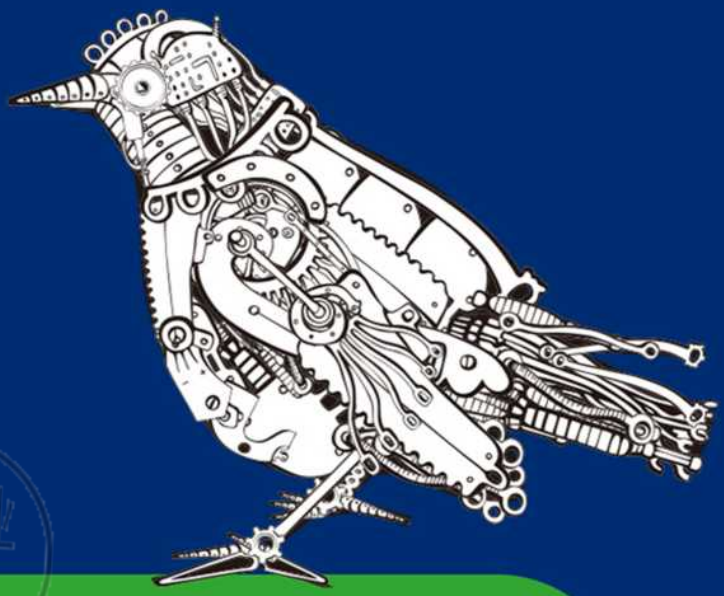
本书深入浅出地介绍了Nginx+Lua在实战场景中的各种使用技巧和方法，涉及Nginx配置、常用模块、缓存系统、日志分析、静态容灾、反向代理、爬虫、性能分析与优化等众多方面，掌握这些知识有助于提升你所开发的服务的性能。

Broadview
www.broadview.com.cn



Nginx 实战

基于Lua语言的配置、开发与架构详解

王力 汤永全 著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Nginx 实战

基于Lua语言的配置、开发与架构详解

王力 汤永全 著

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

内 容 简 介

本书主要讲解了 Nginx 在反向代理和应用开发中的作用，阅读本书可以了解 Nginx 在互联网开发中扮演的多个角色，充分利用这些角色的各项功能有助于提升服务的整体性能。本书所介绍的大部分功能是通过 Nginx+Lua 进行开发和配置的，但并不要求读者精通 Lua，在必要的位置，本书会对 Lua 进行选择性讲解。涉及实战的内容会有配套源码，方便读者学习和使用。

本书适合广大运维人员和开发人员学习，对使用 Nginx 完成各种服务架构感兴趣的架构师也可以阅读本书。阅读本书需要有对 Nginx 的初级或中级配置经验。

图书在版编目（CIP）数据

Nginx 实战：基于 Lua 语言的配置、开发与架构详解 / 王力，汤永全著. —北京：电子工业出版社，2019.3
ISBN 978-7-121-35460-1

Ⅰ. ①N… Ⅱ. ①王… ②汤… Ⅲ. ①互联网络－网络服务器－程序设计 Ⅳ. ①TP368.5

中国版本图书馆 CIP 数据核字（2018）第 254599 号

策划编辑：付 睿
责任编辑：牛 勇
印 刷：三河市华成印务有限公司
装 订：三河市华成印务有限公司
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编：100036
开 本：787×980 1/16 印张：21.5 字数：465 千字
版 次：2019 年 3 月第 1 版
印 次：2019 年 3 月第 1 次印刷
印 数：2500 册 定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

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

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

前　言

Nginx 自 2004 年发布第一个公开版本以来，就因其稳定性强、配置灵活、占用内存少、反向代理功能强大，而被越来越多的人喜爱和使用。随着人们对 Nginx 关注度的不断上升，Nginx 有了更多的使用场景，但在大多数公司中仍然只是扮演着反向代理的角色。

笔者在折 800 电商平台（以下简称折 800）工作多年，深感对一个电商平台来说，在成本控制和服务性能之间取得平衡是能够可持续发展的前提之一。与同类型且流量级别相近的公司相比，折 800 的计算机硬件成本要低很多，这主要得益于公司对软件技术的极致追求。在这样的背景下，我们热衷于研究 Nginx 反向代理等小众技术，并使用 Nginx 实现了大量功能，极大地提升了服务的性能和灵活度。

目前，Nginx 在国内的影响力还比较有限，因此很多开发人员并不了解 Nginx 的魅力，笔者写这本书的目的就是和大家分享 Nginx 的使用技巧，并一起来推广 Nginx。

市面上已经有一些介绍 Nginx 技术的书籍，但大多偏向于对 Nginx 配置和原理的讲解，还有一些书籍是基于 C 或 C++来介绍 Nginx 模块的。这样的书籍要求读者对 Nginx 的原理和源码有所了解，甚至还要有深厚的 C 语言或 C++语言的开发功底，这对大多数初学者来说要求太高了。

怎样才能做到既发挥出 Nginx 的威力，又尽量降低学习成本呢？笔者选择使用 OpenResty 的核心组件，因为 OpenResty 是 Nginx+Lua 的完美结合，它既能够实现 Nginx 的异步功能，又兼有 Lua 的易学优势，这样，在使用 Nginx 进行开发的过程中就不需要深入了解 Nginx 的原理了（如果已经掌握了 Nginx 原理当然更好）。所以，与市面上的大多数同类书籍不同，本书不会过多介绍 Nginx 配置和原理方面的内容，学习难度较低。

近几年来，OpenResty 的热度不断上升，各大互联网公司纷纷尝试使用 Nginx+Lua 的开发模式，在反向代理、网关系统、API 调度服务中都能看到这种开发模式的“身影”。希望本书可以让更多的开发人员了解 Nginx+Lua 的魅力，也让更多技术团队能够合理地使用 Nginx，降低硬件的投入成本，同时提升服务的性能。

本书由王力、汤永全著。全书内容共分 18 章，其中第 1~5 章介绍 Nginx 常见的配置方式，属于 Nginx 的入门知识；第 6~10 章对 Nginx+Lua 核心功能进行解读，是熟悉 Nginx+Lua 开发模式的必备知识；第 11~18 章是 Nginx+Lua 开发模式在实际业务中的实践应用，是灵活运用该模式的具体体现。

感谢彭赫、杨明翰、冯浩、刘凯、屈耀华对本书的支持；也感谢折 800 技术平台，在这里我得到了持续的历练和成长。同时，感谢电子工业出版社博文视点的编辑付睿和崔志伟，他们在本书的语言表述方面给了很多建议。

本书包含了作者的技术实践，如果你对本书内容有任何建议和疑惑，可以发电子邮件至 leehomewl@gmail.com。谢谢！

王力

2018 年 12 月

目　　录

第 1 章
Nginx 学前必知

Nginx 是一个以高性能、高并发著称的 HTTP 服务器，它支持 HTTP（HyperText Transfer Protocol，超文本传输协议）反向代理、TCP（Transmission Control Protocol，传输控制协议）代理、负载均衡、HTTP 缓存及 Web 开发等。本书将围绕 Nginx 在这些方面的特性进行讲解，希望随着我们对 Nginx 配置和开发的深入理解，能够充分发挥它在开发中的作用，让它不只是作为一个反向代理而存在。

在开始学习之前，首先需要了解 Nginx 的一些基础属性，为后续的学习做好准备。当然，如果你已是资深的开发人员，可以跳过本章。

本书将以 Nginx 1.12.2 版本为例进行讲解，并在 CentOS 6 以上的系统上进行实践。

1.1 HTTP 请求报文

首先，很有必要了解一下 HTTP 请求报文。当 Nginx 提供 HTTP 服务时，在 HTTP 请求中由客户端传出的内容就是 HTTP 请求报文。HTTP 请求报文由 3 部分组成。

- 请求行，包括请求方法、请求 URL（Uniform Resoure Locator，统一资源定位符）、HTTP 及其版本号。需要注意的是 GET、HEAD、POST 等请求方法支持 HTTP 1.0 和 HTTP 1.1 版本；而 PUT、DELETE、CONNECT、OPTIONS、TRACE、PATCH 等请求方法只支持 HTTP 1.1 版本。
- 请求头，指客户端向服务器传递请求时附加的一些信息，由 key/value 组成。key 和 value 用冒号隔开，每行一对，请求头常见的 key 有 Cookie、User_Agent、Accept-Encoding 等。

- 请求体，一般由 POST 请求方法进行提交，可能是图片、文件或字符串。

1.2 HTTP 响应报文

HTTP 响应报文也是一个必备知识点。客户端发出请求后，服务器接收到请求并返回给客户端的内容称为 HTTP 响应报文。HTTP 响应报文也由 3 部分组成，它们分别是响应行、响应头和响应体。

- 响应行，包括 HTTP 版本号、状态码和状态码描述。其中 HTTP 状态码说明见表 1-1。

表 1-1 HTTP 状态码说明

状 态 码	作 用
1XX	表示请求已经被接收，正在继续处理。这种响应是临时响应，不会返回响应体
2XX	表示请求被服务器接收并已完成处理过程
3XX	重定向，告知客户端需要继续执行操作才可以完成请求
4XX	出现问题，和客户端有关。如 401 表示权限问题，404 表示访问了一个不存在的 URL
5XX	出现问题，和服务端有关。如 500 表示内部错误，504 表示请求超时

- 响应头，为响应报文附加的额外信息，和请求头相似。区别在于响应头返回给客户端，而请求头是从客户端发起的。常见的响应头 key 有 Content-Type 和 Content-Encoding。
- 响应体，指请求返回到客户端的正文数据。

下面是一个完整的响应报文示例：

```
HTTP/1.1 200 OK          # HTTP 版本号、状态码、状态码描述，用空格分开
Server: nginx/1.12.2                                          # 响应头
Date: Sun, 01 Jul 2018 03:10:11 GMT                           # 响应头
Content-Type: application/octet-stream                        # 响应头
Transfer-Encoding: chunked                                    # 响应头
Connection: keep-alive                                        # 响应头
{"test":"Nginx","hello":"world!"}                             # 响应体
```

响应报文并非只有后端服务器才能发送，有时 Nginx 也可以作为服务器对请求报文的内容进行响应（通过使用 return、echo 等指令），后续章节会有说明。

1.3 安装 Nginx

Nginx 支持目前几乎所有主流的服务器系统，它的包管理安装方式和系统有关，并已集成

到系统源码里面，例如在 Ubuntu 下可以使用 sudo apt-get install nginx 进行安装，这种安装方式对定制模块和插件不太友好，所以下面会直接使用源码安装。

以 CentOS 操作系统为例，首先，Nginx 需要依赖一些 lib 库，请先安装如下所示的包：

```
    # yum -y install wget  gcc gcc-c++ autoconf automake make  zlib zlib-
devel  pcre-devel pcre
```

然后，下载源码：

```
    # wget https://nginx.org/download/nginx-1.12.2.tar.gz
```

最后，使用默认配置进行安装：

```
    # cd nginx-1.12.2
    # ./configure
    # make && make install
```

这样，Nginx 就安装完成了，很显然默认配置太简陋了，不是我们想要的结果。Nginx 提供了很多配置供我们自定义初始环境。自定义初始环境需要用到./configure 命令。./configure 命令的常见参数说明见表 1-2。

表 1-2　./configure 命令的常见参数说明

参　数	说　明
--prefix=PATH	设置 Nginx 的存放目录，默认目录为/usr/local/nginx
--conf-path=PATH	设置配置文件路径，默认为/usr/local/ nginx/conf/nginx.conf 也可以在 Nginx 启动服务时通过参数-c 指定文件路径
--with-threads	开启 Nginx 线程池，主要是为了提升 Nginx 读取硬盘的性能
--with-file-aio	在 Linux 2.6.22 以上的版本中启用异步 I/O
--with-http_gzip_static_module	启用 ngx_http_gzip_module 压缩模块，一般用来压缩响应信息，以节约带宽
--with-http_realip_module	一般用来获取客户端的真实 IP 地址
--with-http_ssl_module	启用 HTTPS 支持
--without-http_gzip_module	禁用 ngx_http_gzip_module --without-http_[模块名]可以禁用在编译时不需要的模块

执行./configure --help 命令可以看到更多帮助说明。

第一次接触 Nginx 的读者，可以先尝试默认的安装方式。安装成功后，在默认路径下找到 conf 目录，会看到一个 nginx.conf 文件。该文件的绝对路径通常为/usr/local/nginx/conf/nginx.conf，执行 vim 命令可以查看该文件。

1.4 支持 HTTPS

支持 HTTPS 是提高网站安全性的必要技术手段之一。可以使用系统已有的 OpenSSL 的 lib 包来激活 Nginx 支持 HTTPS，激活方式如下：

```
# ./configure  --prefix=/usr/local/nginx --with-http_ssl_module
# make && make install
```

但自从 OpenSSL 在 2014 年被曝出有致命漏洞后，这种方式就不那么安全了。况且对 CentOS 来说，默认安装包 OpenSSL 的版本较低且升级麻烦。所以最好使用源码进行编译安装，到 OpenSSL 的官网下载较新的稳定版，解压后编译进 Nginx 中，安装方式如下：

```
# ./configure  --prefix=/usr/local/nginx --with-http_ssl_module \
   —with-openssl=/path/openssl-1.0.2o
# make && make install
```

如果其他 lib 包需要通过源码编译安装，也可以使用类似的方式，如 PCRE 的 lib 包，需要执行--with-pcre=DIR 命令；而 zlib 的 lib 包则要执行--with-zlib=DIR 命令。

1.5 添加模块

Nginx 是一个开源软件，很多资深使用者提供了大量第三方模块来扩展 Nginx 的使用范围。在自定义初始环境时，使用--add-module=PATH，就可以添加第三方包，其中 PATH 是包的路径。

这些模块在开发和使用时可能会依赖某个固定的 Nginx 版本，或者只在某些版本中测试过，因此，使用前需要留意模块的 Wiki 是否特别说明过支持哪些 Nginx 版本。

1.6 小结

本章对 Nginx 的安装、模块的添加和禁用进行了初步介绍，还介绍了 HTTP 请求报文和 HTTP 响应报文。在下一章中各位读者将会看到 Nginx 的配置是如何发挥代理作用的。

第 2 章
基础配置

和大多数软件一样，Nginx 也有自己的配置文件，但它又有很多与众不同的地方，本章就来揭开 Nginx 基础配置的面纱。

2.1 Nginx 指令和指令块

了解指令和指令块有助于大家了解配置的上下文，下面是一个配置模板示例：

```
Main 1;
events  {}
http    {
     Main 2;
     server  {
          Main 3;
          location  {}
     }
         server  {
          Main 3;
          location  {}
     }
  }
```

在这个配置模板中主要包含两种指令。

- 简单指令：由名称和参数组成，以空格分隔，以分号结尾。上述示例中的 Main 1、Main 2、Main 3 就是简单指令。
- 指令块：由名称和大括号{}内的附加指令组成，不以分号结尾。

在前面的配置示例中，http 块是全局参数，对整体产生影响；server 块是虚拟主机，主要对指定的主机和端口进行配置；location 块在虚拟主机下根据请求 URI（Uniform Resource Identifier，统一资源标识符）进行配置，URI 即去掉参数后的 URL。

简单指令在指令块中的配置存在一定的区段。有些简单指令不能在某些指令块中使用；而有些简单指令既可以在 http 块也可以在 server 块中配置，甚至可以在 location 块中配置。当某个变量同时出现在多个指令块中时，最终会以在最小指令块中的赋值为准。

例如，如果在 location 中设置 expires 为 1m，那么 expires 就会使用 location 中的设置；如果没有在 location 中设置 expires，那么 expires 的值则会使用 http 块中 的 1d。这有点类似于编程语言里的变量。

```
http   {
    expires 1d;  #简单指令
    #指令块 server
    server  {
        location / { expires 1m; }
    }
}
```

2.2 Nginx 基本配置说明

对指令和指令块有了初步了解之后，下面将根据 2.1 节中的配置示例，对 Nginx 的指令块进行逐一说明，以帮助读者理解每个指令块的作用。

2.2.1 main 配置

在 http 块之前的配置是全局参数，如 2.1 节配置示例中的 Main 1，全局参数对整个 Nginx 块都产生作用。下面是一个简单示例：

```
user  nobody;
worker_processes  1;
error_log  /var/log/error_log
worker_rlimit_nofile 1024;
events {
    worker_connections  1024;
    use epoll;
}
```

2.2.2　与客户端有关的配置

与客户端有关的配置主要在 http 块中设置，如 2.1 节配置中的指令 Main 1 和 server 块之间的 Main 2 就是对客户端进行的配置，其作用是处理与客户端相关的信息。客户端配置常用的指令见表 2-1。

表 2-1　客户端配置常用的指令

指　令	说　明
client_body_buffer_size	设置读取客户端请求体的缓冲区大小。如果请求体的大小大于缓冲区的大小，则整个或一部分请求体会被写入临时文件。在默认情况下，会为 32 位系统、x86-64 系统设置 8KB 的缓冲区，其他 64 位系统为 16KB 的缓冲区
client_body_temp_path	定义存储客户端请求体的临时文件目录，最多可以定义 3 个子集目录
client_body_timeout	定义读取客户端请求体的超时时间，即两个连续的读操作之间的时间间隔。如果超时 HTTP 会抛出 408 错误
client_header_buffer_size	设置客户端请求头的缓冲区大小，默认为 1KB
client_max_body_size	设置客户端请求的最大主体的大小，默认为 1MB
client_header_timeout	设置客户端请求头的超时时间
etag	如果设置为 on，表示静态资源自动生成 ETag 响应头
large_client_header_buffers	设置大型客户端请求头的缓冲区大小
keepalive_timeout	设置连接超时时间。服务器将在超过超时时间后关闭 HTTP 连接
send_timeout	指定客户端的响应超时时间
server_names_hash_bucket_size	设置 server_names（Nginx 中配置的全部域名）散列表的桶的大小，默认值取决于处理器缓存行的大小
server_names_hash_max_size	设置 server_names 散列表的最大值
server_tokens	启用或禁用在错误页面和服务器响应头字段中标识的 Nginx 版本
tcp_nodelay	启用或禁用 TCP_NODELAY 选项。只有当连接保持活动时，才会被启用
tcp_nopush	仅当 sendfile 时使用，能够将响应头和正文的开始部分一起发送

很多指令都可以在多个 Main 中配置，但不是所有的指令都可以出现在 2.1 节的配置中的 Main 1、Main 2 和 Main 3 里面。例如 client_body_timeout 可以在 http、server、location 块中设置，但 server_names_hash_bucket_size 却只能出现在 http 块中。

