngx_uint_t（无符号整型），它是 Nginx 忽略大小写等情形时解析完用户请求后得到的方法类型，其取值范围如下所示。

```
#define NGX_HTTP_UNKNOWN                  0x0001
#define NGX_HTTP_GET                      0x0002
#define NGX_HTTP_HEAD                     0x0004
#define NGX_HTTP_POST                     0x0008
#define NGX_HTTP_PUT                      0x0010
#define NGX_HTTP_DELETE                   0x0020
#define NGX_HTTP_MKCOL                    0x0040
#define NGX_HTTP_COPY                     0x0080
#define NGX_HTTP_MOVE                     0x0100
#define NGX_HTTP_OPTIONS                  0x0200
#define NGX_HTTP_PROPFIND                 0x0400
#define NGX_HTTP_PROPPATCH                0x0800
#define NGX_HTTP_LOCK                     0x1000
#define NGX_HTTP_UNLOCK                   0x2000
#define NGX_HTTP_TRACE                    0x4000
```

当需要了解用户请求中的 HTTP 方法时，应该使用 r->method 这个整型成员与以上 15 个宏进行比较，这样速度是最快的（如果使用 method_name 成员与字符串做比较，那么效

率会差很多），大部分情况下推荐使用这种方式。除此之外，还可以用 method_name 取得用户请求中的方法名字符串，或者联合 request_start 与 method_end 指针取得方法名。method_name 是 ngx_str_t 类型，按照 3.2.2 节中介绍的方法使用即可。

request_start 与 method_end 的用法也很简单，其中 request_start 指向用户请求的首地址，同时也是方法名的地址，method_end 指向方法名的最后一个字符（注意，这点与其他 xxx_end 指针不同）。获取方法名时可以从 request_start 开始向后遍历，直到地址与 method_end 相同为止，这段内存存储着方法名。

注意　Nginx 中对内存的控制相当严格，为了避免不必要的内存开销，许多需要用到的成员都不是重新分配内存后存储的，而是直接指向用户请求中的相应地址。例如，method_name.data、request_start 这两个指针实际指向的都是同一个地址。而且，因为它们是简单的内存指针，不是指向字符串的指针，所以，在大部分情况下，都不能将这些 u_char* 指针当做字符串使用。

（2）URI

ngx_str_t 类型的 uri 成员指向用户请求中的 URI。同理，u_char* 类型的 uri_start 和 uri_end 也与 request_start、method_end 的用法相似，唯一不同的是，method_end 指向方法名的最后一个字符，而 uri_end 指向 URI 结束后的下一个地址，也就是最后一个字符的下一个字符地址（HTTP 框架的行为），这是大部分 u_char* 类型指针对“xxx_start”和“xxx_end”变量的用法。

ngx_str_t 类型的 extern 成员指向用户请求的文件扩展名。例如，在访问“GET /a.txt HTTP/1.1”时，extern 的值是 {len = 3, data = "txt"}，而在访问“GET /a HTTP/1.1”时，extern 的值为空，也就是 {len = 0, data = 0x0}。

uri_ext 指针指向的地址与 extern.data 相同。

unparsed_uri 表示没有进行 URL 解码的原始请求。例如，当 uri 为“/a b”时，unparsed_uri 是“/a%20b”（空格字符做完编码后是 %20）。

（3）URL 参数

arg 指向用户请求中的 URL 参数。

args_start 指向 URL 参数的起始地址，配合 uri_end 使用也可以获得 URL 参数。

（4）协议版本

http_protocol 指向用户请求中 HTTP 的起始地址。

http_version 是 Nginx 解析过的协议版本，它的取值范围如下：

```
#define NGX_HTTP_VERSION_9                 9
#define NGX_HTTP_VERSION_10                1000
#define NGX_HTTP_VERSION_11                1001
```

建议使用 http_version 分析 HTTP 的协议版本。

最后，使用 request_start 和 request_end 可以获取原始的用户请求行。

3.6.3 获取 HTTP 头部

在 ngx_http_request_t* r 中就可以取到请求中的 HTTP 头部，比如使用下面的成员：

```
struct ngx_http_request_s {
    …
    ngx_buf_t                        *header_in;
     ngx_http_headers_in_t          headers_in;
    …
};
```