表 2-1 中的指令和客户端的请求操作有直接关系，熟悉这些指令对配置和优化 Nginx 有很大的帮助。

2.2.3　server 块

server 块即虚拟主机部分，如果请求中的 Host 头和 server_name 相匹配，则将请求指向对

应的 server 块，示例如下：

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

server_name 支持使用通配符正则表达式，支持配置多域名、服务名称。当有多个 server 块时，会存在匹配的优先级问题，优先级顺序如下：

1．精确的名字；

2．以*开头的最长通配符名称，如*.testnginx.com；

3．以*结尾的最长通配符名称，如 testnginx.*；

4．按照文件顺序，第 1 个匹配到的正则表达式；

5．如果没有匹配到对应的 server_name，则会访问 default_server。

2.2.4 location 块

location 块在 server 块中使用，它的作用是根据客户端请求 URL 去定位不同的应用。即当服务器接收到客户端请求之后，需要在服务器端指定目录中去寻找客户端所请求的资源，这就需要使用请求 URL 匹配对应的 location 指令。表 2-2 是 URL 在 location 块中的匹配规则说明。

表 2-2　URL 在 location 块中的匹配规则说明

配置格式	作　　用
location = /uri	= 表示精确匹配
location ^~ /uri	^~ 匹配以某个 URL 前缀开头的请求，不支持正则表达式
location ~	~ 区分大小写的匹配，属于正则表达式
location ~*	~* 不区分大小写的匹配，属于正则表达式
location /uri	表示前缀匹配，不带修饰符，但是优先级没有正则表达式高
location /	通用匹配，默认找不到其他匹配时，会进行通用匹配
location @	命名空间，不提供常规的请求匹配

表 2-2 中匹配的优先级顺序为如下。

“=”优先级最高，如果“=”匹配不到，会和“^~”进行匹配；继而是“~”，如果有多个“~”，则按照在文件里的先后顺序进行匹配；如果还匹配不到，则与“/uri”进行匹配；通用匹配“/”的优先级最低，如果找不到其他配置，就会进行通用匹配；“@”表示命名空间的位置，通常在重定向时进行匹配，且不会改变 URL 的原始请求。

建议：打开 Debug 模式并观察日志，会看到每个请求的执行过程，包括匹配到对应 location 的操作。

location 块也支持嵌套配置：

```
location /a {
        location /a {
[ configuration A]
        }
        location /a/b {
[ configuration AB]
        }
}
```

有些指令只能在 location 块中执行，主要有如下 3 个

- internal：表示该 location 块只支持 Nginx 内部的请求访问，如支持 rewrite、error_page 等重定向，但不能通过外部的 HTTP 直接访问。
- limit_except：限定该 location 块可以执行的 HTTP 方法，如 GET。
- alias：定义指定位置的替换，如可以使用以下配置。

```
location /a/ {
    alias /c/x/a/;
}
```

上述配置表示如果匹配到/a/test.json 的请求，在进入 location 块后，会将请求变成/c/x/a/test.json。

2.3　include 的使用

include 用来指定主配置文件包含的其他扩展配置文件。扩展文件的内容也要符合 Nginx 的格式规范。include 可以简化主配置文件，使之更易于读取。include 可以出现在全局参数、location 块、server 块等任何一个位置。

include 支持通配符，例如下面的配置会将后缀是.conf 的所有文件都加载到 Nginx 配置中：

```
include /usr/local/nginx/conf/vhost/*.conf;
```

因此，可以将 Nginx 配置成多个文件，并提取出相同的数据，从而精简配置，方便管理。

2.4　常见配置

前面的章节介绍了 Nginx 中的 main 指令、server 块、location 块、include，以及与客户端

相关的一些配置知识，本节将进一步介绍 Nginx 常见的配置及其实战技巧。

2.4.1 常见配置注解

```
user  www www;                          #定义运行 Nginx 的用户和用户组
worker_processes  2;                    #Nginx 进程数
worker_cpu_affinity auto;               #配置 Nginx 进程的 CPU 亲缘性
error_log  /var/log/error_log  info; #定义全局错误日志的类型，默认是 error
worker_rlimit_nofile 65535;             #一个 worker 进程最多能够打开的文件数量
pid /var/run/nginx.pid;                 #进程文件
worker_priority -10;                    #在 Linux 系统下资源使用的优先级
worker_shutdown_timeout  30;  #若在 30s 内 Nginx 无法“平滑”退出，则强行关闭进程

events {
    #单个进程的最大连接数（整个 Nginx 的最大连接数=单个进程的最大连接数连接数×进程数）
    worker_connections  10000;
    #epoll 用在 Linux 2.6 以上版本的内核高性能的网络 I/O 上
    #如果是在 FreeBSD 上，则用 kqueue 模型
    use epoll;
}

http {
    include       conf/mime.types;            #文件扩展名与文件类型映射表
    default_type  application/octet-stream;  #默认文件类型
    log_format main     '$remote_addr - $remote_user [$time_local]'
                         '"$request" $status $bytes_sent'
                         '"$http_referer" "$http_user_agent" '
                         ' "$http_cookie" ';  #定义日志格式
    client_header_buffer_size    1k;   #设置用户请求头所使用的 buffer 的大小
    large_client_header_buffers  4 4k;#当默认的缓冲区大小不够用时就会使用此参数
    server_names_hash_bucket_size 128; #设置 server_names 散列表的桶的大小，
                                       #在域名比较多的情况下，需要调整这个值
    client_header_buffer_size 32k;     #对上传文件大小的限制
    gzip on;                           #开启 gzip
    gzip _comp_level 6;                #设置压缩等级
    gzip _min_length  1100;            #设置允许压缩的页面最小字节数
    gzip _buffers     4 8k;            #设置系统需要获取多大的缓存用于存储 gzip
                                       #的压缩结果数据流。4 8k 代表按照原始数
                                       #据的大小，即以 8KB 为单位的 4 倍申请内存
    gzip _types   text/plain text/css; #匹配 MIME 类型进行压缩
    output_buffers   2 32k;            #设置用于从磁盘读取响应的缓冲区的数量和
                                       #大小。2 32k 代表按照原始数据的大小
                                       #即以 32KB 为单位的 2 倍申请内存空间
```

```
    sendfile         on;                #启用 sendfile()函数
    tcp_nopush       on;                #为了防止网络阻塞，需要开启 sendfile
    tcp_nodelay      on;                #为了防止网络阻塞，需要开启 sendfile
    keepalive_timeout 90s;              #长连接超时时间，单位是秒
    upstream backend {}                 #upstream 块，weight 代表权重
    server 192.168.1.12:8081 weight=3; #权重越高，则请求的比例越高
    server 192.168.1.13:8081 weight=2;

    server {
        listen       80;                                    #HTTP 监听端口
        server_name  your.example.com;                      #域名
        access_log   /var/log/nginx.access_log  main;      #访问日志记录
        charset  koi8-r;                                    #默认编码
        location / {
            proxy_pass         http://backend ;
            proxy_redirect     off;
            proxy_set_header   Host             $host;
            proxy_set_header   X-Real-Ip        $remote_addr;
            #后端服务器通过 X-Forwarded-For 获取用户的真实 IP 地址
            proxy_set_header    X-Forwarded-For  $proxy_add_x_forwarded_for;
    }
    error_page  404  /404.html;  #对后端服务器抛出的错误 404 进行页面重定向
        location /404.html {
            root  /spool/www;
        }
                            #匹配以 jpg、jpeg、gif 结尾的 URL，直接去系统文件读取
        location ~* \.(jpg|jpeg|gif)$ {
            root         /spool/www;
            expires 30d;    #浏览器保留缓存的时间，如有 CDN，则 CDN 也会进行缓存
        }
    }

}
```

2.4.2　常见配置实战技巧

学完常见配置的注解，相信读者已经对 Nginx 配置有了基本的认识。但在实际应用中，Nginx 配置可以通过巧妙的变化实现不同的功能，下面将会讲解在实战中应如何进行合理的配置。

- user

默认是 nobody，但如果使用 nobody，会导致没有权限执行写硬盘等操作。所以一般会选

择低于 root 级别的用户，如 www，并在 Linux 系统下禁止用户通过 SSH（Secure Shell，远程连接工具）登录服务器，以提高安全性。

- worker_processes

代表 worker 的进程数，一般情况下建议和服务器的 CPU（Central Processing Unit，中央处理器）核数相同；也可以配置 worker_processes auto（用于 Nginx 1.2.5 版本之后），它会自动根据 CPU 核数启动进程。

但在实际应用中，可能除 Nginx 外服务器还会同时运行其他多个服务，所以需要考虑服务器资源在不同服务上的分配，避免因进程启动过多导致过多的上下文切换。

- worker_cpu_affinity auto

这是在 Nginx 1.9.10 版本中添加的功能，表示可以根据服务器的 CPU 核数自动设置 CPU 亲缘性，以提升 Nginx 的性能。

- error_log & access_log

关于日志记录的配置。如果将 error_log 配置为 error 级别，可以减少不必要的日志记录；如果是测试环境可以设置为 info 级别。配置日志记录需要考虑硬盘的独立性，不要使用 Linux 的根分区，以避免出现大量的 I/O 影响 Linux 服务器的吞吐能力；要单独挂载到一个磁盘上，使用独立的 I/O。

另外，需要注意硬盘的使用寿命，关注 message 的日志，定期检查硬盘（Nginx 在记录日志时是异步处理的，因此不会因为硬盘问题导致请求异常，但会影响日志的记录）。

- worker_priority

配置 Nginx 在 Linux 服务器上使用资源的优先级，作为反向代理服务，Nginx 应该拥有极高的优先级，因此建议配置为-10。

- gzip_comp_level

配置压缩等级，等级最高为 9，等级越高压缩后的文件越小，但是消耗的 CPU 资源，也会越多。经测试，文件压缩等级为 7 和等级为 9 时，在文件大小上只有细微的差别，一般用 5~7 的等级就可以了。

- upstream 块

配置后端服务器，可以结合 proxy_next_upstream 等指令进行大量的优化。具体内容将会在以后的章节中进一步讲解。

- error_page

对错误进行重定向，在捕获后端服务器错误的状态码后，将请求重定向到其他位置，如友好提示页面。关于 error_page 强大的功能，将在后面的章节进行单独说明。

- location & root

通过 root 路径可以读取静态文件，在 Nginx 1.7.11 版本之前，当 Nginx 读取硬盘文件时，都是进行阻塞型操作；后来引入了线程池，为读取硬盘文件提供了非阻塞型的操作，极大地提升了硬盘 I/O 的读/写速度，也提升了 proxy_cache 的缓存能力。

Nginx 的常见配置在使用中有着不同的变化，熟悉 Nginx 的官方 Wiki 是发挥其巨大作用的前提。

2.5　内置变量

在客户端请求过程中，Nginx 提供了内置变量来获取 HTTP 或 TCP 的信息。充分了解这些内置变量，才能够对应用场景中的业务进行合理的配置，下面先来熟悉一下常见的内置变量。

注意：Nginx 1.9 之后的版本开始支持 TCP 代理，这使 Nginx 的功能更为丰富（后面会有单独的介绍和案例示范）。本书默认以 HTTP 代理为例进行讲解，中间涉及 TCP 的地方会有特别说明。

2.5.1　常见内置变量

常见内置变量的说明见表 2-3。

表 2-3　常见内置变量的说明

变　量　名	说　　明
$arg_name	指 URL 请求中的参数，name 是参数的名字
$args	代表 URL 中所有请求的参数
$binary_remote_addr	客户端地址以二进制数据的形式出现，通常会和限速模块一起使用
$body_bytes_sent	发送给客户端的字节数，不包含响应头
$bytes_sent	发送给客户端的总字节数
$document_uri	设置$uri 的别名
$hostname	运行 Nginx 的服务器名
$http_referer	表示请求是从哪个页面链接过来的
$http_user_agent	客户端浏览器的相关信息

续表

变 量 名	说 明
$remote_addr	客户端 IP 地址
$remote_port	客户端端口号
$remote_user	客户端用户名，通常在 Auth Basic 模块中使用
$request_filename	请求的文件路径，基于 root alias 指令和 URI 请求生成
$request_time	请求被 Nginx 接收后，一直到响应数据返回给客户端所用的时间
$request_uri	请求的 URI，带参数
$request	记录请求的 URL 和 HTTP
$request_length	请求的长度，包括请求行、请求头和请求正文
$server_name	虚拟主机的 server_name 的值，通常是域名
$server_port	服务器端口号
$server_addr	服务器的 IP 地址
$request_method	请求的方式，如 POST 或 GET
$scheme	请求协议，如 HTTP 或 HTTPS
$sent_http_name	任意响应头，name 为响应头的名字，注意 name 要小写
$realip_remote_addr	保留原来的客户地址，在 real_ip 模块中使用
$server_protocol	请求采用的协议名称和版本号，常为 HTTP/1.0 或 HTTP/1.1
$uri	当前请求的 URI，在请求过程中 URI 可能会被改变，例如在内部重定向或使用索引文件时
$nginx_version	Nginx 的版本号
$pid	worker 进程的 PID
$pipe	如果请求是 HTTP 流水线（pipelined）发送的，pipe 值为"p"，否则为"."
$connection_requests	当前通过一个连接获得的请求数量
$cookie_name	name 即 Cookie 的名字，可得到 Cookie 的信息
$status	HTTP 请求状态
$msec	日志写入时间。单位为秒，精度是毫秒
$time_local	在通用日志格式下的本地时间
$upstream_addr	请求反向代理到后端服务器的 IP 地址
$upstream_port	请求反向代理到后端服务器的端口号
$upstream_response_time	请求在后端服务器消耗的时间
$upstream_status	请求在后端服务器的 HTTP 响应状态
$geoip_city	城市名称，在 geoip 模块中使用

注意：随着 Nginx 版本的不断更新，会出现更多新的指令，更多与 Nginx 内置变量有关的解释可参考 Nginx 官方 Wiki。

2.5.2　常见内置变量实战技巧

Nginx 的内置变量主要用于日志记录和分析，以及业务逻辑的处理。下面将介绍一些常用内置变量的配置方式。

- $arg_name

```
location /  {
     if ($arg_at= '5') {
        proxy_pass http://b;
     }
     proxy_pass http://a;
}
```

上述代码的请求默认路径是 http://a，如果 URL 中的参数是 at=5，则路径变为 http://b。关于 if、location 的配置后面会有单独的章节进行说明。

- $body_bytes_sent 和$bytes_sent

这两个变量的值之差就是 HTTP 响应头的大小。如果两个值相差悬殊，那么响应头就很大，需要确保 proxy_buffer 设置了合适的大小，因为如果超过 proxy_buffer 设置的值，error.log 就会显示如下内容：

```
2017/11/24 11:49:06 [error] 8376#0: *28071 upstream sent too big header
while reading response header from upstream,
```

读者可以修改如下的参数值（具体大小需要根据自身需求来确定）来解决这个问题，例如：

```
proxy_buffers   4 256k;
proxy_buffer_size   128k;
proxy_busy_buffers_size   256k;
```

- $realip_remote_addr

在 Nginx 1.9.7 版本以后加入 ngx_http_realip_module 的变量，此变量可以获取用户的 IP 地址（如常见的代理或 CDN 的节点 IP 地址）。

- $request_time 和$upstream_response_time

$upstream_response_time 指的是在 Nginx 启用了 upstream 的情况下，从 Nginx 与后端建立连接开始到接收完数据然后关闭连接为止的时间。$request_time 指从接收到用户请求到发送完响应数据的时间，包括接收请求数据的时间、程序响应的时间和输出响应数据的时间。

如果要检查后端服务的性能，需要使用$upstream_response_time 的值。

- $uri 和$request_uri

$uri 记录的是执行一系列内部重定向操作后最终传递到后端服务器的 URL（不包含参数$args 的值）。

$request_uri 记录的是当前请求的原始 URL（包含参数），如果没有执行内部重定向操作，$request_uri 去掉参数后的值和$uri 的值是一样的。在线上环境中排查问题时，如果在后端服务中看到的请求和在 Nginx 中存放的$request_uri 无法匹配，可以考虑去 $uri 里面进行查找。

- $scheme

近几年来，HTTPS 非常流行，很多互联网企业都把 HTTP 切换成了 HTTPS，但是当用户手动输入网站地址时，很少会主动加上 https://。为了让用户的请求能够顺利跳转到 HTTPS，首先需要判断用户输入的是 HTTP 还是 HTTPS。$scheme 就具有此功能，如果用户输入的是 HTTP 可以通过重定向跳转到 HTTPS，示例代码如下：

```
if ($scheme = 'http') {
    rewrite ^/(.*)$     https://$host/$1 redirect;
}
```

2.6 小结

本章讲解了 Nginx 的一些基础配置，较为规范的格式使上手学习 Nginx 比较容易，这也是 Nginx 能够流行开来的一个原因。本章还讲了指令和指令块，这为后面的学习做好了铺垫。

第 3 章
强化基础配置

通过上一章的学习，各位读者已经了解了一些 Nginx 的配置方式和基础内容，本章会介绍更多的模块来丰富这些配置，但这仍属于基础配置，是 Nginx 配置的进阶内容。

注意：本书以开源的 Nginx 内容为主，对只有商业版 Nginx 支持的指令不进行说明。

3.1 牢记 Context

Context 是 Nginx 中每条指令都会附带的信息，用来说明指令在哪个指令块中使用，可以将 Context 理解为配置环境。

每个指令都拥有自己的配置环境，如果把配置环境记错了，或者在设计时未考虑配置环境的作用，就很可能导致配置无法达到预期。举例来说，观察图 3-1 所示的根据请求参数定制响应头的逻辑。

图 3-1　根据请求参数定制响应头的逻辑

根据图 3-1 的需求，有的读者可能会不假思索地写出如下代码：

```
if ($arg_id) {
    proxy_set_header X-id '1';    # 设置请求头
}
```

```
    proxy_pass http://backend
```

但启动 Nginx 却出现如下报错信息：

```
    nginx: [emerg] "proxy_set_header" directive is not allowed here in
/usr/local/nginx/conf/test.ngx.conf:69
```

通过阅读 Nginx 的官方 Wiki 可以找到出现错误的原因：

```
    Syntax: proxy_set_header field value;
    Default:
    proxy_set_header Host $proxy_host;
    proxy_set_header Connection close;
    Context: http, server, location
```

可知 proxy_set_header 所支持的配置环境（Context），只有 http、server、location 这 3 个指令块，不支持 if 语句。因此对于要用到的指令，使用前需要知道它的配置环境。

3.2 获取请求的 IP 地址

Nginx 会将客户端的 IP 地址信息存放在$remote_addr 变量中，但在生产环境下往往会有各种代理，让获取真实的 IP 地址变得困难重重。

3.2.1 获取用户的真实 IP 地址

大部分互联网公司的反向代理都会用到图 3-2 所示的 CDN 加速代理流程图。用户的请求并不直接和 Nginx 打交道，而是由 CDN（Content Delivery Network，即内容分发网络）传递给 Nginx。所以在默认情况下，Nginx 看到的客户端 IP 地址是经过 CDN 处理后的 IP 地址，这对日志的记录和分析、后端服务的业务逻辑都可能产生不良的影响。如果需要获取用户的真实 IP 地址，就要用到 realip 模块。

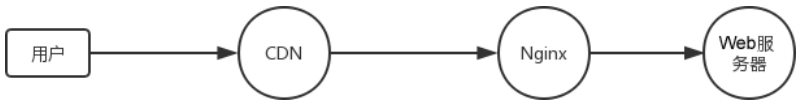


图 3-2 CDN 加速代理流程图

使用 realip 模块需要执行--with-http_realip_module 命令，并在 Nginx 的 http 块中配置如下代码：

```
    set_real_ip_from  CDN_IP;
    real_ip_header    X-Forwarded-For;
    real_ip_recursive on;
```

- set_real_ip_from：设置可信任的 IP 地址，即白名单，之后会使用 real_ip_header 从这些 IP 地址中获取请求头信息。
- real_ip_header：从指定的请求头中获取客户端的 IP 地址，IP 地址是通过请求头传递给 Nginx 的，请求头可能包括多个 IP 地址（以逗号分隔），此时只会获取最左边的 IP 地址并赋值给$remote_addr（客户端地址），此请求头一般会用到 X-Forwarded-For。
- real_ip_recursive：如果设置为 on，则表示启用递归搜索，realip_module 在获取客户端地址时，会在 real_ip_header 指定的请求头信息中逐个匹配 IP 地址，最后一个不在白名单中的 IP 地址会被当作客户端地址。默认为 off，表示禁用递归搜索，此时，只要匹配到白名单中的 IP 地址，就会把它作为客户端地址。

由此，可以获取到用户的真实 IP 地址，如果希望获取 CDN 的节点 IP 地址（有助于排查和 CDN 有关的问题），可以使用$realip_remote_addr（此变量在 2.5.2 小节中有介绍）。

3.2.2　防止 IP 地址伪造

X-Forwarded-For 请求头已经是业界的通用请求头，有些恶意攻击会伪造这个请求头进行访问，通过混淆服务器的判断来掩藏攻击者的真实 IP 地址。但如果在 Nginx 之前还有一个 CDN 的话，可以使用如下方案解决这个问题。

- 和 CDN 厂商确定约束规范，使用一个新的秘密的请求头，如 X_Cdn_Ip。
- 在 CDN 代理客户端请求时，要求 CDN 传递用户的 IP 地址，并且在建立连接的过程中，通过 CDN 清除伪造的请求头，避免透传到后端。图 3-3 是 CDN 传递用户 IP 地址的流程图。

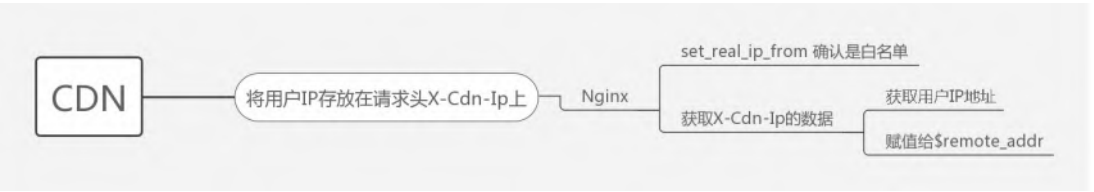


图 3-3　CDN 传递用户 IP 地址的流程图

注意：如果没有使用 CDN，客户端的请求会直接和 Nginx 打交道。Nginx 可以使用自定义的请求头传递用户的 IP 地址，如 proxy_set_header X-Real-IP $remote_addr。

3.2.3　后端服务器对 IP 地址的需求

有时后端服务器也要用到用户的客户端 IP 地址，在这种情况下，研发团队需要在 IP 地址

的获取上制定统一的规范，从规定的请求头信息中获取客户端 IP 地址。请求头中的 IP 地址可能有多个（经过了多次代理），它们以逗号分隔，其中第一个 IP 地址就是客户端 IP 地址。

3.3 管理请求的行为

作为 HTTP 服务，有些请求不适合暴露在公网上，那么就需要配置访问限制来提高安全性，此时可以通过 Nginx 来限制后台或内部的接口。

3.3.1 限制 IP 地址的访问

要对 IP 地址的访问进行限制，首先需要了解 allow 和 deny 这两个指令，allow 和 deny 指令的说明见表 3-1。

表 3-1 allow 和 deny 指令的说明

指　令	作　用	配置环境
allow	允许 IP 地址或 IP 地址段访问	http、server、location、limit_except
deny	禁止 IP 地址或 IP 地址段访问	http、server、location、limit_except

指令 allow 和 deny 都可以在多个指令块中配置，图 3-4 是指令 deny 在不同指令块中的配置效果，allow 的配置亦是如此。

图 3-4 指令 deny 在不同指令块中的配置效果

举个例子，对访问某个 location 块的 IP 地址进行限制，代码如下：

```
location / {
    deny  192.168.1.1;          # 禁止 192.168.1.1 访问
    allow 192.168.1.0/24;       # 允许 192.168.1.0/24 访问
    allow 17.1.1.2;             # 允许 17.1.1.2 访问
    deny  all;                  # 除 allow 的 IP 地址外，其他的 IP 地址都禁止访问
}
```

通过对访问的 IP 地址进行限制，可以阻挡可疑 IP 地址对服务的攻击，也可以确保内部接

口只被内网访问。

3.3.2　auth 身份验证

指令 allow 和 deny 基于 IP 地址来配置访问限制，除此之外，还可以通过密码验证的方式对访问进行限制，即通过配置 auth_basic 来设置用户须输入指定的用户名和密码才能访问相关资源。这样做既不用限制用户的 IP 地址，又在一定程度上保证了资源的安全。

```
server {
    listen       80;
    server_name  localhost;
    location / {
       auth_basic           " Nginx Basic ";
       auth_basic_user_file   conf/htpasswd;  # 存放密码的文件地址
    }
}
```

相关指令说明如下。

指令：auth_basic

语法：auth_basic string/off;

默认值：auth_basic off;

环境：http、server、location、limit_except

含义：其中 string 指的是字符串信息，是用户自定义的内容。如果设置为 off，表示禁用此功能；如果不设置为 off，则会在浏览器访问时看到 string 字符串的内容被输出。

指令：auth_basic_user_file

语法：auth_basic_user_file file;

默认值：无

环境：http、server、location、limit_except

含义：file 指的是文件名，该文件存放的是登录用户名和密码，形式类似于 testuser：1XlKs2P mC$xfxImYPQPMTloK5J7dar.1。

其中密码并不是明文显示的，而是加密过的。加密工具可以用 htpasswd 或 OpenSSL。OpenSSL 是进行 HTTPS 加密、解密时常用的工具，它也可以用来对密码进行加密，以账号 testuser、密码 Pass123 为例，执行加密命令如下：

```
# printf "testuser:$(openssl passwd -1 Pass123)\n" >> .htpasswd
```

得到的加密后的内容是 testuser:1DRCZTLTx$dRBMISe3SBnw/VZdBfhCg1，把它存放到 file 文件中，这样密码就更加安全了。重启 Nginx，打开浏览器输入 IP 地址进行访问就会显示如图 3-5 所示的界面。

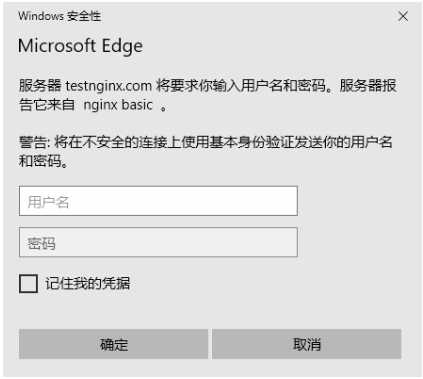


图 3-5　打开浏览器后显示的界面

注意：对于配置了密码加密的文件，一定要确保在 Nginx 进程中用户有可读权限，否则会出现的 403 错误。

3.3.3　利用 LDAP 服务加强安全

如果 auth_basic 使用统一的账号和密码会让请求无法对访问的用户进行区分，这对安全性要求较高的服务，还是不够安全，特别是当用户流动性较大时。此时，可以使用更精确的账号管理接口。常用的接口是 LDAP（Lightweight Directory Access Protocol，即轻量目录访问协议），LDAP 最基础的功能就是让每个用户都使用自己的账号和密码。通过配置 LDAP 认证，可以提升 Nginx 权限配置的灵活性。

首先，需要让 Nginx 能够读取 LDAP 的内容：

```
# yum  install openldap-devel -y
```

然后，添加对 LDAP 的支撑，在编译 Nginx 时，添加模块编译参数：

```
# git clone https://github.com/kvspb/nginx-auth-ldap      # 下载模块
# ./configure --add-module=path_to_http_auth_ldap_module  # 添加编译参数
# make
# make install
```

安装完成后，在 Nginx 的 http 块内配置如下内容：

```
        ldap_server testldap {
          URL  ldap://192.168.1.100:3268/DC=testnginx,DC=com?sAMAccountName?
```

```
sub?(objectClass=testuser);  # 此处的具体参数和参数值需要和搭建的 LDAP 服务相吻合
            binddn "TEST\\LDAPUSER";
            binddn_passwd LDAPPASSWORD;
            group_attribute uniquemember;
            group_attribute_is_dn on;
            require valid_user;
     }
```

最后，在 server 块中配置如下内容：

```
    server {
        listen       80;
        server_name  testnginx.com;
        auth_ldap "Forbidden";
        auth_ldap_servers testldap;
        location / {
            root   html;
        }
    }
```

配置了 LDAP 认证，就可以在人员变更后，快速更新用户的账号和密码，并可以定期通知使用者更新密码，加强安全管理。当然这不在本书讲解范围内，故不做深入介绍。

3.3.4　satisfy 二选一的访问限制功能

那么，如果希望在公司时不用输入账号和密码就能直接登录，该怎么办呢？很简单，加入 satisfy 指令，satisfy 指令可以提供二选一的逻辑判断，配置如下：

```
    satisfy any;
    auth_ldap "Forbidden";
    auth_ldap_servers testldap;

    allow 192.168.0.0/16;
    allow 10.10.10.10/32;
```

上述配置的作用是当请求的 IP 地址在 192.168.0.0/16 和 10.10.10.10/32 网段内时，不需要使用 LDAP 认证即可直接登录；如果 IP 地址不在这两个网段内，则需要通过 LDAP 认证进行登录。如此，鱼与熊掌可以兼得。

3.4　proxy 代理

Nginx 使用 ngx_http_proxy_module 来完成对后端服务的代理。这一节，我们将一起来见识 Nginx 最流行的 proxy 代理功能。

3.4.1 proxy_pass 请求代理规则

语法：proxy_pass URL;

环境：location、if in location、limit_except

含义：将请求代理到后端服务器，设置后端服务的 IP 地址、端口号以及 HTTP/HTTPS。

示例：将 URI 为/test 的请求代理到 127.0.0.1 上，端口号为 81，使用 HTTP，代码如下。

```
location = /test {
    proxy_pass http://127.0.0.1:81;
}
```

在代理过程中，URL 的传递会有如下几种变化：

```
location /test/v1/ {
    # 替换 URL 的代理方式： /test/v1/ 会被替换为/abc/
    # 如 /test/v1/xxx?a=1  到达后端就会变为 /abc/xxx?a=1
    proxy_pass http://127.0.0.1:81/abc/;
}
location /aaa/ {
    # 替换 URL 的代理方式，和上面的/test/v1/功能一样，只是替换成了"/"
    # 刚开始使用代理时 常常会和下面 /abc 的代理混淆
    proxy_pass http://127.0.0.1:81/;
}
location /abc {
    # 什么都不会改变，直接传递原始的 URL
    proxy_pass http://127.0.0.1:81;
}
```

注意：如果 location 块配置的 URI 使用了正则表达式，那么在使用 proxy_pass 时，就不能将 URI 配置到 proxy_pass 指定的后端服务器的最后面了，即禁止使用类似 proxy_pass http://127.0.0.1:81/abc/的方式，否则可能会导致一些不可预测的问题出现。

3.4.2 减少后端服务器的网络开销

有很多请求的内容只和 URL 有关，即后端服务器不需要读取请求体和请求头，只根据 URL 的信息即可生成所需的数据。在这种情况下，可以使用如下两个指令，并将其配置为 off，禁止传输请求体和请求头。

- proxy_pass_request_body：确定是否向后端服务器发送 HTTP 请求体，支持配置的环境有 http、server、location。
- proxy_pass_request_headers：确定是否向后端服务器发送 HTTP 请求头，支持的配置的

环境有 http、server、location。

通过配置以上两个指令，后端服务接收到的流量将会变小。

3.4.3　控制请求头和请求体

在请求被代理到后端服务器时，可以通过表 3-2 所示的指令去控制请求头和请求体。

表 3-2　请求头和请求体的控制指令

指令名称	作　　用	支持配置的环境
proxy_hide_header	禁止某个请求头被转发到后端服务器，如禁止转发 Cache-Control 请求头，则用 proxy_hide_header Cache-Control。默认情况下，"Date" "Server" "X-Pad" "X-Accel-..."都不会被转发	http、server、location
proxy_pass_header	允许已被禁止转发的请求头继续转发 如 proxy_hide_header Cache-Control	http、server、location
proxy_set_header	添加或修改请求头信息 如 proxy_set_header HOST 'www.abc.co'	http、server、location
proxy_set_body	对请求体进行覆盖，后端服务器收到的将会是被覆盖后的值 如 proxy_set_body 'b=123xxx'	http、server、location

注意：在设置 proxy_set_header 后，下一层级会继承这个请求头的内容。但如果下一层级也配置了 proxy_set_header 指令，那么当请求到达下一层级时，在上一层级配置的请求头将会被全部清除。

举例如下：

```
server {
    listen 80;
    proxy_set_header Host $host;
    proxy_set_header AB 'ab';

    location /abc {
    # 如果 location 块没有配置 proxy_set_header 指令，则会继承 server 层级的
    # proxy_set_header 指令。但现在它设置了一个请求头 A
    # 那么从 server 块传递过来的请求头 AB 就被清空了
    proxy_set_header A  'acv';
    proxy_pass http://127.0.0.1:81;
}
```

如果要 A 和 AB 两个请求头都保留下来，可以用下面的方法：

```
server {
    listen 80;
    proxy_set_header Host $host;
    proxy_set_header AB 'ab';
    location /abc {
       # 重新设置一次
       proxy_set_header Host $host;
       proxy_set_header AB 'ab';
       proxy_set_header A 'acv;'
       proxy_pass http://127.0.0.1:81;
    }
```

3.4.4 控制请求和后端服务器的交互时间

控制请求和后端服务器交互时间的指令见表 3-3。

表 3-3 控制请求和后端服务器交互时间的指令

指令名称	作　用	支持的配置环境
proxy_connect_timeout	设置请求和后端服务器建立连接的超时时间，默认是 60s，例如：proxy_connect_timeout 5s;	http、server、location
proxy_read_timeout	设置后端服务器读取响应的超时时间，即两个相邻请求之间的时间。若在规定时间内客户端没有收到返回内容，就会关闭连接，默认是 60s，例如：proxy_read_timeout 10s;	http、server、location
proxy_send_timeout	设置请求发送到后端服务器的超时时间，即两个相邻请求之间的时间，如果在规定时间内后端服务器没有收到任何请求，就会关闭连接，默认是 60s，例如：proxy_send_timeout 10s;	http、server、location

如果使用默认的设置，即 60s，请求可能需要等待很久才会做出下一步反应，而客户端往往不会等待那么久，所以需要合理设置交互时间，并且最好能在超时后做一些合理的措施。如搭配使用 proxy_next_upstream*命令，这将在下一节进行说明。

3.5 upstream 使用手册

利用 proxy_pass 可以将请求代理到后端服务器，上一节中的配置示例都指向同一台服务器，如果需要指向多台服务器就要用到 ngx_http_upstream_module。它为反向代理提供了负载均衡及故障转移等重要功能。

3.5.1　代理多台服务器

先来看一个简单的版本：

```
# 定义一个 HTTP 服务组
upstream test_servers {
  # 用 server 定义 HTTP 地址
  server 127.0.0.1:81  max_fails=5 fail_timeout=10s weight=10;
  server 127.0.0.1:82  max_fails=5 fail_timeout=10s weight=5;
  server test.123.com  backup;
  server 127.0.0.1:82  down;

}
server {
    listen 80;
    location / {
       # 通过代理将请求发送给 upstream 命名的 HTTP 服务
       proxy_pass http://test_servers;
    }
}
```

指令：upstream

语法：upstream name { ... }

环境：http

含义：定义一组 HTTP 服务器，这些服务器可以监听不同的端口，以及 TCP 和 UNIX 套接字。在同一个 upstream 中可以混合使用不同的端口、TCP 和 UNIX 套接字。

指令：server

语法：server address [parameters];

环境：upstream

含义：配置后端服务器，参数可以是不同的 IP 地址、端口号，甚至域名。

server 指令拥有丰富的参数，其参数说明见表 3-4。

表 3-4　server 指令参数说明

参　数	作　用
weight	设置请求分发到后端服务器的权重，即每台后端服务器能够响应的请求数量的比例。如果设置相同的比例，意味着每台后端服务器能够响应的请求数量是一样的，默认值是 1。如在前面的配置示例中，假如有 30 个请求进入服务器，那么第 1 台会进入 20 个请求，第 2 台会进入 10 个请求

续表

参　数	作　用
max_fails	请求最大失败次数，在指定时间内请求失败的最大次数，默认是 1。如果设置为 0，代表禁用这个设置
fail_timeout	在指定时间内请求失败的次数，超过这个次数则认为服务器不可用。如在前面的配置示例中 fail_timeout =10s，即如果 10s 内请求失败次数超过 max_fails 的值，则在接下来的 10s 内，此 server 不再接收任何请求，10s 后重新恢复为可用，并重新计算失败次数
down	标记服务不可用
backup	当 upstream 中所有的后端服务器都被设置为不可用时（如全都超过了请求最大失败次数），upstream 会对 backup 的服务器进行分流

3.5.2　故障转移

如果在前面的配置示例中出现了超过请求失败次数的服务器，下面这些参数可以用来对这些服务器进行配置：proxy_next_upstream、fastcgi_next_upstream、uwsgi_next_upstream、scgi_next_upstream、memcached_next_upstream 和 grpc_next_upstream。

下面用最常见的 proxy_next_upstream 为例进行说明。

指令：proxy_next_upstream

语法：proxy_next_upstream error | timeout | invalid_header | http_500 | http_502 | http_503 | http_504 | http_403 | http_404 | http_429 | non_idempotent | off ...;

默认值：proxy_next_upstream error timeout;

环境：http、server、location

含义：定义转发条件，当请求返回 Nginx 时，如果 HTTP 状态满足 proxy_next_upstream 设置的条件，就会触发 Nginx 将请求重新转发到下一台后端服务器，并累加出现此状态的服务器的失败次数（当超过 max_fails 和 fail_timeout 的值时就会设置此服务器为不可用）。如果设置为 off，则表示关闭此功能。