其中，header_in 指向 Nginx 收到的未经解析的 HTTP 头部，这里暂不关注它（在第 11 章中可以看到，header_in 就是接收 HTTP 头部的缓冲区）。ngx_http_headers_in_t 类型的 headers_in 则存储已经解析过的 HTTP 头部。下面介绍 ngx_http_headers_in_t 结构体中的成员。

```
    typedef struct {
        /* 所有解析过的 HTTP 头部都在 headers 链表中，可以使用 3.2.3 节中介绍的遍历链表的方法来获取所有的 HTTP 头部。注意，这里 headers 链表的每一个元素都是 3.2.4 节介绍过的 ngx_table_elt_t 成员 */
        ngx_list_t                        headers;

        /* 以下每个 ngx_table_elt_t 成员都是 RFC1616 规范中定义的 HTTP 头部，它们实际都指向 headers 链表中的相应成员。注意，当它们为 NULL 空指针时，表示没有解析到相应的 HTTP 头部 */
        ngx_table_elt_t                  *host;
        ngx_table_elt_t                  *connection;
        ngx_table_elt_t                  *if_modified_since;
        ngx_table_elt_t                  *if_unmodified_since;
        ngx_table_elt_t                  *user_agent;
        ngx_table_elt_t                  *referer;
        ngx_table_elt_t                  *content_length;
        ngx_table_elt_t                  *content_type;

        ngx_table_elt_t                  *range;
        ngx_table_elt_t                  *if_range;

        ngx_table_elt_t                  *transfer_encoding;
        ngx_table_elt_t                  *expect;

    #if (NGX_HTTP_GZIP)
        ngx_table_elt_t                  *accept_encoding;
        ngx_table_elt_t                  *via;
    #endif

        ngx_table_elt_t                  *authorization;
```

```
    ngx_table_elt_t                  *keep_alive;
#if (NGX_HTTP_PROXY || NGX_HTTP_REALIP || NGX_HTTP_GEO)
    ngx_table_elt_t                  *x_forwarded_for;
#endif

#if (NGX_HTTP_REALIP)
    ngx_table_elt_t                  *x_real_ip;
#endif

#if (NGX_HTTP_HEADERS)
    ngx_table_elt_t                  *accept;
    ngx_table_elt_t                  *accept_language;
#endif

#if (NGX_HTTP_DAV)
    ngx_table_elt_t                  *depth;
    ngx_table_elt_t                  *destination;
    ngx_table_elt_t                  *overwrite;
    ngx_table_elt_t                  *date;
#endif

    /*user 和 passwd 是只有 ngx_http_auth_basic_module 才会用到的成员，这里可以忽略 */
    ngx_str_t                         user;
    ngx_str_t                         passwd;

    /*cookies 是以 ngx_array_t 数组存储的，本章先不介绍这个数据结构，感兴趣的话可以直接跳到 7.3 节
了解 ngx_array_t 的相关用法 */
    ngx_array_t                       cookies;
    //server 名称
    ngx_str_t                         server;
    // 根据 ngx_table_elt_t  *content_length 计算出的 HTTP 包体大小
    off_t                             content_length_n;
    time_t                            keep_alive_n;

   /*HTTP 连接类型，它的取值范围是 0、NGX_http_CONNECTION_CLOSE 或者 NGX_HTTP_CONNECTION_
KEEP_ALIVE*/
    unsigned                          connection_type:2;
     /* 以下 7 个标志位是 HTTP 框架根据浏览器传来的“useragent”头部，它们可用来判断浏览器的类型，
值为 1 时表示是相应的浏览器发来的请求，值为 0 时则相反 */
    unsigned                          msie:1;
    unsigned                          msie6:1;
    unsigned                          opera:1;
    unsigned                          gecko:1;
    unsigned                          chrome:1;
    unsigned                          safari:1;
    unsigned                          konqueror:1;
} ngx_http_headers_in_t;
```

获取 HTTP 头部时，直接使用 r->headers_in 的相应成员就可以了。这里举例说明一下如何通过遍历 headers 链表获取非 RFC2616 标准的 HTTP 头部，读者可以先回顾一下 ngx_list_

t 链表和 ngx_table_elt_t 结构体的用法。前面 3.2.3 节中已经介绍过，headers 是一个 ngx_list_t 链表，它存储着解析过的所有 HTTP 头部，链表中的元素都是 ngx_table_elt_t 类型。下面尝试在一个用户请求中找到“Rpc-Description”头部，首先判断其值是否为“uploadFile”，再决定后续的服务器行为，代码如下。

```
    ngx_list_part_t *part = &r->headers_in.headers.part;
    ngx_table_elt_t *header = part->elts;

    // 开始遍历链表
    for (i = 0; /* void */; i++) {
        // 判断是否到达链表中当前数组的结尾处
        if (i >= part->nelts) {
            // 是否还有下一个链表数组元素
            if (part->next == NULL) {
                break;
            }

            /* part 设置为 next 来访问下一个链表数组；header 也指向下一个链表数组的首地址；i 设置为 0 时，表示从头开始遍历新的链表数组 */
            part = part->next;
            header = part->elts;
            i = 0;
        }

        //hash 为 0 时表示不是合法的头部
        if (header[i].hash == 0) {
            continue;
        }

        /* 判断当前的头部是否是“Rpc-Description”。如果想要忽略大小写，则应该先用 header[i].lowcase_key 代替 header[i].key.data，然后比较字符串 */
        if (0 == ngx_strncasecmp(header[i].key.data,
                (u_char*) "Rpc-Description",
                header[i].key.len))
        {
            // 判断这个 HTTP 头部的值是否是“uploadFile”
            if (0 == ngx_strncmp(header[i].value.data,
                   "uploadFile",
                   header[i].value.len))
            {
                // 找到了正确的头部，继续向下执行
            }
        }
    }
```

对于常见的 HTTP 头部，直接获取 r->headers_in 中已经由 HTTP 框架解析过的成员即可，而对于不常见的 HTTP 头部，需要遍历 r->headers_in.headers 链表才能获得。