指令：proxy_next_upstream_tries

语法：proxy_next_upstream_tries number;

默认值：proxy_next_upstream_tries 0;

环境：http、server、location

含义：定义尝试请求的次数，达到次数上限后就停止转发，并将请求内容返回客户端。

指令：proxy_next_upstream_timeout

语法：proxy_next_upstream_timeout time;

默认值：proxy_next_upstream_timeout 0;

环境：http、server、location

含义：限制尝试请求的超时时间，如果第一次请求失败，下一次请求就会被此参数值控制。若设置为 0，则表示无超时时间，但尝试的请求仍会受到 proxy_read_timeout、proxy_send_timeout、proxy_connect_timeout 的影响。

注意：通过这些配置，可以在后端服务器的某些节点出现请求异常时，快速做出故障切换的操作，从而屏蔽这些异常的请求。但是这存在一种隐患，即如果 proxy_next_ upstream_tries 设置的值比较大，且 proxy_next_upstream 也设置了很多状态，当发生大面积异常时，重试不断累加，可能会导致请求反复向多个服务器发送，这样会给后端服务器带来更大的压力。

3.5.3　负载均衡

Nginx 不仅支持代理多台后端服务器，也支持各种负载均衡模式，负载均衡在 upstream 的配置环境内设置（默认根据权重轮询）。负载均衡指令见表 3-5。

表 3-5　负载均衡指令

指　令	用　途
hash	按照指定的 key 将请求分布到后端服务器上，key 可以是变量、文本或它们的组合，如$request_uri 为 hash、$user_agent，key 相同的请求会被代理到同一台后端服务器上
ip_hash	根据 IP 地址将请求分流到后端服务器上，同一个 IP 地址的请求会被代理到同一台后端服务器上。如需移除其中一台后端服务器，建议使用 down 对服务器设置停止分流，这样可以保留当前 IP 地址的 hash
least_conn	当将请求分流到后端服务器时，请求量最小的服务器会优先获得分流。如果配置的服务器很多，会使用增加权重的 round-robin 负载方式
sticky	根据 Cookie 将请求分布到后端服务器上，同一个 Cookie 的请求只会进入同一台服务器。如果请求被分布到某台服务器上，但是在请求时这台服务器已经无法提供服务，那么会重新选择一台服务器进行“捆绑”，并且下次会直接进入重新“捆绑”的服务器

3.5.4　通过 hash 分片提升缓存命中率

缓存系统是减少后端服务压力的重要组件，常见的 HTTP 缓存系统有 Nginx 的 proxy_cache、varnish、squid。如果通过反向代理去获取缓存数据，一般需要使用 hash 分片，以避免 URL 的请求随机进入缓存系统的某个分片，导致缓存命中率低、后端服务器压力上升。

基于 URL 缓存的服务配置一般如下所示，相同的 URL（包含参数）会进入相同的后端缓存系统。

```
upstream test_servers {
  hash $request_uri;
  server 127.0.0.1:81  max_fails=5 fail_timeout=10s weight=10;
  server 127.0.0.1:82  max_fails=5 fail_timeout=10s weight=5;
}
```

注意：

- 增减节点会导致 hash 重新计算，因此增减节点最好选择在服务的低峰期进行。
- 在缓存系统上使用 max_fails 不一定是最好的选择，但一旦使用请确保 proxy_next_upstream 的合理性，尽量不要配置各种 HTTP 状态码，因为缓存系统代理的是后端服务，当后端服务异常时会将错误的状态码返回给 Nginx，这样会让 Nginx 以为缓存系统出了问题，从而将缓存节点当作失败的节点，停止分流。

缓存系统的故障转移应该只以存活检查方式（一般指检查缓存系统的端口是否存活，以及固定检查一个接口是否能返回正常的响应）为主。可以结合健康检测功能，或者动态剔除异常缓存节点的功能来使用，详细介绍请看后续章节。

3.5.5 利用长连接提升性能

在 Nginx 中，使用 upstream 进行后端访问默认用的是短连接，但这会增加网络资源的消耗。可以通过配置长连接，来减少因建立连接产生的开销、提升性能。和长连接有关的配置示例如下：

```
keepalive_requests 1000;
keepalive_timeout 60;

upstream test_servers {
  server 127.0.0.1:81  max_fails=5 fail_timeout=10s weight=10;
  server 127.0.0.1:82  max_fails=5 fail_timeout=10s weight=5;
  keepalive 100;
}
server {
    listen 80;
    proxy_set_header Host  $Host;
    proxy_set_header x-forwarded-for $remote_addr;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_http_version 1.1;
    proxy_set_header Connection "";
```

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

长连接配置指令说明见表 3-6。

表 3-6　长连接配置指令说明

指　令	作　用	支持配置的环境
keepalive_requests	设置每个连接的最大请求次数，超过这个次数就会关闭该连接建立新的连接	http、server、location
keepalive_timeout	设置 keep-alive 客户端连接在服务器端保持开启的超时时间	http、server、location
keepalive	设置 worker 进程和后端服务器之间保持空闲连接的最大值，如果空闲连接数大于这个值，将会关闭使用最少的连接	upstream
proxy_http_version 1.1	设置 HTTP 请求协议，要确保是 HTTP 1.1 的长连接协议	http、server、location
proxy_set_header Connection ""	清空 Connection 请求头，避免客户端传递短连接的请求头信息	http、server、location

注意：如果没有添加长连接，在压力测试（以下简称压测）环境中，可能会出现这样的情景：当压测达到一定的 QPS（Query Per Second，每秒查询率）后，Nginx 服务器突然“卡死”，QPS 直接降到几乎为 0，但是压测并没有停；几分钟后又会自动恢复，然后再压测一段时间后，QPS 又会突然降到接近于 0。这种情况就要考虑是不是 timewait 的状态过多了。

3.5.6　利用 resolver 加速对内部域名的访问

proxy_pass 可以直接将域名代理到后端服务器上，请求前会先解析出 IP 地址，例如：

```
proxy_pass http:// test2.zhe800.com:82
```

反复的 DNS（Domain Name System，域名系统）解析会影响请求的速度，并且如果出现连接 DNS 服务器超时的情况，可能会导致请求无法发送，这里需要用 DNS 缓存来解决这个问题，示例代码如下：

```
server {
    listen 80;

    location / {
        resolver 10.19.7.33 valid=30s;
        resolver_timeout 5s;
        set $upstream_host test2.zhe800.com;
        proxy_pass http://$upstream_host:82;
    }
}
```

resolver 指令说明见表 3-7。

表 3-7　resolver 指令说明

指　令	作　用	支持配置的环境
resolver	配置用来解析后端服务器的 DNS 地址，如 10.19.7.33:5353，如果不配置端口号，则默认是 53。参数 valid 用来设置 DNS 的缓存时间，当执行 resolver 指令时，在设定的缓存时间内不会触发 DNS 解析，而是使用之前的 IP 地址	http、server、location
resolver_timeout	设置解析的超时时间，如果超过这个超时时间请求会报错	http、server、location

注意：解析 DNS 后，通过 set $upstream_host test2.zhe800.com 的方式，将获取的 IP 地址再赋值给 proxy_pass，这是为了让 Nginx 重新去解析 DNS 中的 IP 地址。利用 valid 的配置，可以减少 DNS 的解析次数，从而提高请求的效率。当然对 DNS 缓存时间的控制也要有度，避免出现 DNS 切换 IP 地址后，Nginx 无法快速切换到新 IP 地址的情况。

如果需要在 upstream 内部对域名的配置进行解析，使用 Nginx 的开源版会受到一些限制，因为此功能被放到了商业版中，需要和 zone 的功能配合使用。

3.6　rewrite 使用手册

rewrite 是 ngx_http_rewrite_module 模块下的指令，使用频率非常高，本节会列举一些 rewrite 的常见配置。

3.6.1　内部重定向

rewrite 支持的配置环境有 server、location、if，它通过 break 和 last 来完成内部重定向功能。内部重定向是在 Nginx 内部发送请求的操作，它可以将请求转发到其他的 location 或对 URL 进行修改，而不必通过 HTTP 连接请求，整个操作非常高效。

相关示例如下：

```
# 匹配以/a 结尾的 URI，匹配成功后将其修改成/b 的 URI，即后端服务器看到的 URI 会是/b
# 并停止 rewrite 阶段，执行下一个阶段，即 proxy_pass
   rewrite /a$  /b break;

# 匹配以/a 开头的 URI，匹配成功后将其修改成/b 的 URI
# 并停止 rewrite 阶段，执行下一个阶段，即 proxy_pass
   rewrite ^/a  /b break;

# 匹配/a 的 URI，成功后将其修改成/b 的 URI
```

```
# 并停止 rewrite 阶段，执行下一个阶段，即 proxy_pass
   rewrite ^/a$   /b break;

# 匹配包含/a 的 URI，匹配成功后将其修改成/b 的 URI
# 并停止 rewrite 阶段，执行下一个阶段，即 proxy_pass
   rewrite /a    /b  break;

# 匹配以/a/开头的请求，并将/a/后面的 URI 全部捕获，(.*)的作用就是捕获全部 URI
# 然后重定向成 /b/$1 ,其中$1 就是前面捕获到的 URI。匹配成功后将其修改成/b 的 URI
# 并停止 rewrite 阶段，执行下一个阶段，即 proxy_pass

  rewrite ^/a/(.*)    /b/$1  break;

# last 的匹配规则和 break 完全一样，只是当它匹配并修改完 URI 后
# 会将请求从当前的 location 中“跳”出来，找到对应的 location
   rewrite /a /b last;
   proxy_pass http://test_servers;
```

如果需要将内部重定向的请求记录到日志里，请使用 rewrite_log。

指令：rewrite_log

语法：rewrite_log on | off;

默认值：rewrite_log off;

环境：http、server、location、if

注意：在 rewrite 后面跟随的参数始终是正则表达式，并且当内部重定向时 URL 的参数是不会丢失的。

3.6.2　域名跳转

通过 rewrite 可以实现域名间的跳转，常见的跳转类型有 301、302。

```
# permanent 参数表示永久重定向，将所有的请求全部跳转到指定域名上
# 通过(.*)将 URL 保留下来，跳转过程中参数不会丢失。HTTP 状态码为 301
  rewrite ^/(.*)$ http://www.zhe800.com/$1 permanent;

# redirect 参数表示临时重定向，将所有的请求全部跳转到指定域名上
# 通过(.*)将 URL 保留下来，跳转过程中参数不会丢失。HTTP 状态码 302
  rewrite ^/(.*)$ http://www.zhe800.com/$1 redirect;
```

永久重定向和临时重定向的区分是为爬取搜索引擎而设置的。如果状态码是 301，表示网页地址永久改变，搜索引擎会将旧页面的地址替换成新页面的地址，用户在搜索引擎搜索网站时，就不会再去旧地址了。

如果状态码是 302，表示临时重定向，搜索引擎会保留旧页面的地址，因为它认为跳转只是暂时的。当用户使用搜索引擎搜索网站时，会首先进入旧地址然后再被跳转到 302 指定的新地址。

3.6.3 跳转 POST 请求

301 和 302 的跳转并不适合用于 POST 请求，如果 POST 请求被跳转的话，会先被转化为 GET 请求，且请求体的内容会丢失，为此 HTTP 1.1 提供了新的跳转状态码 307 和 308。

状态码 307 的意义和 302 一样，都是临时重定向；而 308 和 301 一样，都是永久重定向。但如果要求在跳转过程中保持客户端的请求方法不变，需要使用 return 指令，示例如下：

```
return 307 http://www.zhe800.com/$request_uri;
return 308 http://www.zhe800.com/$request_uri;
```

指令：return

语法：return code [text];

其中 code 是状态码。return 指令会立即返回一个 HTTP 状态码给客户端，并附加一个文本信息到响应体中，因为此方式缺少 Default_Type 响应头，所以当使用浏览器打开时无法直接展示文本信息，而会将文本下载到本地。

return 指令用法示例如下。

- return code URL; 支持 301、302、303、307 和 308 跳转。
- return URL; 默认支持 302 跳转。

环境：server、location、if

注意：如果把 return 和 proxy_pass 配置为同一级别，那么会直接执行 return，而不会执行 proxy_pass，同一 location 块下的其他指令也不会再被执行。

3.6.4 设置变量的值

指令：set

语法：set $variable value;

环境：server、location、if

含义：设置指定变量的值，值可以是文本、变量或文本和变量的组合。因为是为当前请求设置的值，所以会在请求结束后被清除掉。

```
location / {
```

```
    set $a '1';
    set $b '2';
    set $ab  $a$b;      # 可以合并两个变量的值
    return 200 $ab;     # 输出为 12，状态码为 200
}
```

3.7　限速白名单

Nginx 提供了 ngx_http_limit_req_module 和 ngx_http_limit_conn_module 等模块来完成限制请求访问的功能，可以对请求的访问频率、User-Agent、带宽等各种条件进行限速。

例如当 User-Agent 有问题时，要对其进行限速，则代码如下：

```
# 当匹配到 MSIE 类型的用户时，客户端限速 2KB/s
if ($http_user_agent ~* "MSIE") {
  limit_rate 2k;
}
```

可以参考 Wiki，网上也有很多相关的文章，这里不再一一赘述。

限速一般会对请求的访问频率进行控制，但如果有些请求是内部访问不需要限速该如何处理呢？相关配置如下所示：

```
# geo 指令会获取当前请求的 IP 地址，然后在整个字典里面进行匹配，如果匹配成功
# 就会获取 IP 地址对应的变量的值，如下面的 0 和 1；如果匹配不成功，则会执行 default
# 获取到的参数值会赋值给$wip
http {
  geo $wip  {
     default 0;
     # 下面的 IP 地址属于白名单，不会被限速
     127.0.0.1 1;
     172.16.0.0/16 1;
     172.17.0.0/16 1;
     172.18.0.0/16 1;
     10.0.0.0/8 1;
   }
# map 指令，如果$wip 是 0，表示执行的是 default，此时会将$binary_remote_addr
# 赋值给$limit；  如果$wip 是 1，表示 IP 地址属于白名单
   map $wip  $limit {
      0 $binary_remote_addr;
      1 "";
   }
```

```
# 如果是白名单，则$limit 的值为空，限速时会忽略为空的请求
limit_req_zone $limit zone=zone_acode:100m rate=25r/s;
server {...}
}
```

注意：整个配置都在 http 块内。

3.8 日志

Nginx 通过 ngx_http_log_module 来对日志的记录进行配置。

3.8.1 记录自定义变量

指令：log_format

语法：log_format name [escape=default|json|none] string ...;

默认值：log_format combined "...";

环境：http

含义：配置日志的格式。

绝大部分请求信息都可以通过 Nginx 现有的变量来获取，例如常见的 Cookie、IP 地址、User_Agent、请求体大小、单个请求头、server_ip、后端节点和端口号等。当然还有自定义变量，如需将自定义变量存放到日志中，只需简单的两步。

- 使用自定义变量前需要先进行初始化：

```
set $a '123';
```

- 为自定义变量添加日志格式：

```
log_format  main  '$remote_addr - $remote_user [$time_local] $a'
```

3.8.2 日志格式规范

日志经常会被用来分析和查找问题，为了提升日志的可读性，需要规范日志的格式以减少解析日志时出现的麻烦。

在 Nginx 1.11.8 版本之后，提供了[escape=default|json|none] 配置，它可以让 Nginx 在记录变量时使用 JSON 格式或默认字符。如果使用默认字符则会被转义，特别是当 POST 请求中包含中文字符或需要转义的字符时，默认转义的操作会使日志无法记录明确的信息。

日志记录推荐使用如下格式：

```
log_format json_log escape=json '{"ip":"$remote_addr","timestamp":
            "$time_iso8601",'
            '"host":"$http_host","request":"$request",'
            '"cookie":"$http_cookie","req_time":"$request_time",
            "uri":"$uri","referer":"$http_referer" }';
```

使用 escape=json 则日志内容不会被转义，中文字符可以直接在日志里面显示。日志的格式采用 JSON 的方式进行配置，对后期的数据采集会有很大的帮助，当在日志里添加新的变量时，不会影响日志分析的流程。

3.8.3　日志存储

日志存储通过 access_log 来完成。

指令：access_log

语法：access_log path [format [buffer=size] [gzip[=level]] [flush=time] [if=condition]];

access_log off;

默认值：access_log logs/access.log combined;

环境：http、server、location、if in location、limit_except

- 在同一个阶段中，支持多种格式和文件的收集。
- 若不在同一个阶段，则会以最小配置阶段的 access_log 为指定文件。
- 支持压缩存储，可以减少硬盘的使用空间。
- 支持只记录某些状态。
- 支持关闭日志记录。

指令 access_log 只记录访问日志，关于错误信息的日志记录在 error_log 上，示例如下：

```
# json_log 定义了日志格式
access_log /data1/access.log  json_log;

# 日志压缩记录，每缓存 5MB 的数据就进行日志压缩
access_log /data1/access_1.log combined gzip flush=5m;

# 根据条件进行记录，当 if 等于 0 或为空时日志不会被记录
# 下面代码的意思是当 HTTP 状态码以 4 开头时，日志不会被记录
map $status $loggable {
    ~^[4]  0;
```

```
        default 1;
    }
    access_log /path/to/access.log combined if=$loggable;
```

注意：日志应该存放在独立的硬盘上，最好和系统盘及存放 Nginx 配置文件的硬盘相互独立，避免让日志硬盘的 I/O 影响服务器的其他服务。在高并发的情况下，可能每秒会生成几十兆的日志。

3.9 HTTP 执行阶段

Nginx 对请求的处理发生在多个 HTTP 执行阶段，前面介绍的指令都是在这些阶段中执行的，了解这些阶段的执行顺序和用途，对后续原生模块、第三方模块的学习，以及使用 Lua 开发新的功能都有非常重要的作用。

HTTP 执行阶段配置在 ngx_http_core_module 中，下载 Nginx 的源码包后在 src/http/ngx_http_core_module.h 中可以看到。

下面是 ngx_http_core_module.h 中的一段源码：

```
typedef enum {
    NGX_HTTP_POST_READ_PHASE = 0,

    NGX_HTTP_SERVER_REWRITE_PHASE,

    NGX_HTTP_FIND_CONFIG_PHASE,
    NGX_HTTP_REWRITE_PHASE,
    NGX_HTTP_POST_REWRITE_PHASE,

    NGX_HTTP_PREACCESS_PHASE,

    NGX_HTTP_ACCESS_PHASE,
    NGX_HTTP_POST_ACCESS_PHASE,

    NGX_HTTP_TRY_FILES_PHASE,
    NGX_HTTP_CONTENT_PHASE,

    NGX_HTTP_LOG_PHASE
} ngx_http_phases;
```

通过上述配置，可以看出 Nginx 包含 11 个阶段，当请求进入 Nginx 后，每个 HTTP 执行阶段的作用见表 3-8（按照执行顺序进行排列）。

表 3-8　HTTP 执行阶段的作用

阶段顺序	阶段名称	作　　用
1	NGX_HTTP_POST_READ_PHASE = 0	接收并读取请求阶段
2	NGX_HTTP_SERVER_REWRITE_PHASE	修改 URL 阶段，通常有重定向和变量设置的操作
3	NGX_HTTP_FIND_CONFIG_PHASE	查找 URL 对应的配置，如匹配 location
4	NGX_HTTP_REWRITE_PHASE	在匹配到对应的 location 后，再次进入修改 URL 阶段
5	NGX_HTTP_POST_REWRITE_PHASE	检查 URL 是否执行过阶段 4，如果执行过，就会重新执行阶段 3，每个请求的最大检查次数是 10，超过 10 次就会报错
6	NGX_HTTP_PREACCESS_PHASE	一般用来在请求前设置对资源的控制，例如限速
7	NGX_HTTP_ACCESS_PHASE	控制访问权限，例如限制某个 IP 地址的访问或外层密码的登录
8	NGX_HTTP_POST_ACCESS_PHASE	验证阶段 7 的权限控制结果
9	NGX_HTTP_TRY_FILES_PHASE	只有当使用 try_files 指令时才会生效
10	NGX_HTTP_CONTENT_PHASE	处理 HTTP 请求内容的阶段，一般会和后端服务器进行交互
11	NGX_HTTP_LOG_PHASE	日志记录阶段

3.10　小结

通过本章的学习，各位读者能够了解 HTTP 请求在经过 Nginx 时常见的处理方式和执行流程，它们一起构成了反向代理的基础配置。

第 4 章
常用模块精解

前面几章介绍了 Nginx 自带的指令和模块。随着 Nginx 的广泛使用，大量第三方开源模块也被引入进来，这为 Nginx 的功能锦上添花。本章的主要内容是对 Nginx 的常用模块特别是目前比较流行的第三方模块进行讲解。

注意：本书中出现的模块均在 Nginx 1.2 以上的版本中进行使用和测试，如果在其他版本上使用，请参考官方说明，避免出现兼容性问题。

4.1 定制 HTTP 头信息

定制 HTTP 头信息，是实际业务中一个很重要的功能。例如，如果需要将请求结果在浏览器上缓存一段时间，或者在请求代理到后端服务器的过程中生成一个唯一的 ID 进行识别。通过对 Nginx 进行配置，可以轻松实现这些功能。

4.1.1 使用 ngx_http_headers_module 设置响应头

ngx_http_headers_module 是在 Nginx 编译时默认自带的模块，主要包含 add_header 和 expires 两个指令。

1．expires

语法：expires [modified] time;

expires epoch | max | off;

默认值：expires off;

环境：http、server、location、if in location

用途：设置 Expires 和 Cache-Control 响应头字段，主要作用是控制缓存时间，如在浏览器上的缓存时间、CDN 的缓存时间。参数值可以是正数、负数或零，示例如下：

```
expires -1;    # 输出的响应头是 cache-control: no-cache，表示不缓存
```

如果要求在浏览器第一次访问后，数据在浏览器上缓存 1h，则配置如下：

```
expires 1h;    # 输出的响应头是 cache-control: max-age=3600
               # 表示缓存 1h，max-age 的单位是秒
```

此配置只能在 HTTP 状态码是 200、201、204、206、301、302、303、304、307 或 308 时才会生效。即只有当请求处于正常的返回状态时，才会发送缓存头，毕竟在浏览器上缓存一个错误状态不是什么好事情。

在 Nginx 1.7.9 版本之后，expires 指令可以使用变量来配置响应头，并根据响应头的 Content-Type 来定义缓存时间，也就是可以根据不同的条件动态地调整缓存时间。例如，如果 Content-Type 是 application/pdf，则添加 cache-control: max-age=3600 响应头信息；如果 Content-Type 是 image/，则添加 ache-control: max-age=36000 响应头信息；如果没有匹配到对应的 Content-Type，则执行 default 的配置 off，即不进行缓存，代码如下：

```
map $sent_http_content_type $expires {
    default         off;
    application/pdf 1h;
    ~image/         10h;
}
expires $expires;
```

在上述配置中，map 指令会根据响应头 Content_Type 的值对$expires 进行赋值（通过$sent_http_content_type 获取 Content-Type 的值）。

2．add_header

语法：add_header name value [always];

默认值：无

环境：http、server、location、if in location

用途：添加自定义的响应头。例如，可以用来添加 Cache-Control 响应头，以达到 expires 指令的效果，不过它不如 expires 那样简洁、明了，示例如下：

```
add_header Cache-Control no-cache;  # 等同于 expires -1;的效果
```

在默认情况下，add_header 只能在 HTTP 状态码是 200、201、204、206、301、302、

303、304、307 或 308 时输出响应头，如果出现 404、500 等异常状态码则无法输出响应头。但 Nginx 1.7.5 以上的版本新增了 always 参数，使之可以在任何 HTTP 状态下输出响应头，示例如下：

```
add_header Cache-Control no-cache always;  # 等同于 expires -1;的效果
                                           # 即使 HTTP 的状态码是 500
```

响应头信息输出如下：

```
 [root@testnginx ~]# curl -I http://testnginx.com/
HTTP/1.1 500 Internal Server Error
Server: nginx/1.12.2
Date: Tue, 30 Jan 2018 08:17:39 GMT
Content-Type: application/octet-stream
Content-Length: 1
Connection: keep-alive
Cache-Control: no-cache
```

测试结果表明，即使发生了 500 错误，加入 always 参数后依然可以输出 add_header 响应头信息。

3．实战经验

添加请求头有很多需要注意的地方，下面是在长期使用中总结的一些经验，供读者参考。

- 关于指令 expires 1h。

expires 1h 表示在浏览器上的缓存时间是 1h，但如果缺少 Last-Modified 响应头，大部分浏览器不会使用缓存。在业务代码生成过程中，部分代码可能没有使用规范的框架输出数据，因此可能会缺少 Last-Modified 信息，无法在浏览器上进行缓存，但请求仍会回源到服务器上，导致带宽压力增大。因此负责网络带宽和用户体验的技术人员要对使用规范进行说明和管理，避免出现缓存效果不理想的情况。

- 关于指令 add_header。

如果使用 always 参数，那么即使在业务出现异常时，也会输出 add_header 响应头信息，因此要慎重使用此参数。另外，如果后端服务器在返回响应体时返回了一个和 add_header 相同的响应头，则会导致两个响应头重复输出到浏览器上，从而引起不必要的 Bug，示例如下：

```
[root@testnginx ~]# curl -I http://testnginx.com/
HTTP/1.1 200 OK
Server: nginx/1.12.2
Date: Tue, 30 Jan 2018 08:50:40 GMT
Content-Type: application/octet-stream
Content-Length: 1
```

```
Connection: keep-alive
Cache-Control: 1000
Cache-Control: no-cache
```

4.1.2　使用 headers-more-nginx 控制请求头和响应头

前面讲到的 add_header 指令，只适合用来添加响应头，如需对 HTTP 请求头进行处理，可以使用第三方模块 headers-more-nginx，它可以用来添加、删除、修改 HTTP 请求头和响应头。该模块地址为 https://github.com/openresty/headers-more-nginx-module.git，其安装方式如下：

```
# wget 'http://nginx.org/download/nginx-1.12.2.tar.gz'
# git clone https://github.com/openresty/set-misc-nginx-module.git
# tar -xzvf nginx-1.12.2.tar.gz
# cd nginx-1.12.2/  &&  ./configure --prefix=/opt/nginx \
    --add-module=/path/to/headers-more-nginx-module
# make && make install
```

安装完成后，对其进行配置，让它实现和 add_header 指令一样的功能。

示例 1：

```
more_set_headers 'Cache-Control: 1000';
```

上述配置等同于“add_header Cache-Control 1000 always;”，作用是无论返回的状态码是什么都会输出响应头。

示例 2：

```
more_set_headers  -s '200 301' 'Cache-Control: 1000';
```

上述配置等同于 add_header Cache-Control 1000，表示当状态码是 200 或 301 时才输出响应头，从而加强对响应头的控制。

当后端服务器返回的响应头和使用 more_set_headers 指令时相同时，响应头的值会被 more_set_headers 替换掉，这样就不会同时出现两个相同的响应头了。

headers-more-nginx 模块常用的操作指令还有很多，例如 more_set_headers、more_clear_headers、more_set_ input_headers、more_clear_input_headers 等。

1．根据 HTTP 状态控制响应头

指令：more_set_headers

语法：more_set_headers [-t <content-type list>]... [-s <status-code list>]... <new-header>...

默认值：无

环境：http、server、location、location if

执行阶段：output-header-filter

示例：more_set_headers -s 404 -s '500 502' 'Result: error' 'F: X-re';

含义：在返回响应报文前对响应头进行新增或替换操作。示例的意思是当响应状态码（-s 参数的作用就是匹配对应的状态码）是 404、500 或 502 时，如果添加'Result: error'和'F: X-re'两个响应头，当后端服务器返回其中任何一个响应头时，more_set_headers 都会替换返回的值。

2．根据 HTTP 状态清除响应头

指令：more_clear_headers

语法：more_clear_headers [-t <content-type list>]... [-s <status-code list>]... <new-header>...

默认值：无

环境：http、server、location、location if

执行阶段：output-header-filter

示例：more_clear_headers -s 200 -t 'text/plain' F Result;

含义：在返回响应报文前清除指定的响应头。示例中的意思是当响应状态码是 200 时清除'text/plain'、F、Result 这 3 个响应头；如果不用-s 参数（即不对状态码进行匹配），指定的响应头在任何状态码下都会被清除。该指令还可以使用通配符，例如，想要清除以 X-开头的响应头，只需使用 more_clear_headers -s 200 -t 'X-*'即可。

3．设置 HTTP 请求头

指令：more_set_input_headers

语法：more_set_input_headers 'Host: testnginx.com ';

环境：http、server、location、location if

执行阶段：rewrite tail

含义：当请求在 rewrite 阶段并执行到最后时，再添加一个请求头，该请求头将和请求一起到达下一个阶段，示例如下：

```
location = /testnginx {
    set $test 'testnginx';
    more_set_input_headers 'Host: $test';
```

```
}
```

4．清除 HTTP 请求头

指令：more_clear_input_headers

语法：more_clear_input_headers -t Cache-Control;

环境：http、server、location、location if

执行阶段：rewrite tail

含义：当请求在 rewrite 阶段并执行到最后时，如果匹配到的请求头是 Cache-Control 就清除掉。该指令支持通配符，如“more_clear_input_headers 'Test-*';”，指的就是以 Test 开头的请求头将全部被清除。

5．实战经验

- 关于指令 more_clear_headers

清除响应头操作可以用来隐藏服务内部的一些敏感信息，如 varnish 自带的响应头，或者 PHP 框架生成的响应头等。在实际业务中首先要确定需要传递给客户端的响应头信息，而不必传递给客户端的信息则可以使用此方法进行清除。

- 关于指令 more_set_input_headers 和 more_clear_input_headers

这两个指令都在 rewrite tail 阶段使用，如果在同一个执行阶段使用这两个指令，会按照指令的前后顺序执行。

- headers-more-nginx 模块的指令都是区分大小写的，所以要注意区分字母的大小写。

4.2　第三方模块 set-misc-nginx

第三方模块 set-misc-nginx 在 rewrite 阶段使用，它功能丰富，具有设置变量、URL 转义、生成随机数、防止 SQL 注入、解码与编码等多项功能。

该模块地址为 https://github.com/openresty/set-misc-nginx-module.git。安装时直接编译即可，如下所示：

```
# wget 'http://nginx.org/download/nginx-1.12.2.tar.gz'
# git clone https://github.com/simplresty/ngx_devel_kit.git
# git clone https://github.com/openresty/set-misc-nginx-module.git
# tar -xzvf nginx-1.12.2.tar.gz
```

```
# cd nginx-1.12.2/
# ./configure --prefix=/opt/nginx \
     --add-module=/path/to/ngx_devel_kit \
     --add-module=/path/to/set-misc-nginx-module
# make && make install
```

注意：很多第三方模块在编译时都需要加入 ngx_devel_kit 模块，因为 ngx_devel_kit 扩展了 Nginx 服务器的核心功能，很多第三方模块都是基于它实现的。

下面将对 set-misc-nginx 模块的主要功能进行一一介绍。

4.2.1 设置变量

指令：set_if_empty

语法：set_if_empty $dst <src>

环境：location、location if

执行阶段：rewrite

示例如下：

```
location = /a  {
     set $t $arg_test;        # 获取 URL 中的参数 test 的值
     set_if_empty $t 123;     # 如果没有获取到，则使用默认值 123
}
```

如果不使用此模块，而直接用 Nginx 进行配置会比较麻烦。下面的代码展开了 Nginx 原生的配置方式，它使用了 if，而 if 是“邪恶”的（详见第 18 章）：

```
location = /a  {
     set $t $arg_test;       # 获取 URL 中的参数 test 的值
     if ($t = '') {
        set  $t 123;         # 如果没有获取到，则使用默认值 123
     }
}
```

4.2.2 防止 SQL 注入

指令：set_quote_sql_str

语法：set_quote_sql_str $dst <src> 和 set_quote_sql_str $dst

环境：location、location if

执行阶段：rewrite

示例如下：

```
location /test {
    set $v1 "testnginx\n\r'\"\\"; # 包含有回车符等字符的一个变量
    set_quote_sql_str $v2 $v1;    # 通过 set_quote_sql_str 将$v1 赋值给$v2，通
                                  # 常用于 MySQL 数据库的字符转义，防止 SQL 注入
    echo $v1;                     # 使用 echo 输出内容 testnginx\n\r\'\"\\
}
```

如果所使用的数据库是 MySQL，添加此配置可以防止 SQL 注入；而如果所使用的数据库是 PostgreSQL，可使用 set_quote_pgsql_str 来设置变量，使用方式与 set_quote_sql_str 指令一样。

上面的示例进行了字符转义，下面来看看不进行字符转义的情况，示例如下：

```
location /test {
    set $v1 "testnginx\n\r'\"\\";
    echo $v1;
}
```

输出结果如下：

```
[root@testnginx ~]# curl http://testnginx.com/test
testnginx
'"\
```

4.2.3　字符串非转义和转义

指令：set_unescape_uri

语法：set_unescape_uri $dst <src> 和 set_unescape_uri $dst

环境：location、location if

执行阶段：rewrite

示例如下：

```
location /test {
                  # 对字符串进行非转义，将非转义后的数据赋值给变量$key
    set_unescape_uri $key 'hello+world%21';
    echo $key;    # 使用 echo 输出内容 hello world!
}
```

如果想要对 URL 中的参数进行非转义，也可以用同样的方式操作，即 set_unescape_uri $key $arg_name。

指令：set_escape_uri

与 set_unescape_uri 的作用正好相反，set_escape_uri 是用来对数据进行转义的，使用方式和 set_unescape_uri 一样。

4.2.4 基于键值的集群分片

Nginx 通过第三方模块可以和数据库进行交互，这使 API（Application Programming Interface，应用程序编程接口）服务更容易部署和开发。当需要对数据库中的数据分片读取时，可以使用指令 set_hashed_upstream 进行配置。

指令：set_hashed_upstream

语法：set_hashed_upstream $dst <upstream_list_name> <src>

环境：location、location if

执行阶段：rewrite

示例如下：

```
upstream memcache1 { ... }
upstream memcache2 { ... }
upstream memcache3 { ... }

upstream_list universe memcache1 memcache2 memcache3;

location /test_ngx {
    set_unescape_uri $key $arg_key;
    set $list_name universe;
    set_hashed_upstream $backend $list_name $key;
    echo $backend;
}
```

上述配置中 upstream_list 声明并存放了 3 个后端服务，当 HTTP 请求访问/test_ngx 时，会根据 URL 的参数 key 的值，进行 hash 分片，然后在 upstream_list 中选出一个 upstream 后端服务。使用此方式可以实现分片请求，例如，制造一个 Memcahed 分片集群。

4.2.5 base 编码

有时可能需要对一些信息进行加密或简化其传输的数据内容，这时就要用到与 base 编码有关的功能了。

指令：set_encode_base32

语法：set_encode_base32 $dst <src> / set_encode_base32 $dst

环境：location、location if

执行阶段：rewrite

示例 1：

```
location /test {
    set $a "nginx";
    # 对变量$a 进行 base32 编码，并赋值给$test，$a 不会改变
    set_encode_base32 $test $a;
    echo $test;
}
```

输出结果如下：

```
[root@testnginx ~]# curl http://testnginx.com/test
dpjmirjo
```

示例 2：

```
location /test {
    set_encode_base32 $test $request_uri;  # 对 request_uri 进行编码
    echo $test;
}
```

输出结果如下：

```
[root@testnginx ~]# curl http://testnginx.com/test?sdaddsads12312
5tq6asrk7tpm8ob4chpm2p3j64p36c9i
```

指令：set_decode_base32

语法：set_decode_base32 $dst <src> / set_decode_base32 $dst

环境：location、location if

执行阶段：rewrite

该指令与 set_encode_base32 指令的用法相似，只是结果正好相反，是对<src>进行 base32 解码。另外，还有 set_encode_base64（base64 编码）、set_decode_base64（base64 解码）的使用方式与 base32 一样，只是编码或解码的位数不一样。

4.2.6 md5 编码

指令：set_md5

语法：set_md5 $dst <src> | set_md5 $dst

默认值：无

环境：location、location if

执行阶段：rewrite

含义：对<src>进行 md5 编码，然后赋值给$dst。

示例如下：

```
location /test {
    # 将$request_uri 的值赋值给$test，并进行 md5 编码
    set_md5  $test $request_uri;
    echo $test;
}
```

输出结果如下：

```
[root@testnginx ~]#  curl http://testnginx.com/test?sdaddsads12312
9a9d7101420a744b68debe3d7bb91eb3
```

4.2.7 生成随机数

指令：set_random

语法：set_random $res <from> <to>

默认值：无

环境：location、location if

执行阶段：rewrite

含义：在 from 和 to 之间生成一个随机正整数

示例如下：

```
location /test {
    set $from 1;
    set $to 100;
    # 随机生成一个 1～100 之间的正整数，并赋值给$result
    set_random $result $from $to;
```

```
        echo $result;
    }
```

输出结果如下：

```
[root@testnginx ~]#  curl http://testnginx.com/test
96
```

指令：set_secure_random_alphanum

语法：set_secure_random_alphanum $res <length>

默认值：无

环境：location、location if

执行阶段：rewrite

含义：生成一个<length> 长度的随机变量，这个变量可以包含 a~z 之间、A~Z 之间、0~9 之间的字符。

示例如下：

```
location /test {
    set_secure_random_alphanum $result 16;
    echo $result;
}
```