3.6.4　获取 HTTP 包体

HTTP 包体的长度有可能非常大，如果试图一次性调用并读取完所有的包体，那么多半会阻塞 Nginx 进程。HTTP 框架提供了一种方法来异步地接收包体：

```
    ngx_int_t ngx_http_read_client_request_body(ngx_http_request_t *r, ngx_http_
client_body_handler_pt post_handler);
```

ngx_http_read_client_request_body 是一个异步方法，调用它只是说明要求 Nginx 开始接收请求的包体，并不表示是否已经接收完，当接收完所有的包体内容后，post_handler 指向的回调方法会被调用。因此，即使在调用了 ngx_http_read_client_request_body 方法后它已经返回，也无法确定这时是否已经调用过 post_handler 指向的方法。换句话说，ngx_http_read_client_request_body 返回时既有可能已经接收完请求中所有的包体（假如包体的长度很小），也有可能还没开始接收包体。如果 ngx_http_read_client_request_body 是在 ngx_http_mytest_handler 处理方法中调用的，那么后者一般要返回 NGX_DONE，因为下一步就是将它的返回值作为参数传给 ngx_http_finalize_request。NGX_DONE 的意义在 3.6.1 节中已经介绍过，这里不再赘述。

下面看一下包体接收完毕后的回调方法原型 ngx_http_client_body_handler_pt 是如何定义的：

```
    typedef void (*ngx_http_client_body_handler_pt)(ngx_http_request_t *r);
```

其中，有参数 ngx_http_request_t *r，这个请求的信息都可以从 r 中获得。这样可以定义一个方法 void func(ngx_http_request_t *r)，在 Nginx 接收完包体时调用它，另外，后续的流程也都会写在这个方法中，例如：

```
    void ngx_http_mytest_body_handler(ngx_http_request_t *r)
    {
       ...
    }
```

注意　ngx_http_mytest_body_handler 的返回类型是 void，Nginx 不会根据返回值做一些收尾工作，因此，我们在该方法里处理完请求时必须要主动调用 ngx_http_finalize_request 方法来结束请求。

接收包体时可以这样写：

```
            ngx_int_t rc = ngx_http_read_client_request_body(r, ngx_http_mytest_body_
handler);

            if (rc >= NGX_http_SPECIAL_RESPONSE) {
                return rc;
```

```
    }
    return NGX_DONE;
```

Nginx 异步接收 HTTP 请求的包体的内容将在 11.8 节中详述。

如果不想处理请求中的包体，那么可以调用 ngx_http_discard_request_body 方法将接收自客户端的 HTTP 包体丢弃掉。例如：

```
ngx_int_t rc = ngx_http_discard_request_body(r);
if (rc != NGX_OK) {
    return rc;
}
```

ngx_http_discard_request_body 只是丢弃包体，不处理包体不就行了吗？何必还要调用 ngx_http_discard_request_body 方法呢？其实这一步非常有意义，因为有些客户端可能会一直试图发送包体，而如果 HTTP 模块不接收发来的 TCP 流，有可能造成客户端发送超时。

接收完请求的包体后，可以在 r->request_body->temp_file->file 中获取临时文件（假定将 r->request_body_in_file_only 标志位设为 1，那就一定可以在这个变量获取到包体。更复杂的接收包体的方式本节暂不讨论）。file 是一个 ngx_file_t 类型，在 3.8 节会详细介绍它的用法。这里，我们可以从 r->request_body->temp_file->file.name 中获取 Nginx 接收到的请求包体所在文件的名称（包括路径）。

3.7 发送响应

请求处理完毕后，需要向用户发送 HTTP 响应，告知客户端 Nginx 的执行结果。HTTP 响应主要包括响应行、响应头部、包体三部分。发送 HTTP 响应时需要执行发送 HTTP 头部（发送 HTTP 头部时也会发送响应行）和发送 HTTP 包体两步操作。本节将以发送经典的“Hello World”为例来说明如何发送响应。

3.7.1 发送 HTTP 头部

下面看一下 HTTP 框架提供的发送 HTTP 头部的方法，如下所示。

```
ngx_int_t ngx_http_send_header(ngx_http_request_t *r);
```

调用 ngx_http_send_header 时把 ngx_http_request_t 对象传给它即可，而 ngx_http_request_t 的返回值是多样的，在本节中，可以认为返回 NGX_ERROR 或返回值大于 0 就表示不正常，例如：

```
ngx_int_t  rc = ngx_http_send_header(r);
if (rc == NGX_ERROR || rc > NGX_OK || r->header_only) {
    return rc;
}
```

下面介绍设置响应中的 HTTP 头部的过程。

如同 headers_in，ngx_http_request_t 也有一个 headers_out 成员，用来设置响应中的 HTTP 头部，如下所示。

```
struct ngx_http_request_s {
    ...
    ngx_http_headers_in_t             headers_in;
    ngx_http_headers_out_t            headers_out;
    ...
};
```

只要指定 headers_out 中的成员，就可以在调用 ngx_http_send_header 时正确地把 HTTP 头部发出。下面介绍 headers_out 的结构类型 ngx_http_headers_out_t。

```
typedef struct {
    // 待发送的 HTTP 头部链表，与 headers_in 中的 headers 成员类似
    ngx_list_t                        headers;

    /* 响应中的状态值，如 200 表示成功。这里可以使用 3.6.1 节中介绍过的各个宏，如 NGX_HTTP_OK */
    ngx_uint_t                        status;
    // 响应的状态行，如“HTTP/1.1 201 CREATED”
    ngx_str_t                         status_line;

    /* 以下成员（包括 ngx_table_elt_t）都是 RFC1616 规范中定义的 HTTP 头部，设置后，ngx_http_
header_filter_module 过滤模块可以把它们加到待发送的网络包中 */
    ngx_table_elt_t                  *server;
    ngx_table_elt_t                  *date;
    ngx_table_elt_t                  *content_length;
    ngx_table_elt_t                  *content_encoding;
    ngx_table_elt_t                  *location;
    ngx_table_elt_t                  *refresh;
    ngx_table_elt_t                  *last_modified;
    ngx_table_elt_t                  *content_range;
    ngx_table_elt_t                  *accept_ranges;
    ngx_table_elt_t                  *www_authenticate;
    ngx_table_elt_t                  *expires;
    ngx_table_elt_t                  *etag;

    ngx_str_t                         *override_charset;

    /* 可以调用 ngx_http_set_content_type(r) 方法帮助我们设置 Content-Type 头部，这个方法会
根据 URI 中的文件扩展名并对应着 mime.type 来设置 Content-Type 值 */
    size_t                            content_type_len;
    ngx_str_t                         content_type;
    ngx_str_t                         charset;
    u_char                           *content_type_lowcase;
    ngx_uint_t                        content_type_hash;

    ngx_array_t                       cache_control;
```

```
        /* 在这里指定过 content_length_n 后，不用再次到 ngx_table_elt_t *content_ length 中设置响应
长度 */
        off_t                            content_length_n;
        time_t                           date_time;
        time_t                           last_modified_time;
    } ngx_http_headers_out_t;
```

在向 headers 链表中添加自定义的 HTTP 头部时，可以参考 3.2.3 节中 ngx_list_push 的使用方法。这里有一个简单的例子，如下所示。

```
ngx_table_elt_t* h = ngx_list_push(&r->headers_out.headers);
if (h == NULL) {
   return NGX_ERROR;
}

h->hash = 1;
h->key.len = sizeof("TestHead") - 1;
h->key.data = (u_char *) "TestHead";
h->value.len = sizeof("TestValue") - 1;
h->value.data = (u_char *) "TestValue";
```

这样将会在响应中新增一行 HTTP 头部：

```
TestHead: TestValud\r\n
```

如果发送的是一个不含有 HTTP 包体的响应，这时就可以直接结束请求了（例如，在 ngx_http_mytest_handler 方法中，直接在 ngx_http_send_header 方法执行后将其返回值 return 即可）。

注意 ngx_http_send_header 方法会首先调用所有的 HTTP 过滤模块共同处理 headers_out 中定义的 HTTP 响应头部，全部处理完毕后才会序列化为 TCP 字符流发送到客户端，相关流程可参见 11.9.1 节。

3.7.2 将内存中的字符串作为包体发送

调用 ngx_http_output_filter 方法即可向客户端发送 HTTP 响应包体，下面查看一下此方法的原型，如下所示。

```
ngx_int_t ngx_http_output_filter(ngx_http_request_t *r, ngx_chain_t *in);
```

ngx_http_output_filter 的返回值在 mytest 例子中不需要处理，通过在 ngx_http_mytest_handler 方法中返回的方式传递给 ngx_http_finalize_request 即可。ngx_chain_t 结构已经在 3.2.6 节中介绍过，它仅用于容纳 ngx_buf_t 缓冲区，所以需要先了解一下如何使用 ngx_buf_t 分配内存。下面介绍 Nginx 的内存池是如何分配内存的。

为了减少内存碎片的数量，并通过统一管理来减少代码中出现内存泄漏的可能性，Nginx 设计了 ngx_pool_t 内存池数据结构。本章我们不会深入分析内存池的实现，只关注内存池的用法。在 ngx_http_mytest_handler 处理方法传来的 ngx_http_request_t 对象中就有这个请求的内存池管理对象，我们对内存池的操作都可以基于它来进行，这样，在这个请求结束的时候，内存池分配的内存也都会被释放。

```
struct ngx_http_request_s {
    ...
    ngx_pool_t *pool;
   ...
};
```

实际上，在 r 中可以获得许多内存池对象，这些内存池的大小、意义及生存期各不相同。第 3 部分会涉及许多内存池，本章使用 r->pool 内存池即可。有了 ngx_pool_t 对象后，可以从内存池中分配内存。例如，下面这个基本的申请分配内存的方法：

```
void *ngx_palloc(ngx_pool_t *pool, size_t size);
```

其中，ngx_palloc 函数将会从 pool 内存池中分配到 size 字节的内存，并返回这段内存的起始地址。如果返回 NULL 空指针，则表示分配失败。还有一个封装了 ngx_palloc 的函数 ngx_pcalloc，它多做了一件事，就是把 ngx_palloc 申请到的内存块全部置为 0，虽然，多数情况下更适合用 ngx_pcalloc 来分配内存。