可以看出每次的输出结果都是不一样的：

```
[root@testnginx ~]# curl http://testnginx.com/test
e3nwGeMF39FebMVY
[root@testnginx ~]# curl http://testnginx.com/test
aF1c89KLugUXaU9T
[root@testnginx ~]# curl http://testnginx.com/test
Yd8dkzXjE5DdBUx7
[root@testnginx ~]# curl http://testnginx.com/test
CSa2m3fyoUDywRAv
```

指令：set_secure_random_lcalpha

语法：set_secure_random_lcalpha $res <length>

默认值：无

环境：location、location if

执行阶段：rewrite

含义：和 set_secure_random_alphanum 的作用相似，会生成一个<length> 长度的随机变

量，区别是该变量只能包含 a~z 之间的字符。

4.2.8 本地时间的输出

指令：set_local_today

语法：set_local_today $dst

默认值：无

环境：location、location if

执行阶段：rewrite

含义：获取系统时间，并设置格式为("yyyy-mm-dd")，然后将其赋值给$ds。

示例如下：

```
location /test {
    set_local_today $a;# 获取系统时间并格式化为("yyyy-mm-dd")之后，将其赋值给$a
    echo $a;
}
```

输出结果如下：

```
[root@testnginx ~]# curl http://testnginx.com/test
2018-03-13
```

更多关于通过 Nginx 定制时间格式的知识会在 Ngx_Lua 相关章节中进行讲解。

4.2.9 实战经验

- 关于指令 set_if_empty

要尽量减少 if 语句的使用，而且使用 set_if_empty 能够提升 Nginx 的可读性。

- 关于指令 set_quote_sql_str

一旦让 Nginx 和数据库直接交互，要确保 set_quote_sql_str 指令的存在，以防止 SQL 注入。

- 关于指令 set_unescape_uri 和 set_escape_uri

在处理数据时，如 URL，需要确保转义和非转义的统一性，避免出现前面的代码做了转义，但后面的某些环节却使用非转义数据的情况，因为这样会导致在匹配和对比时出现数据不一致的情况。

- 关于指令 set_hashed_upstream

该类型的分片比较简单，不涉及高可用。如果某个 upstream 服务全部“挂掉”，则会报

错。因此，在设计分片时要考虑足够多的可能性。

4.3　图片的处理

Nginx 可以使用 ngx_http_image_filter_module 对图片进行切割裁剪，这为动态处理图片提供了支持。

- 安装方式：只需在编译 Nginx 时，添加--with-http_image_filter_module 即可。
- 依赖库：libgd（gd-devel），在 CentOS 系统下通过 yum 安装即可（推荐使用最新版本）。

4.3.1　image_filter 图片处理

指令：image_filter

语法如下：

```
image_filter off;
image_filter test;
image_filter size;
image_filter rotate 90 | 180 | 270;
image_filter resize width height;
image_filter crop width height;
```

默认值：image_filter off;

环境：location

含义如下。

- image_filter off;

设置是否关闭此区域图片的处理功能，默认是关闭的。

- image_filter test;

确保图片类型是 JPEG、GIF、PNG 或 WebP，否则会返回 415 错误。

- image_filter size;

以 JSON 格式输出图像的信息，格式如下：

```
{ "img" : { "width": 750, "height": 286, "type": "jpeg" } }
```

- image_filter rotate 90 | 180 | 270;

规定逆时针旋转的度数，参数值可以是变量。该模式可以单独使用，也可以与 resize 和

crop 一起使用。

- image_filter resize width height;

按 width（宽）、height（高）的比例将图像缩放到指定尺寸。如果只减少其中一个维度，可以将另一个维度指定为-。参数值可以是变量。当与旋转参数一起使用时，旋转将在缩小后发生。如果发生错误，服务器会返回 415。

- image_filter crop width height;

按 width（宽）、height（高）的比例减少图像较大的侧面积和另一侧多余的边缘，如果只减少其中一个维度，可以将另一个维度指定为-。参数值可以是变量。当与旋转参数一起使用时，旋转将在缩小之前发生。如果发生错误，服务器会返回 415。

示例 1：

将一张 JPG 格式的图片命名为 test.jpg，并存放到/usr/local/nginx_1.12.2/conf 目录下，进行测试：

```
location /test.jpg {
    image_filter size;
    root /usr/local/nginx_1.12.2/conf;
}
```

结果将会以 JSON 格式输出图片的信息，如下所示：

```
{ "img" : { "width": 750, "height": 286, "type": "jpeg" } }
```

示例 2：

```
location /test.jpg {
    image_filter resize 150 100;  # 指定图片大小
    image_filter rotate 180;      # 将图片逆时针旋转 180°
    root /usr/local/nginx_1.12.2/conf;
}
```

原图如图 4-1 所示。

图 4-1　原图

处理后的图片如图 4-2 所示。

图 4-2　处理后的图片

对图片进行裁剪，示例代码如下：

```
location /test.jpg {
    image_filter crop 150 100;     # 裁剪为 150×100 的图片
    image_filter rotate 180;       # 将图片逆时针旋转 180°
    root /usr/local/nginx_1.12.2/conf;
}
```

裁剪后的图片如图 4-3 所示。

图 4-3　裁剪后的图片

4.3.2　采用渐进式方式打开 JPEG 图片

图片被传输到客户端浏览器上，以往是以从上到下的扫描方式呈现的，即用户在浏览器上看到的图片是一行一行或一块一块打开的，类似于瀑布流的呈现方式。Nginx 提供了渐进式的功能，支持图片通过渐进式方式打开。渐进式图片打开时会先出现一个模糊的全图，再逐步变清晰。此功能需要用到的指令是 image_filter_interlace。

指令：image_filter_interlace

语法：image_filter_interlace on | off;

默认值：image_filter_interlace off;

环境：http、server、location

含义：如果设置为 on，则表示启用此功能，采用渐进式方式打开 JPEG 格式的图片。

注意：

- 该指令是在 Nginx 1.3.15 版本之后加入的，建议在 1.4 以上的版本中使用。
- 渐进式的呈现方式被大多数人认为比瀑布流的呈现方式更好看，但它也会消耗更多的 CPU 和内存资源。
- 大图片更适合使用渐进式呈现方式打开，只有几 KB 的小图片就没必要了。

4.3.3 WebP 格式

随着 WebP 格式的诞生，图片在质量和大小之间有了更好的平衡，WebP 格式可以在保持同样清晰度的情况下，将图片的大小减少 30%以上。

Nginx 在 1.11.6 版本之后加入了对 WebP 格式的支持，如果要把图片转换为 WebP 格式，需要安装 WebP 的依赖库。

4.3.4 优化图片

指令：image_filter_buffer

语法：image_filter_buffer size

默认值：image_filter_buffer 1M

环境：http、server、location

含义：设置图像读取缓冲区的最大值。当超过此值时，服务器会返回 415 错误。

注意：在图片的应用服务中，原图一般是从后台或某些接口上传的，因此需要让上传图片的大小小于或等于 image_filter_buffer 设置的值。避免出现由于 image_filter_buffer 的限制无法处理图片的情况。

如果读取到的图片大小超过了所设置的缓冲区最大值，在 error.log 中会出现如下错误：

```
   [error] 13313#0: *272 image filter: too big response: 161036 while
sending response to client,
```

指令：image_filter_jpeg_quality

语法：image_filter_jpeg_quality quality；

默认值：无

环境：http、server、location

含义：设置图片转换为 JPEG 格式后的质量。参数值的范围在 1~100 之间，参数值越小意味着图片质量越差，同时，数据传输量也越小。参数值可以是变量，但推荐最大设为 95。

注意：参数值越大，图片的字节数就越大，这意味着网络传输的时间也就越长，会影响用户体验。根据以往的经验，参数值在 70~85 之间时，图片的清晰度可以满足大部分需求。读者可以根据产品自身的属性设置参数值。

指令：image_filter_sharpen

语法：image_filter_sharpen percent；

默认值：image_filter_sharpen 0;

环境：http、server、location

含义：设置图片的清晰度，参数值（锐度百分比）可以超过 100；默认为 0，表示禁用锐化；参数值可以是变量。

指令：image_filter_transparency

语法：image_filter_transparency on | off;

默认值：image_filter_transparency on;

环境：http、server、location

含义：设置当把图片转换为 GIF 格式或 PNG 格式时是否保留透明度。PNG 格式中的 Alpha 通道透明度始终保持不变，因此在处理 PNG 格式的图片时，透明度不会随着参数值的变化而变化。

指令：image_filter_webp_quality

语法：image_filter_webp_quality quality;

默认值：image_filter_webp_quality 80;

环境：http、server、location

含义：设置图片转换为 WebP 格式后的质量。可接受的参数值在 1~100 之间，参数值越小意味着图片质量越差，同时，数据传输量也越小。

注意：该指令在 Nginx 1.11.6 版本之后加入。WebP 格式图片的大小明显小于 JPEG 格式图片的大小，如果继续降低 WebP 格式图片的质量，可能会影响图片清晰度，所以请谨慎使用指令 image_filter_webp_quality。

4.3.5 实战经验：动态切图

ngx_http_image_filter_module 中的大多数指令都支持使用变量，这为动态处理图片提供了帮助。那么如何根据图片格式和大小等参数动态地提供图片呢？下面是一个简单的配置示例：

```
location ~* ^/(.+)\.(\d+|-)x(\d+|-)\.(gif|png|webp|jpg)$ {
    set $w $2;   # 获取匹配到的 URL 中的字符，把捕获的值设为宽
    set $h $3;   # 获取匹配到的 URL 中的字符，把捕获的值设为高
    image_filter resize $w $h;    # 动态裁剪操作
    image_filter_buffer 3M;       # 读取原始图片的最大值
    try_files /$1.$4  /404.jpg;   # 在硬盘中读取文件时，如果读取不到，
                                  # 则读取/404.jpg
    root /usr/local/nginx_1.12.2/conf;
}

location = /404.jpg {
    return 404;
}
```

注意：

- try_files 的作用：如果不用 try_files，而直接使用 root 的对应路径查找文件，当源文件不存在时，就会抛出 415 错误，这会让人误以为在图片切割时出现了异常。而如果使用 try_files，当源文件读取不到时，会直接请求/404.jpg 文件，显示 404 页面，提醒维护者源文件不存在了。
- image_filter resize $w $h：如果将 0 作为变量，会发生 415 错误，可以将 0 换为中横线-。

4.4 TCP 和 UDP 代理

在 Nginx 1.9.0 版本之前，TCP 代理一般会使用 Haproxy 或 Nginx 的第三方包 nginx_tcp_proxy_module。但 Nginx 在 1.9.0 版本之后加入了 ngx_stream_proxy_modulea，它提供了 TCP 代理，只需在编译 Nginx 时加入--ngx_stream_proxy_module 即可。

4.4.1 代理配置说明

先看一个示例：

```
worker_processes 2;

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

events {
    worker_connections  1024;
}

stream {
    upstream backend {
        hash $remote_addr consistent;
        server x.testnginx.com:11233 weight=5 max_fails=3 fail_timeout=30s;
        server 127.0.0.1:11233       weight=2 max_fails=3 fail_timeout=30s;
        server 192.168.100.10:11233  weight=3 max_fails=3 fail_timeout=30s;
        server unix:/tmp/backend3    weight=1 max_fails=3 fail_timeout=30s;
}

    server {
        listen 11233;
        proxy_connect_timeout 1s;
        proxy_timeout 3s;
        proxy_pass backend;
    }

}
```

上述配置充分体现了 TCP 代理的特点。

- TCP 代理在 stream 指令块内进行声明，位于 main 内，和 http 指令块同级。
- 反向代理的 upstream 支持 DNS 域名（如 ip_hash、socket）配置、权重即故障转移（如 max_fails）配置。
- 当 proxy_pass 代理 TCP 时，没有 http://前缀，注意配置时不要写错。
- 支持和 HTTP 一样的连接超时参数 proxy_timeout、proxy_connect_timeout。

TCP 代理还包含其他很多指令，下面将介绍一些常用的指令。

指令：proxy_bind

语法：proxy_bind address [transparent] | off;

默认值：无

环境：stream、server

含义：如果此配置为 off，则表示请求将会使用系统自动分配的本地 IP 地址，即后端服务

看不到用户的真实 IP 地址；如果配置为“proxy_bind $remote_addr transparent;”，则后端服务可以看到用户的真实 IP 地址。为了使参数生效，需要以超级用户权限运行 Nginx 的 worker 进程，并配置核心路由表以截获反向代理服务器的网络流量。

指令：proxy_download_rate

语法：proxy_download_rate rate;

默认值：proxy_download_rate 0;

环境：stream、server

含义：设置后端服务器读取数据的速度。速度定义为字节每秒。默认值为 0，表示禁用限速。限速设置的值对每个连接都有效。

指令：proxy_next_upstream

语法：proxy_next_upstream on | off;

默认值：proxy_next_upstream on;

环境：stream、server

含义：当无法与当前的后端服务器建立连接时，该指令用来确定是否将客户端连接传递给下一台后端服务器。这可能会受到尝试次数（proxy_next_upstream_tries）和时间（proxy_next_upstream_timeout）的影响。

指令：proxy_next_upstream_timeout

语法：proxy_next_upstream_timeout time;

默认值：proxy_next_upstream_timeout 0;

环境：stream、server

含义：设置传递连接到下一台后端服务器的时间。默认值为 0，表示关闭这个限制。

指令：proxy_next_upstream_tries

语法：proxy_next_upstream_tries number;

默认值：proxy_next_upstream_tries 0;

环境：stream、server

含义：设置传递连接到下一台后端服务器的尝试次数。默认值为 0，表示关闭这个限制。

指令：proxy_pass

语法：proxy_pass address;

默认值：无

环境：server

含义：设置被代理的服务器地址。地址可以是一个域名，也可以是 IP 地址加端口号。从 Nginx 1.11.3 版本开始支持配置变量，如“proxy_pass $upstream;”。

指令：resolver

语法：resolver address ... [valid=time] [ipv6=on|off];

环境：stream、server

含义：在进行 DNS 解析时，该指令用于对 upstream 中出现的域名进行 IP 地址解析，然后将请求代理到解析到的 IP 地址上；Valid=time 表示 DNS 解析后的缓存时间，使用缓存时间可以减少 DNS 解析的次数；关于 ipv6=on|off，如果设置为 on，则表示在查询 IPv4（Internet Protocol version 4，互联网协议版本 4）无果后会继续查询 IPv6（Internet Protocol Version 6，互联网协议版本 6），如果不需要查找 IPv6，可以配置为 off，以缩短查找时间。

指令：resolver_timeout

语法：resolver_timeout time;

默认值：resolver_timeout 30s;

执行阶段：stream、server

含义：设置解析 DNS 的超时时间，如果超时，DNS 会使用上一次解析到的 IP 地址。

指令：proxy_protocol_timeout

语法：proxy_protocol_timeout timeout;

默认值：proxy_protocol_timeout 30s;

环境：stream、server

含义：设置读取 proxy 协议头的时间。如果在所设置的时间内没有发送完整的报头，则关闭连接。

随着 Nginx 版本的不断更新，新的参数也在不断加入，读者可以随时关注官网的 Wiki。

4.4.2 DNS 服务的反向代理

在 1.9.13 版本之后，Nginx 加入了对 UDP（User Datagram Protocol）代理的支持，因此可以使用 Nginx 来代理 DNS 的 UDP 端口，配置如下：

```
stream {
    resolver_timeout 8.8.8.8;

     upstream dns {
       server 192.168.1.3:5343;
       server dns.testnginx.com:5343;
    }

    server {
        listen 127.0.0.1:53 udp;
        proxy_responses 1;
        proxy_timeout 20s;
        proxy_pass dns;
    }

}
```

4.4.3 MySQL 集群代理配置

在 MySQL 数据库一主多从的架构中，可以使用 Nginx 的 TCP 代理来管理和维护从库的数据，配置如下：

```
stream {
    upstream mysql_servers {
        least_conn;  # 请求会优先发给连接少的服务器
        server 192.168.0.34:3306 max_fails=2 fail_timeout=30s;
        server 192.168.0.35:3306 max_fails=2 fail_timeout=30s;
    }

server {
    listen 3306;
    proxy_connect_timeout 1s;
    proxy_timeout 3s;
    proxy_pass mysql_servers;    }
}
```

上述配置看上去很“清爽”，如果某台 MySQL 服务器变慢或宕机，Nginx 会自动将它“踢”出去。

4.4.4　实战经验

1．指令 nginx_tcp_proxy_module 和 ngx_stream_core_module 都可以在 TCP 代理中使用。其中 ngx_stream_core_module 包含有关日志记录的一些常用变量，通过这些日志信息可以分析每个 TCP 请求的情况。

2．在做 TCP 代理时，Nginx 代理节点服务器可能不止一台，可以使用 DNS 轮询或 LVS（Linux Virtual Server，Linux 虚拟服务器）等功能对 Nginx 进行负载均衡。

3．Redis 属于单核服务，所以在一台服务器上启动多个 Redis 实例是很正常的。如果多个 Redis 从库部署在同一台服务器上，可以通过 TCP 代理进行转发，否则会因为端口不一致，导致配置非常烦琐。

4.5　常用模块介绍

下面介绍一些常用的 Nginx 模块，它们为反向代理提供了更多的功能，并且可以降低后端代码的复杂度。

4.5.1　基于访问 IP 地址跳转到对应城市

在有些情况下，网站页面的展示和用户所在的城市有关，如团购、优惠服务等。那么，App 可以通过 GPS 定位用户所在的城市，PC 端则可以通过访问 IP 地址来判断对应的城市。

当 PC 端发送请求时，会根据客户端的 IP 地址判断其属于哪个城市，并将请求通过 302 跳转到对应城市的网站页面上，根据访问 IP 地址跳转到对应城市页面的业务逻辑如图 4-4 所示。

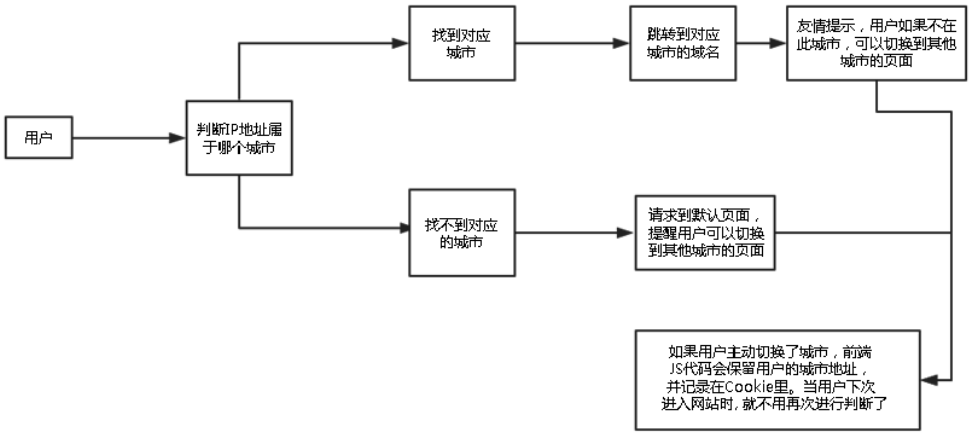


图 4-4　根据访问 IP 地址跳转到对应城市页面的业务逻辑

图 4-4 中的关键点如下。

- 需要一份 IP 地址所属城市的清单。网上有很多开源的 IP 地址对应城市库，读者可自行下载。如果对 IP 地址信息的准确度要求较高，建议使用付费平台。
- IP 地址对应城市库可能会存在误差，也可能用户所在的城市没有相关网站的服务，以致无法完成跳转，因此需要做一些友好提示，方便用户切换到想去的城市页面。
- 如果用户执行了切换城市的操作，可以认为用户会在这个城市继续访问，所以不必在每次请求时都对 IP 地址进行判断，只需通过前端的 JS 代码设置 Cookie 即可。