假如要分配一个 ngx_buf_t 结构，可以这样做：

```
ngx_buf_t* b = ngx_pcalloc(r->pool, sizeof(ngx_buf_t));
```

这样，ngx_buf_t 中的成员指向的内存仍然可以继续分配，例如：

```
b->start = (u_char*)ngx_pcalloc(r->pool, 128);
b->pos = b->start;
b->last = b->start;
b->end = b->last + 128;
b->temporary = 1;
```

实际上，Nginx 还封装了一个生成 ngx_buf_t 的简便方法，它完全等价于上面的 6 行语句，如下所示。

```
ngx_buf_t *b = ngx_create_temp_buf(r->pool, 128);
```

分配完内存后，可以向这段内存写入数据。当写完数据后，要让 b->last 指针指向数据的末尾，如果 b->last 与 b->pos 相等，那么 HTTP 框架是不会发送一个字节的包体的。

最后，把上面的 ngx_buf_t *b 用 ngx_chain_t 传给 ngx_http_output_filter 方法就可以发送 HTTP 响应的包体内容了。例如：

```
ngx_chain_t out;
out.buf = b;
out.next = NULL;

return ngx_http_output_filter(r, &out);
```

注意 在向用户发送响应包体时，必须牢记 Nginx 是全异步的服务器，也就是说，不可以在进程的栈里分配内存并将其作为包体发送。当 ngx_http_output_filter 方法返回时，可能由于 TCP 连接上的缓冲区还不可写，所以导致 ngx_buf_t 缓冲区指向的内存还没有发送，可这时方法返回已把控制权交给 Nginx 了，又会导致栈里的内存被释放，最后就会造成内存越界错误。因此，在发送响应包体时，尽量将 ngx_buf_t 中的 pos 指针指向从内存池里分配的内存。

3.7.3 经典的“Hello World”示例

下面以经典的返回“Hello World”为例来编写一个最小的 HTTP 处理模块，以此介绍完整的 ngx_http_mytest_handler 处理方法。

```
static ngx_int_t ngx_http_mytest_handler(ngx_http_request_t *r)
{
    //必须是 GET 或者 HEAD 方法，否则返回 405 Not Allowed
    if (!(r->method & (NGX_HTTP_GET|NGX_HTTP_HEAD))) {
        return NGX_HTTP_NOT_ALLOWED;
    }

    //丢弃请求中的包体
    ngx_int_t rc = ngx_http_discard_request_body(r);
    if (rc != NGX_OK) {
        return rc;
    }

    /* 设置返回的 Content-Type。注意，ngx_str_t 有一个很方便的初始化宏 ngx_string，它可以把
ngx_str_t 的 data 和 len 成员都设置好 */
    ngx_str_t type = ngx_string("text/plain");
    //返回的包体内容
    ngx_str_t response = ngx_string("Hello World!");
    //设置返回状态码
    r->headers_out.status = NGX_HTTP_OK;
    //响应包是有包体内容的，需要设置 Content-Length 长度
    r->headers_out.content_length_n = response.len;
    //设置 Content-Type
    r->headers_out.content_type = type;

    //发送 HTTP 头部
    rc = ngx_http_send_header(r);
    if (rc == NGX_ERROR || rc > NGX_OK || r->header_only) {
```

```
        return rc;
    }

    // 构造 ngx_buf_t 结构体准备发送包体
    ngx_buf_t *b;
    b = ngx_create_temp_buf(r->pool, response.len);
    if (b == NULL) {
        return NGX_HTTP_INTERNAL_SERVER_ERROR;
    }
    // 将 Hello World 复制到 ngx_buf_t 指向的内存中
    ngx_memcpy(b->pos, response.data, response.len);
    // 注意，一定要设置好 last 指针
    b->last = b->pos + response.len;
    // 声明这是最后一块缓冲区
    b->last_buf = 1;

    // 构造发送时的 ngx_chain_t 结构体
    ngx_chain_t out;
    // 赋值 ngx_buf_t
    out.buf = b;
    // 设置 next 为 NULL
    out.next = NULL;

    /* 最后一步为发送包体，发送结束后 HTTP 框架会调用 ngx_http_finalize_request 方法结束请求 */
    return ngx_http_output_filter(r, &out);
}
```

3.8　将磁盘文件作为包体发送

上文讨论了如何将内存中的数据作为包体发送给客户端，而在发送文件时完全可以先把文件读取到内存中再向用户发送数据，但是这样做会有两个缺点：

- 为了不阻塞 Nginx，每次只能读取并发送磁盘中的少量数据，需要反复持续多次。
- Linux 上高效的 sendfile 系统调用不需要先把磁盘中的数据读取到用户态内存再发送到网络中。

当然，Nginx 已经封装好了多种接口，以便将磁盘或者缓存中的文件发送给用户。

3.8.1　如何发送磁盘中的文件

发送文件时使用的是 3.7 节中所介绍的接口。例如：

```
ngx_chain_t out;
out.buf = b;
out.next = NULL;

return ngx_http_output_filter(r, &out);
```