了解到核心的关键点之后，就可以通过 Nginx 进行处理了。

模块：ngx_http_geo_module（Nginx 默认支持此模块，不需要额外添加）。

指令：geo

语法：geo [$address] $variable { ... }

环境：http

示例如下：

```
geo $geo {
    ranges;    # 以 IP 地址段的形式定义地址，IP 地址段必须放在首位，且为了提升装载 IP
               # 地址库的性能，应按升序排列
    default  beijing;    # 若无法匹配到对应的 IP 地址段，则使用默认值
    1.0.32.0-1.0.63.255 guangzhou;   # IP 地址段对应的城市，用空格分开
    1.2.2.0-1.2.2.255 beijing;
    1.2.4.0-1.2.4.255 beijing;
    1.24.0.0-1.24.31.255 huhehaote;
    1.24.32.0-1.24.39.255 wulanchabu;
}
```

含义：当请求经过 http 执行阶段时，geo 指令默认会从 $remote_addr 处获取 IP 地址的值，然后去和$address 进行匹配，如果匹配成功，则获取$address 对应的值，并将此值赋给$geo；如果匹配失败，则使用 default 的值，即 beijing。

下面提供一个简单的 Nginx 配置，它可以实现通过 IP 地址跳转到对应城市页面的功能：

```
location = / {
    if ($cookie_current_city){
       set $geo $cookie_current_city;
       # $cookie_current_city 表示用户可能已经选择过城市，或者已经根据 IP 地址跳转
       # 到了对应的城市页面，所以直接跳转到 Cookie 中对应的城市页面即可
       rewrite ^/ $geo.testnginx.com redirect;
    }
```

```
        rewrite ^/ $geo.testnginx.com redirect;
    }
```

在上述配置中，在用户进入网站首页后，浏览器会根据用户的 IP 地址跳转到对应的城市页面，如果用户发现跳转有误差或想换个城市页面看看，那么可以切换到自己想看的城市页面，此时，前端 JS 代码可以通过设置 Cookie，确保下次用户访问首页时会默认进入切换后的城市页面。

注意：

- Cookie 应该设置有效期。
- 要确保每个城市的服务都是存在的，如果某些城市没有对应的服务，那么应该删除对应城市页面的跳转配置，避免跳转到空页面。

4.5.2　修改响应内容

你是否遇到过这样的场景：在应用服务发布新代码后，发现某个字符或内容有 Bug，而如果修改代码的话，需要重新合并、验证死链、检查服务，还要避免影响到其他功能，甚至还可能需要回归一次服务。那么在时间有限的情况下，该如何处理呢？

Nginx 的过滤模块 ngx_http_sub_module 可以解决这个问题。它是 Nginx 的内置模块，可以用来修改响应内容的数据。该模块默认未被激活，如果要激活的话，只需在编译 Nginx 时添加--with-http_sub_module 即可。

指令：sub_filter

语法：sub_filter string replacement;

环境：http、server、location

含义：将在响应内容中匹配到的 string（忽略大小写）替换成 replacement。

示例如下：

```
    location / {
        sub_filter 'href="http:// '  'href="//';
        sub_filter   '<img   src="http://img.testnginx.com'   '<img   src="//i1.
testnginx.com';
    proxy_pass http://servers;
    }
```

在上面的示例中，通过指令 sub_filter 修改了响应内容的两个位置。其中一个是将网络协议从 HTTP 切换为 HTTPS，即将 'http:// ' 替换成'//'。这样，页面中包含的链接就可以根据当前

访问的域名协议来决定请求的协议了，可以方便协议的切换。

此模块还包含一些其他的指令。

指令：sub_filter_last_modified

语法：sub_filter_last_modified on | off;

默认值：sub_filter_last_modified off;

环境：http、server、location

含义：在替换内容期间，保留来自原始响应的 Last-Modified 响应头内容，以便将响应数据进行缓存。如果将其设置为 off，则表示不保留。

指令：sub_filter_once

语法：sub_filter_once on | off;

默认值：sub_filter_once on;

环境：http、server、location

含义：如果将其设置为 on，表示替换所有匹配到的响应内容；如果将其设置为 off，则只会替换第一次匹配到的响应内容。

指令：sub_filter_types

语法：sub_filter_types mime-type ...;

默认值：sub_filter_types text/html;

环境：http、server、location

含义：替换操作默认对 MIME（Multipurpose Internet Mail Extensions，多功能因特网邮件扩展）类型为 text/html 的内容生效，如需替换其他类型的内容，可以使用*来表示替换所有类型的内容。

4.5.3 零像素文件的生成及其作用

推荐系统近几年非常火爆，后端服务通过大数据分析给用户提供需要的信息，那么大数据的数据从何而来呢？

比较常用的方式是，将用户的访问行为记录在一个 URL 参数里，通过 JS 发送到服务器上，然后使用大数据技术分析这些 URL 中的参数。这里 URL 的作用是记录用户的访问轨迹，不需要有响应内容返回，因此 URL 可以使用一个零像素的图片。而 Nginx 的模块

ngx_http_empty_ gif_module 正好提供了此功能，它默认在 Nginx 编译时安装。

指令：empty_gif;

默认值：无

环境：location

示例如下：

```
location ~* fx_\.gif$ {
    add_header Cache-Control "no-cache, max-age=0, must-revalidate";
    empty_gif;
}
```

含义：任何以 fx_.gif 结尾的文件都会生成一个空白的零像素图片，并且通过设置 Cache-Control 确保请求必须回源，不能被缓存。在 location 中使用了正则表达式，主要是为了方便研发团队。如果前端需要一个新的零像素图片，直接访问 new1_fx_.gif 文件就可以了，不需要通过运维或上线服务来生成 URL。

注意：

- 零像素的请求资源虽然非常小，但如果发送太频繁也会消耗一定的客户端 CPU 和带宽资源，因此需要结合实际情况，合理设置发送频率。
- 提供正则表达式意味着可以通过随意组合来合理安排 URL，避免因 URL 重复导致计算不合理的情况发生。另外，为了避免被攻击，需要做好安全保障。

4.5.4　图片的防盗链

为了阻止盗链的情况出现，可以使用 ngx_http_referer_module 模块。此模块是 Nginx 的内置模块，不需要重新编译。

防盗链示例如下：

```
location ~* \.(gif|jpg|png|webp)$ {
    valid_referers none blocked server_names
                *.testnginx.com   ~\.baidu\.
                ~\.google\.;

    if ($invalid_referer) {
       return 403;
    }
}
```

含义：示例中的 referer 是*.testnginx.com、baidu、google 的域名，这些域名都可以正常访

问此 location 下的内容。如果 referer 不是这些域名，$invalid_referer 的值就是 1，此时会返回 403 错误。

注意：大多数图片的域名可能被传递给第三方 CDN，如果图片已经被缓存到了该 CDN 上，此配置只有在请求没有命中 CDN 时回源才会生效，因此要想做好防盗链工作，需要和 CDN 厂家进行协商配置。

4.6 小结

通过对本章的学习，各位读者应该对 Nginx 的使用有了进一步的了解。随着对 Nginx 的深入使用，大家会发现很多功能是目前已知的模块不能实现的，这时就需要编写自定义模块了，后面的章节会介绍一些 Nginx 编写自定义模块的技巧。

第 5 章
缓存系统

几乎所有的大型互联网平台都会用到缓存系统，缓存系统在整个服务中扮演着极为重要的角色。它既可以提升请求的访问速度，又可以减少后端的请求压力，还能在后端服务出现异常时提供一份备份数据，避免出现白屏等风险。

目前，在开源项目中比较流行的 HTTP 缓存系统包括 varnish、squid、proxy_cache 等，这些缓存系统各有特色，本节将要介绍的是 Nginx 下的 proxy_cache 缓存系统。虽然它只是 Nginx 的一个小小的组件，但却拥有非常丰富的功能。

proxy_cache 是被包含在 ngx_http_proxy_module 模块中的，第 3 章中的 proxy 代理也在这个模块下。proxy_cache 在 Nginx 编译时默认安装。

5.1 缓存配置说明

先来了解一下 proxy_cache 缓存系统的基本配置，如下所示：

```
    upstream test_servers {
        server 127.0.0.1:81  max_fails=5 fail_timeout=10s weight=10;
        server 127.0.0.1:82  max_fails=5 fail_timeout=10s weight=10;
    }
    # 设置缓存空间的名字及其存放路径和存放方式
    proxy_cache_path  /data1/nginxcache  levels=1:2  keys_zone=cachedata:100m
inactive=7d max_size=50g use_temp_path=off;
    server {
        listen 80;
        location / {
```

```
            # 指定缓存的空间大小
            proxy_cache cachedata;
            # 指定缓存的 HTTP 状态及缓存时间
            proxy_cache_valid        200 304  10s;
            proxy_cache_valid        301 302 100s;

            # 请求最少被访问两次才会被缓存
            proxy_cache_min_uses 2;

            # 指定缓存的 key
            proxy_cache_key $scheme$host$uri$is_args$args;
            # 缓存的 HTTP 请求方法类型
            proxy_cache_methods GET HEAD ;
            # 添加一个响应头，用来标识请求是否命中缓存
            add_header N-Cache-Status $upstream_cache_status;

            # 设置为 on 表示允许将请求的 HEAD 方法改成 GET 方法进行缓存
            proxy_cache_convert_head on;

            sendfile on;

            proxy_set_header     Host $host:$server_port;
            proxy_set_header     X-Real-IP $remote_addr;
            proxy_set_header     X-Forwarded-For $proxy_add_x_forwarded_for;

            # 当缓存丢失时，会去后端服务器获取数据，然后，将之返回给用户并进行缓存
            proxy_pass http://test_servers;
        }
    }
```

下面对上述配置中与 proxy_cache 相关的指令进行说明，如表 5-1 所示。

表 5-1　proxy_cache 相关指令说明

指　令	用　途	支持的配置环境
proxy_cache	表示启动缓存，并且指定缓存信息存放的共享存储区域，指定的值就是 keys_zone 设置的值，支持使用变量，默认为 off	http、server、location
proxy_cache_valid	根据状态码来定义缓存的有效期 支持多行配置，支持不同状态码配置不同的有效期	http、server、location
proxy_cache_key	设置缓存的 key，默认 key 是完整的 URL，如$scheme$proxy_hosturiis_args$args	http、server、location

续表

指　令	用　途	支持的配置环境
proxy_cache_path	设置缓存文件路径，有如下参数 1．levels 代表缓存的目录结构，如 levels=1：2 表示目录有两级 2．keys_zone 用来设置缓存空间的名字和大小 如 keys_zone=cachedata:100m，表示缓存名是 cachedata，最多可以存放 100MB 的缓存信息 3．Inactive 用于指定缓存内容的有效期，如 Inactive=7d，表示有效期是 7 天 4．max_size 指缓存空间的最大值，当缓存量达到最大值时，会依次删除访问量最少的缓存内容 5．use_temp_path 默认为 on，表示缓存数据会首先写入临时文件，即 proxy_temp_path 指定的目录，写完后重命名该文件并保存到指定的缓存目录下。如果临时文件和缓存目录不在同一个硬盘上，此时会产生 I/O。如果 use_temp_path 为 off，则会直接将临时文件写入缓存目录，而不进行重命名和保存操作	http、server、location
proxy_cache_convert_head	如果设置为 on 表示允许将请求的 HEAD 方法改成 GET 方法进行缓存；如果设置为 off，则缓存的 key 需要加上$request_method，即 $host$request_uri$request_method	http、server、location
proxy_cache_methods	指定可以被缓存的请求方法，默认值为 GET HEAD	http、server、location
proxy_cache_min_uses	设置 key 在被缓存前被访问的次数，默认是 1 次	http、server、location

5.2　控制缓存有效期

proxy_cache_valid 是统一配置缓存有效期的指令，根据业务需求的不同，需要配置不同的缓存有效期，proxy_cache 支持动态配置缓存有效期及设置缓存优先级等。

Nginx 可以根据请求返回的响应头按照优先级控制缓存的有效期，具体如下。

1．X-Accel-Expires：优先级最高，缓存有效期为 60s，若设置为 0，则表示不缓存，如“X-Accel-Expires:60”。

2．Cache-Control 或 Expires：如果没有 X-Accel-Expires 响应头，则使用本响应头，缓存有效期为 60s，若设置为 0，则表示不缓存，如“Cache-Control:60”。

3．proxy_cache_valid：如果前两个响应头都没有，则使用本响应头，缓存有效期为 10s，如“proxy_cache_valid 10s;”。

需要注意的是，当响应头中有 Set-Cookie 时，不会被缓存；当响应头中有 Vary 且有特殊的值为*时，不会被缓存。那么，如何才能让 Set-Cookie、Vary 等也能被缓存呢？请看下面的

指令。

指令：proxy_ignore_headers

语法：proxy_ignore_headers field ...;

环境：http、server、location

含义：忽视来自后端的响应头信息，目前支持的可以被忽视的响应头有 X-Accel-Redirect 、 X-Accel-Expires 、 X-Accel-Limit-Rate 、 X-Accel-Buffering 、 X-Accel-Charset 、 Expires、Cache-Control、Set-Cookie 和 Vary。当这些响应头信息被忽视之后，即使存在 Set-Cookie 和 Vary 操作，也可以照样使用缓存。

示例如下：

```
proxy_ignore_headers    Vary    Set-Cookie ;
```

proxy_cache 还支持根据变量来判断是否使用缓存，不使用缓存的配置指令见表 5-2。

表 5-2　不使用缓存的配置指令

指　令	用　途	支持配置的环境
proxy_no_cache	设置缓存不保存的条件 如当请求有参数 a 或 b 且值不为空或 0 时，不进行缓存： proxy_no_cache $arg_a $arg_b	http、server、location
proxy_cache_bypass	设置不使用缓存而直接去后端获取数据的条件 如当 Cookie 中有 dd 且不为空或 0 时，就去后端获取数据。它支持配置多个条件：proxy_cache_bypass $cookie_dd	http、server、location

5.3　性能优化

Nginx 缓存系统可以通过微调提升缓存的性能。本节我们将会对硬盘 I/O、多磁盘节点进行调优，使缓存系统更加“健壮”，也会对如何使缓存“优雅”地失效做出介绍。

5.3.1　缓存未命中的最佳实践

缓存系统中的数据有热数据和冷数据之分，热数据请求会比较集中，而冷数据请求会比较零散。如果使用之前讲的知识来配置缓存系统，那么，当缓存失效时，会对后端服务造成压力，出现的异常情况可能有以下两种。

- 当热数据缓存失效时，会有大量并发的重复 URL 回源到后端服务器上，造成请求重

复，此时从监控上看后端服务，可能会出现一些小尖刺状的性能波峰。

- 冷数据大量的零散请求会导致缓存命中率低，并且请求会持续传递到后端服务器上，后端服务会一直处于忙碌状态。

这两种情况都有可能导致后端服务响应缓慢甚至崩溃，从而返回 5xx（500、502、503、504 等）错误。那么该如何应对呢？ngx_http_proxy_module 模块提供了多个指令来避免类似情况的发生。

先拿冷数据来说，如果之前请求访问过的缓存已经过期或根本没有缓存，可以使用与 proxy_cache_lock 有关的指令来减少并发到后端的请求数量。

指令：proxy_cache_lock

语法：proxy_cache_lock on | off;

默认值：proxy_cache_lock off;

环境：http、server、location

含义：如果设置为 on，当多条相同的 URL 请求 Nginx 缓存未命中时，Nginx 只会发送其中一条请求到后端服务器去获取响应数据，并将之存放到缓存系统中，剩下的请求则会调用存放在缓存系统中的数据。但如果等待时间超过 proxy_cache_lock_timeout 设置的时间，所有请求都会去后端服务器获取数据。

指令：proxy_cache_lock_timeout

语法：proxy_cache_lock_timeout time;

默认值：proxy_cache_lock_timeout 5s;

环境：http、server、location

含义：为 proxy_cache_lock 设置超时时间，如果超过这个时间，等待锁的请求就会去后端服务器获取数据，且数据不会被缓存。

上面两个指令也适用于热数据，但热数据的问题会更复杂一些，例如，当数据过期时同一时间相同 URL 的请求会非常多，此时缓存中应该有过期的数据，那么可以先把过期的数据提供给客户端，再在后台内存中更新缓存，下一次请求就会使用新缓存的数据了。这样做的缺点是，客户端会有极小部分请求不能获得最新数据，所以要根据业务需求决定是否使用过期数据。解决热数据缓存失效的指令如下。

指令：proxy_cache_background_update

语法：proxy_cache_background_update on | off;

默认值：proxy_cache_background_update off;

环境：http、server、location

含义：允许后台启动一个子请求去更新过期的缓存数据，并提供一个过期的缓存数据响应给客户端。前提是 proxy_cache_use_stale 的配置中包含 updating。

指令：proxy_cache_use_stale

语法：proxy_cache_use_stale error | timeout | invalid_header | updating | http_500 | http_502 | http_503 | http_504 | http_403 | http_404 | http_429 | off ...;

默认值：proxy_cache_use_stale off;

环境：http、server、location

含义：允许在某种条件下返回请求时使用过期的缓存数据。“某种条件”指后面有特定的参数，如后端服务返回错误状态码 500 或 timeout 等。proxy_cache_use_stale 也支持 updating，表示此请求的缓存正在被更新，并允许先使用过期数据返回给客户端。

再来看一个完整的配置。

```
   location / {
       proxy_cache cachedata;
       proxy_cache_valid        200 304  5m;
       proxy_cache_valid        301 302 2m;

       proxy_cache_key $scheme$host$uri$is_args$args;

       proxy_set_header         Host $host;
       proxy_set_header         X-Real-IP $remote_addr;
       proxy_set_header         X-Forwarded-For $proxy_add_x_forwarded_for;
       # 当缓存未命中时，提供过期缓存给客户端（前提是有过期缓存），并开启子请求去更新缓存
       proxy_cache_background_update on;
       # 当缓存未命中时，设置使用过期缓存的触发条件
       proxy_cache_use_stale      error   timeout   invalid_header   updating
http_502  http_503   http_504;
       # 当缓存未命中时，同一时间的相同请求只会回源一条到后端服务器上
       # 其他请求会继续等待，直到第一条请求拿回数据
       proxy_cache_lock on;
       # 获取请求的超时时间，如果超时，所有等待中的请求都会回源到后端服务器
       # 所以超时时间不建议设置太短
```

```
        proxy_cache_lock_timeout 20;

        proxy_ignore_headers   Vary    Set-Cookie ;
        proxy_pass http://test_servers;
    }
```