两者不同的地方在于如何设置 ngx_buf_t 缓冲区。在 3.2.5 节中介绍过，ngx_buf_t 有一个标志位 in_file，将 in_file 置为 1 就表示这次 ngx_buf_t 缓冲区发送的是文件而不是内存。调用 ngx_http_output_filter 后，若 Nginx 检测到 in_file 为 1，将会从 ngx_buf_t 缓冲区中的 file 成员处获取实际的文件。file 的类型是 ngx_file_t，下面看一下 ngx_file_t 的结构。

```
typedef struct ngx_file_s ngx_file_t;
struct ngx_file_s {
    // 文件句柄描述符
    ngx_fd_t fd;
    // 文件名称
    ngx_str_t name;
    // 文件大小等资源信息，实际就是 Linux 系统定义的 stat 结构
    ngx_file_info_t info;

    /* 该偏移量告诉 Nginx 现在处理到文件何处了，一般不用设置它，Nginx 框架会根据当前发送状态设置它 */
    off_t offset;
    // 当前文件系统偏移量，一般不用设置它，同样由 Nginx 框架设置
    off_t sys_offset;

    // 日志对象，相关的日志会输出到 log 指定的日志文件中
    ngx_log_t *log;
    // 目前未使用
    unsigned valid_info:1;
    // 与配置文件中的 directio 配置项相对应，在发送大文件时可以设为 1
    unsigned directio:1;
};
```

fd 是打开文件的句柄描述符，打开文件这一步需要用户自己来做。Nginx 简单封装了一个宏用来代替 open 系统的调用，如下所示。

```
#define ngx_open_file(name, mode, create, access) \
    open((const char *) name, mode|create, access)
```

实际上，ngx_open_file 与 open 方法的区别不大，ngx_open_file 返回的是 Linux 系统的文件句柄。对于打开文件的标志位，Nginx 也定义了以下几个宏来加以封装。

```
#define NGX_FILE_RDONLY O_RDONLY
#define NGX_FILE_WRONLY O_WRONLY
#define NGX_FILE_RDWR O_RDWR
#define NGX_FILE_CREATE_OR_OPEN O_CREAT
#define NGX_FILE_OPEN 0
#define NGX_FILE_TRUNCATE O_CREAT|O_TRUNC
#define NGX_FILE_APPEND O_WRONLY|O_APPEND
#define NGX_FILE_NONBLOCK O_NONBLOCK

#define NGX_FILE_DEFAULT_ACCESS 0644
#define NGX_FILE_OWNER_ACCESS 0600
```

因此，在打开文件时只需要把文件路径传递给 name 参数，并把打开方式传递给 mode、create、access 参数即可。例如：

```
ngx_buf_t *b;
b = ngx_palloc(r->pool, sizeof(ngx_buf_t));

u_char* filename = (u_char*)"/tmp/test.txt";
b->in_file = 1;
b->file = ngx_pcalloc(r->pool, sizeof(ngx_file_t));
b->file->fd = ngx_open_file(filename, NGX_FILE_RDONLY|NGX_FILE_NONBLOCK, NGX_FILE_
OPEN, 0);
b->file->log = r->connection->log;
b->file->name.data = filename;
b->file->name.len = sizeof(filename)-1;
if (b->file->fd <= 0)
{
   return NGX_HTTP_NOT_FOUND;
}
```

到这里其实还没有结束，还需要告知 Nginx 文件的大小，包括设置响应中的 Content-Length 头部，以及设置 ngx_buf_t 缓冲区的 file_pos 和 file_last。实际上，通过 ngx_file_t 结构里 ngx_file_info_t 类型的 info 变量就可以获取文件信息：

```
typedef struct stat ngx_file_info_t;
```

Nginx 不只对 stat 数据结构做了封装，对于由操作系统中获取文件信息的 stat 方法，Nginx 也使用一个宏进行了简单的封装，如下所示：

```
#define ngx_file_info(file, sb)  stat((const char *) file, sb)
```

因此，获取文件信息时可以先这样写：

```
if (ngx_file_info(filename, &b->file->info) == NGX_FILE_ERROR) {
   return NGX_HTTP_INTERNAL_SERVER_ERROR;
}
```

之后必须要设置 Content-Length 头部：

```
r->headers_out.content_length_n = b->file->info.st_size;
```

还需要设置 ngx_buf_t 缓冲区的 file_pos 和 file_last：

```
b->file_pos = 0;
b->file_last = b->file->info.st_size;
```

这里是告诉 Nginx 从文件的 file_pos 偏移量开始发送文件，一直到达 file_last 偏移量处截止。

注意 当磁盘中有大量的小文件时，会占用 Linux 文件系统中过多的 inode 结构，这时，成熟的解决方案会把许多小文件合并成一个大文件。在这种情况下，当有需要时，只要把上面的 file_pos 和 file_last 设置为合适的偏移量，就可以只发送合并大文件中的某一块内容（原来的小文件），这样就可以大幅降低小文件数量。

3.8.2 清理文件句柄

Nginx 会异步地将整个文件高效地发送给用户，但是我们必须要求 HTTP 框架在响应发送完毕后关闭已经打开的文件句柄，否则将会出现句柄泄露问题。设置清理文件句柄也很简单，只需要定义一个 ngx_pool_cleanup_t 结构体（这是最简单的方法，HTTP 框架还提供了其他方式，在请求结束时回调各个 HTTP 模块的 cleanup 方法，将在第 11 章介绍），将我们刚得到的文件句柄等信息赋给它，并将 Nginx 提供的 ngx_pool_cleanup_file 函数设置到它的 handler 回调方法中即可。首先介绍一下 ngx_pool_cleanup_t 结构体。

```
typedef struct ngx_pool_cleanup_s  ngx_pool_cleanup_t;

struct ngx_pool_cleanup_s {
    // 执行实际清理资源工作的回调方法
    ngx_pool_cleanup_pt   handler;
    //handler 回调方法需要的参数
    void *data;
    // 下一个 ngx_pool_cleanup_t 清理对象，如果没有，需置为 NULL
    ngx_pool_cleanup_t *next;
};
```

设置好 handler 和 data 成员就有可能要求 HTTP 框架在请求结束前传入 data 成员回调 handler 方法。接着，介绍一下专用于关闭文件句柄的 ngx_pool_cleanup_file 方法。

```
void ngx_pool_cleanup_file(void *data)
{
    ngx_pool_cleanup_file_t  *c = data;

    ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, c->log, 0, "file cleanup: fd:%d",c->fd);

    if (ngx_close_file(c->fd) == NGX_FILE_ERROR) {
        ngx_log_error(NGX_LOG_ALERT, c->log, ngx_errno,
                      ngx_close_file_n " \"%s\" failed", c->name);
    }
}
```

ngx_pool_cleanup_file 的作用是把文件句柄关闭。从上面的实现中可以看出，ngx_pool_cleanup_file 方法需要一个 ngx_pool_cleanup_file_t 类型的参数，那么，如何提供这个参数呢？在 ngx_pool_cleanup_t 结构体的 data 成员上赋值即可。下面介绍一下 ngx_pool_

cleanup_file_t 的结构。

```
typedef struct {
    // 文件句柄
    ngx_fd_t fd;
    // 文件名称
    u_char *name;
    // 日志对象
    ngx_log_t *log;
} ngx_pool_cleanup_file_t;
```

可以看到，ngx_pool_cleanup_file_t 中的对象在 ngx_buf_t 缓冲区的 file 结构体中都出现过了，意义也是相同的。对于 file 结构体，我们在内存池中已经为它分配过内存，只有在请求结束时才会释放，因此，这里简单地引用 file 里的成员即可。清理文件句柄的完整代码如下。

```
    ngx_pool_cleanup_t* cln = ngx_pool_cleanup_add(r->pool, sizeof(ngx_pool_cleanup_
file_t));
    if (cln == NULL) {
       return NGX_ERROR;
    }

    cln->handler = ngx_pool_cleanup_file;
    ngx_pool_cleanup_file_t  *clnf = cln->data;

    clnf->fd = b->file->fd;
    clnf->name = b->file->name.data;
    clnf->log = r->pool->log;
```

ngx_pool_cleanup_add 用于告诉 HTTP 框架，在请求结束时调用 cln 的 handler 方法清理资源。

至此，HTTP 模块已经可以向客户端发送文件了。下面介绍一下如何支持多线程下载与断点续传。

3.8.3　支持用户多线程下载和断点续传

RFC2616 规范中定义了 range 协议，它给出了一种规则使得客户端可以在一次请求中只下载完整文件的某一部分，这样就可支持客户端在开启多个线程的同时下载一份文件，其中每个线程仅下载文件的一部分，最后组成一个完整的文件。range 也支持断点续传，只要客户端记录了上次中断时已经下载部分的文件偏移量，就可以要求服务器从断点处发送文件之后的内容。

Nginx 对 range 协议的支持非常好，因为 range 协议主要增加了一些 HTTP 头部处理流程，以及发送文件时的偏移量处理。在第 1 章中曾说过，Nginx 设计了 HTTP 过滤模块，每一个请求可以由许多个 HTTP 过滤模块处理，而 http_range_header_filter 模块就是用来处理

HTTP 请求头部 range 部分的，它会解析客户端请求中的 range 头部，最后告知在发送 HTTP 响应包体时将会调用到的 ngx_http_range_body_filter_module 模块，该模块会按照 range 协议修改指向文件的 ngx_buf_t 缓冲区中的 file_pos 和 file_last 成员，以此实现仅发送一个文件的部分内容到客户端。

其实，支持 range 协议对我们来说很简单，只需要在发送前设置 ngx_http_request_t 的成员 allow_ranges 变量为 1 即可，之后的工作都会由 HTTP 框架完成。例如：

```
r->allow_ranges = 1;
```

这样，我们就支持了多线程下载和断点续传功能。

3.9 用 C++ 语言编写 HTTP 模块

Nginx 及其官方模块都是由 C 语言开发的，那么能不能使用 C++ 语言来开发 Nginx 模块呢？ C 语言是面向过程的编程语言，C++ 则是面向对象的编程语言，面向对象与面向过程的优劣这里暂且不论，存在即合理。当我们由于各种原因需要使用 C++ 语言实现一个 Nginx 模块时（例如，某个子功能是用 C++ 语言写成，或者开发团队对 C++ 语言更熟练，又或者就是喜欢使用 C++ 语言），尽管 Nginx 本身并没有提供相应的方法支持这样做，但由于 C 语言与 C++ 语言的近亲特性，我们还是可以比较容易达成此目的的。

首先需要弄清楚相关解决方案的设计思路。

- 不要试图用 C++ 编译器（如 G++）来编译 Nginx 的官方代码，这会带来大量的不可控错误。正确的做法是仍然用 C 编译器来编译 Nginx 官方提供的各模块，而用 C++ 编译器来编译用 C++ 语言开发的模块，最后利用 C++ 向前兼容 C 语言的特性，使用 C++ 编译器把所有的目标文件链接起来（包括 C 编译器由 Nginx 官方模块生成的目标文件和 C++ 编译器由第三方模块生成的目标文件），这样才可以正确地生成二进制文件 Nginx。
- 保证 C++ 编译的 Nginx 模块与 C 编译的 Nginx 模块互相适应。所谓互相适应就是 C++ 模块要能够调用 Nginx 框架提供的 C 语言方法，而 Nginx 的 HTTP 框架也要能够正常地回调 C++ 模块中的方法去处理请求。这一点用 C++ 提供的 extern “C” 特性即可实现。

下面详述如何实现上述两点内容。

3.9.1 编译方式的修改

Nginx 的 configure 脚本没有对 C++ 语言编译模块提供支持，因此，修改编译方式就有以下两种思路：

1）修改 configure 相关的脚本。

2）修改 configure 执行完毕后生成的 Makefile 文件。

我们推荐使用第 2 种方法，因为 Nginx 的一个优点是具备大量的第三方模块，这些模块都是基于官方的 configure 脚本而写的，擅自修改 configure 脚本会导致我们的 Nginx 无法使用第三方模块。

修改 Makefile 其实是很简单的。首先我们根据 3.3.2 节介绍的方式来执行 configure 脚本，之后会生成 objs/Makefile 文件，此时只需要修改这个文件的 3 处即可实现 C++ 模块。这里还是以 mytest 模块为例，代码如下。

```
    CC =   gcc
    CXX = g++
    CFLAGS =   -pipe   -O -W -Wall -Wpointer-arith -Wno-unused-parameter -Wunused-
function -Wunused-variable -Wunused-value -Werror -g
    CPP =   gcc -E
    LINK =  $(CXX)

    ...
    objs/addon/httpmodule/ngx_http_mytest_module.o: $(ADDON_DEPS) \
            ../sample/httpmodule/ngx_http_mytest_module.c
            $(CXX) -c $(CFLAGS)  $(ALL_INCS) \
                    -o objs/addon/httpmodule/ngx_http_mytest_module.o \
                    ../sample/httpmodule/ngx_http_mytest_module.cpp
    ...
```

下面解释一下上述代码中修改的地方。

- 在 Makefile 文件首部新增了一行 CXX = g++，即添加了 C++ 编译器。
- 把链接方式 LINK = $(CC) 改为了 LINK = $(CXX)，表示用 C++ 编译器做最后的链接。
- 把模块的编译方式修改为 C++ 编译器。如果我们只有一个 C++ 源文件，则只要修改一处，但如果有多个 C++ 源文件，则每个地方都需要修改。修改方式是把 $(CC) 改为 $(CXX)。

这样，编译方式即修改完毕。修改源文件后不要轻易执行 configure 脚本，否则会覆盖已经修改过的 Makefile。建议将修改过的 Makefile 文件进行备份，避免每次执行 configure 后重新修改 Makefile。

注意　确保在操作系统上已经安装了 C++ 编译器。请参照 1.3.2 节中的方式安装 gcc-c++ 编译器。

3.9.2 程序中的符号转换

C 语言与 C++ 语言最大的不同在于编译后的符号有差别（C++ 为了支持多种面向对象特性，如重载、类等，编译后的方法名与 C 语言完全不同），这可以通过 C++ 语言提供的 extern“C”{} 来实现符号的互相识别。也就是说，在 C++ 语言开发的模块中，include 包含的 Nginx 官方头文件都需要使用 extern“C”括起来。例如：

```
extern "C" {
    #include <ngx_config.h>
    #include <ngx_core.h>
    #include <ngx_http.h>
}
```

这样就可以正常地调用 Nginx 的各种方法了。

另外，对于希望 Nginx 框架回调的类似于 ngx_http_mytest_handler 这样的方法也需要放在 extern“C”中。

3.10 小结

本章讲述了如何开发一个基本的 HTTP 模块，这里除了获取请求的包体外没有涉及异步处理问题。通过本章的学习，读者应该可以轻松地编写一个简单的 HTTP 模块了，既可以获取到用户请求中的任何信息，也可以发送任意的响应给用户。当然，处理方法必须是快速、无阻塞的，因为 Nginx 在调用例子中的 ngx_http_mytest_handler 方法时是阻塞了整个 Nginx 进程的，所以 ngx_http_mytest_handler 或类似的处理方法中是不能有耗时很长的操作的。