介绍了这么多指令后，再来说说 5.1 节的配置中出现过的$upstream_cache_status，根据其状态值，可以了解整个缓存系统的命中率及各项指标的情况，$upstream_cache_status 状态值和标识说明见表 5-3。

表 5-3　$upstream_cache_status 状态值和标识说明

状　态　值	标　　识
MISS	未命中，会从后端服务器获取新数据
BYPASS	当使用 proxy_cache_bypass 的请求时，会出现此状态，它表示请求强制从后端服务器获取数据
EXPIRED	命中过期缓存，并回源到后端服务器获取数据
STALE	命中过期缓存，使用过期缓存的原因与 proxy_cache_use_stale 有关
REVALIDATED	与 proxy_cache_revalidate 有关，缓存过期后会验证 if-modified-since 的请求头
UPDATING	当 proxy_cache_background_update 设置为 on 且命中过期的缓存时，就会显示此状态，表示缓存正在被更新
HIT	命中缓存，从缓存中获取数据

5.3.2　横向扩展最佳实践

URL 的种类越多，需要的缓存空间就越大，对硬盘 I/O 的要求也就越高，且为了保证服务在异常时有过期数据支撑，还需要保留大量的过期数据。那么如何才能轻松地完成横向扩展，避免因空间不足导致缓存数据被清除的情况呢？

关于 key，Nginx 的 key 存放在内存中，1MB 的空间大约可以存储 8000 个 key，因此首先需要评估可能会用到的 key 的数量。如果无法评估就尽量把空间大小设置得大一些，毕竟只要 500MB 就可以存放 400 万左右的 key 了，不必吝啬这点内存。

关于 value，proxy_cache 的缓存内容是存放在硬盘上的，所以会受到硬盘空间和硬盘 I/O 的影响。如果一台服务器有多个硬盘，建议都利用起来。

前面介绍过 proxy_cache_path 和 proxy_cache data 的参数，它们支持使用变量，可以利用这个优点，让请求动态地分配到一个变量上。Nginx 提供了 split_clients 指令来完成这项任务，将下面的配置写在 http{}块中：

```
http {
# proxy_cache_path 使用了变量$disk
```

```
    proxy_cache_path   /data1/nginxcache   levels=1:2   keys_zone=data_1:100m
inactive=7d max_size=1000g use_temp_path=off;
    proxy_cache_path   /data2/nginxcache   levels=1:2   keys_zone=data_2:100m
inactive=7d max_size=1000g use_temp_path=off;
    proxy_cache_path   /data3/nginxcache   levels=1:2   keys_zone=data_3:100m
inactive=7d max_size=1000g use_temp_path=off;
    proxy_cache_path   /data4/nginxcache   levels=1:2   keys_zone=data_4:100m
inactive=7d max_size=1000g use_temp_path=off;

    # 根据 URL 进行 hash 分片，将请求的百分比根据 hash 分片设置相应的值，并将此值赋给$disk
    split_clients $request_uri $disk {
        20%     1;
        20%     2;
        30%     3;
        *       4;
    }

    }
```

然后，再将 proxy_cache 设置为根据变量获取：

```
    proxy_cache data_$disk;
```

这样就可以充分利用硬盘空间和硬盘 I/O 了，条件允许的话使用固态硬盘效果会更好。

5.3.3 避免硬盘 I/O 阻塞

Nginx 在读取硬盘文件时会阻塞 worker 进程，这在一定程度上限制了 Nginx 并发读取文件的能力。在 1.7.11 版本之后，Nginx 提供了读取文件线程池的功能，可以利用线程功能同时读取多个文件。

thread_pool 指令只能配置在 main 块（和 error.log、worker_processes 的配置位置一样）中，默认值是“thread_pool default threads=32 max_queue=65536;”，代码如下：

```
    # 线程池数量，要和硬盘数一一对应，如果新增硬盘，就要新增线程池
    # thread_pool 后面跟的值依次代表线程池名字、线程数、最大等待队列
    thread_pool pool_1 threads=32 max_queue=65536;
    thread_pool pool_2 threads=32 max_queue=65536;
    thread_pool pool_3 threads=32 max_queue=65536;
    thread_pool pool_4 threads=32 max_queue=65536;
```

在 proxy_cache 下使用线程池：

```
    proxy_cache data_$disk;
    aio threads=pool_$disk;
```

该线程池也支持 Nginx 读取硬盘的其他内容，如静态文件。

注意：可以使用 gzip 压缩来减少网络 I/O。虽然线程池的引入解决了硬盘 I/O 阻塞的现象，但这并不意味着性能会大幅度提升。在实战中，建议压测一下真实数据，看性能是否达到了想要的效果。有时新增硬盘可能效果会更好。

5.3.4　集群模式

可以利用 Nginx 的 hash 分片功能，将多台 proxy_cache 缓存系统配置为集群模式，只需在 proxy_cache 缓存系统前端部署 Nginx 反向代理即可，代码如下：

```
upstream proxy_cache_servers  {
    hash $request_uri;
    server 172.18.1.5:8083  weight=20 max_fails=10 fail_timeout=30s;
    server 172.18.1.6:8083  weight=20 max_fails=10 fail_timeout=30s;
    server 172.18.1.7:8083  weight=20 max_fails=10 fail_timeout=30s;
    keepalive 300;
}
```

如果将 proxy_cache 缓存系统直接部署到反向代理上，可以减少因 HTTP 与缓存系统建立连接而产生的开销，但它的前端仍然需要一个 hash 分片的负载均衡器，除非允许每台服务器都缓存相同的数据，并且在请求未命中时，后端服务器能承受成倍增加的请求压力。

注意：新增缓存节点会导致 hash 重新计算，对缓存的命中率有很大的影响，因此，应该尽量在低峰期加入新节点。proxy_cache_key 可以自定义 key，让同一个 URL 支持使用多个版本进行缓存，或者按某种特定 Cookie 类型进行缓存。

5.4　高可用方案

当请求使用分片功能将压力分散到多台缓存系统时，如果其中一台服务器宕机该怎么办呢？数据一旦丢失，后端压力就会暴增，因此需要对缓存的高可用性进行设计。下面是几种可选的方案。

1．多级缓存

即在 proxy_cache 缓存系统的前面再加一层缓存系统。如果请求的 URL 比较集中，那么缓存的内容就比较少，此时建议使用 varnish 来做第一层缓存，因为 varnish 的缓存可以存放在内存中，性能更加稳定。双缓存系统可以减少宕机造成的性能问题。如果服务器内存资源比较紧张，建议放弃 varnish，再做一层 proxy_cache 缓存系统。

优点：双保险，稳定性高。

缺点：多级缓存会导致缓存的有效期存在一定的误差。

2．硬盘同步备份数据

即每次保留缓存系统数据时都将数据在其他地方备份一份。在硬盘 I/O 允许的情况下，制作 rsync 同步服务器，将硬盘的数据同步到新的服务器上存放。在当前服务器出现故障时，可以使用备份服务器上的数据。

优点：服务部署相对简单，不需要考虑双缓存配置嵌套的逻辑。

缺点：硬盘 I/O 的压力会增加。

3．使用网络 I/O

如果硬盘同步备份的方案导致 I/O 压力过大，影响到了正常服务。可以考虑使用网络 I/O 来减少硬盘压力。

Nginx 把从硬盘读取到的缓存数据存放到内存中，再通过网络传输到客户端。此时，可以把内存中的缓存数据备份到指定位置，并使用 Lua-Nginx-Module 去读取出来。

优点：利用网络 I/O，减少硬盘同步的 I/O。

缺点：增加网络 I/O 的压力和缓存系统的 CPU 资源消耗。

5.5 proxy_cache 配置模板

下面为本章出现过的内容提供一个完整的配置示例，以供读者参考：

```
user  webuser webuser;
worker_processes  8;

thread_pool pool_1 threads=32;
thread_pool pool_2 threads=32;
thread_pool pool_3 threads=32;
thread_pool pool_4 threads=32;

error_log  logs/error.log  error;

pid /data/nginx.pid;

worker_rlimit_nofile 65535;
```

```
    events {
        worker_connections  65535;
        multi_accept on;
        use epoll;
    }

    http {

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

        log_format  main  '$remote_addr - $remote_user [$time_local] "$request"
"$status $body_bytes_sent "$http_referer" "disk:data_$disk" "cache_status:$upstream_
cache_status"';

        access_log /data/access.log main;

        keepalive_requests 1000;
        keepalive_timeout 60;
        client_max_body_size 300m;
        client_body_buffer_size 512k;
        reset_timedout_connection on;

        sendfile on;
        tcp_nopush on;
        tcp_nodelay on;


        gzip on;
        gzip_min_length  1k;
        gzip_buffers     16 8k;
        gzip_comp_level 7;
        gzip_types        text/plain text/css application/x-javascript text/
xml  application/xml  application/xml+rss  text/javascript  application/json
application/javascript;
        gzip_vary on;

        proxy_connect_timeout   5;
        proxy_send_timeout      30;
        proxy_read_timeout      60;
        proxy_buffering         on;
        proxy_next_upstream error http_500 http_502  http_504 timeout;
        proxy_next_upstream_tries 2;
        proxy_next_upstream_timeout 0;
```

```
       upstream test_servers {
          server 127.0.0.1:81  max_fails=5 fail_timeout=10s weight=10;
          server 127.0.0.1:82  max_fails=5 fail_timeout=10s weight=10;
          keepalive 100;

      }

       proxy_cache_path  /data1/nginxcache  levels=1:2  keys_zone=data_1:100m
inactive=7d max_size=1000g use_temp_path=off;
       proxy_cache_path  /data2/nginxcache  levels=1:2  keys_zone=data_2:100m
inactive=7d max_size=1000g use_temp_path=off;
       proxy_cache_path  /data3/nginxcache  levels=1:2  keys_zone=data_3:100m
inactive=7d max_size=1000g use_temp_path=off;
       proxy_cache_path  /data4/nginxcache  levels=1:2  keys_zone=data_4:100m
inactive=7d max_size=1000g use_temp_path=off;

       split_clients $request_uri $disk {
           20%    1;
           20%    2;
           30%    3;
           *   4;
       }

       server {
           listen       80;
           location / {
               proxy_cache data_$disk;
               aio threads=pool_$disk;
               proxy_cache_valid        200 304  5m;
               proxy_cache_valid        301 302 2m;

               proxy_cache_key $scheme$host$uri$is_args$args;

               proxy_set_header Host $host;
               proxy_set_header X-Real-IP $remote_addr;
               proxy_set_header X-Forwarded-For $proxy_add_x_forwar ded_for;
               proxy_http_version 1.1;
               proxy_set_header Connection "";

               proxy_cache_background_update on;
               proxy_cache_use_stale  error timeout invalid_header updating
http_502  http_503   http_504;
```

```
                proxy_cache_lock on;
                proxy_cache_lock_timeout 10;

                proxy_ignore_headers   Vary    Set-Cookie ;
                proxy_pass http://test_servers;
            }
        }
    }
```

5.6　小结

Nginx 的 proxy_cache 提供了 10 多个指令，结合日常的维护经验，充分利用硬盘 I/O、网络 I/O、CPU 等资源，可以极大地提升缓存系统的性能和高并发能力。通过对这些指令的解读，也可以了解到一个缓存系统需要具备的先决条件。

第 6 章
引入 Lua

第 2 章到第 5 章主要介绍了 Nginx 的基础配置及相关模块，有些读者可能已经感到乏味，是时候展示“真正”的技术了，让我们带着强烈的好奇心走进这一章。

6.1 引入 Lua 的原因

随着业务需求的不断深入，单纯地用 Nginx 的配置已经无法满足需求，特别是在面对定制业务时，例如如下的这些情况。

1．Nginx 缺少 if…elseif…else 的简单逻辑，如果要实现类似功能，就需要多次嵌套 if，配置烦琐且可读性差。

2．Nginx 缺少原生的大于、等于和小于等判断类型的表达式，导致一些简单逻辑的实现也比较复杂。

3．Nginx 缺少动态限速功能，如需相关规则生效则要重启 Nginx，且第三方和原生的配置都缺乏灵活性。

4．Nginx 缺少动态路由功能，如需根据请求的路由动态地调整后端服务器和转发规则，则要重启 Nginx 进程。

5．Nginx 在 API 网关系统中未实现智能化。

6．Nginx 与数据库交互的能力有限。

7．大多数 Nginx 模块是使用 C 语言开发的，且开发人员还需要了解 Nginx 内部的构造，开发难度较大。

笔者在 2013 年年初将 Lua 引入 Nginx 反向代理中，除为了解决上述常见问题外，还参考了当时 Lua 在 Nginx 中的使用情况，具体如下。

1．Tengine（由淘宝网发起的 Web 服务项目）在 2011 年年底宣布开源，其由 Nginx+Lua 开发的优点凸显出来，并在淘宝网等网站开发中广泛应用。

2．章亦春推出兼容 Nginx 版本的 OpenResty 开源软件。Lua-Nginx-Module 是其中的核心模块，可以直接编译到 Nginx 中。作为第三方模块，Lua-Nginx-Module 弥补了 Nginx 的很多不足。

3．Lua 代码学习成本较低，用 1~2 周时间了解 Nginx 的配置和执行阶段后，就可以上手开发，可快速组建开发团队。

4．Lua 开发更为灵活。在使用过程中，开发人员对 Nginx 源码的侵入很少，且不需要过多地深入 Nginx 源码就可以快速迭代产品的功能，再加上 Lua 有着“全世界最快脚本语言”的称号，性能方面也有保障。

注意：本书学习 Lua 的目的是使用 Lua-Nginx-Module 等和 Lua 有关的模块，这些模块已经封装了大量的指令，能够帮助我们简化代码，提升代码性能。本章对 Lua 的学习以数据结构和语法为主。

6.2　Lua 和 LuaJIT

Lua 是巴西里约热内卢天主教大学（Pontifical Catholic University of Rio de Janeiro）的一个研究小组（由 Luiz Henrique de Figueiredo、Roberto Ierusalimschy 和 Waldemar Celes 组成）在 1993 年开发的，其特点如下。

- Lua 由标准 C 语言编写而成，可在绝大多数系统上运行。
- Lua 可以轻松地和 C、C++互相调用，性能强大，非常灵活。
- Lua 5.1 版本在编译后还不到 200KB，是一个轻量化脚本语言。

LuaJIT 是采用 C 语言编写的 Lua 即时编译器，兼容 Lua 5.1 版本，因此标准的 Lua 代码都可以在 LuaJIT 上运行。LuaJIT 把 Lua 代码即时编译（Just-In-Time）成本地机器码后交由 CPU 执行，与标准 Lua 相比，不仅显著提升了性能，且能够支持更多的特性，因此本书所讲的 Lua 大都是基于 LuaJIT 版本的（如果不是会进行说明）。

6.3　环境搭建

首先，需要安装 LuaJIT，目前常用的版本是 LuaJIT 2.0.5 和 LuaJIT 2.1.0-beta3，这里推荐

使用 LuaJIT 2.1.0-beta3，它拥有更多优良的特性，并且 Lua-Nginx-Module 的作者对 LuaJIT 2.1.0-beta3 共享了很多代码，使其更适合 Nginx+Lua 的使用环境，下面的安装就使用了 LuaJIT 2.1.0-beta3 的版本。

LuaJIT 2.1.0-beta3 的安装方式如下：

```
    #   wget   -S   https://codeload.github.com/openresty/luajit2/tar.gz/v2.1-
20181029  -O  LuaJIT-OpenResty-2.1.0-beta3.tar.gz
    # tar -zxvf LuaJIT-OpenResty-2.1.0-beta3.tar.gz
    # cd luajit2-2.1-20181029/
    # make && make install
```

将 lib 的路径写入/etc/profile 目录中，确保环境变量生效，示例如下：

```
    # vim /etc/profile
    export LUAJIT_LIB=/usr/local/lib
    export LUAJIT_INC=/usr/local/include/luajit-2.1
    # source  /etc/profile
```

安装后执行如下命令：

```
    # luajit -v
    LuaJIT    2.1.0-beta3    --    Copyright    (C)    2005-2017    Mike    Pall.
http://luajit.org/
```

上述输出结果显示了 LuaJIT 的版本号，表明安装成功。

下面就来体验一下编程的经典语句“Hello, World”吧：

```
    # echo  'print("Hello, World")' >test.lua
    # cat test.lua
    print("Hello, World")
    # luajit test.lua
    Hello, World
```

注意：在 Lua 语法中，注释采用两个中横线--而不是#来表示，这和 Shell、Python 及 Nginx 中的语法都不一样。

6.4 Lua 的数据类型

Lua 的变量不需要声明数据类型，在使用时直接赋值即可。Lua 支持 8 个基本数据类型，分别为 nil、boolean、number、string、table、function、thread 和 userdata。

6.4.1 类型说明

Lua 的数据类型说明见表 6-1。

表 6-1　Lua 的数据类型说明

数 据 类 型	描 述 说 明
nil	通常代表无效值，用于判断变量是否存在，如果不存在就是 nil，在表达式中使用代表 false
boolean	布尔类型，可选值为 true/false
number	在 LuaJIT 中，number 支持 long int 和 long long int 这两种数据类型，即赋值时是整数就存放整数，是浮点数就用双精度浮点数；而在 Lua 中，number 表示双精度类型的实浮点数
string	字符串类型，可以写在双引号或单引号之间，也可以写在[[]]之间
table	table 其实是一个关联数组，数组的索引可以是数字、字符串，也可以是除 nil 类型以外的任何一种 Lua 类型
function	函数类型，可以用 Lua 语言编写，也可以用 C 语言编写
thread	独立执行的线程，用于实现协同程序。Lua 支持所有系统上的协同程序
userdata	允许将任意 C 语言数据存储在 Lua 变量中

通过 tpye 命令可以获取变量的类型，示例如下：

```
print(type("test lua"))          --> 输出  string
print(type(233+2))               --> 输出  number
print(type(true))                --> 输出  boolean
print(type(nil))                 --> 输出  nil
print(type(type))                --> 输出  function
```

6.4.2　类型示例

下面是 8 种数据类型的使用示例，在 Linux 下输入 luajit -i 进入命令行 ：

```
# luajit -i
```

- nil

示例如下：

```
> print(x)
nil
> print(type(a))
nil
```

在上述示例中，变量 x、变量 a 没有被赋值，所以数据类型为 nil。

- boolean（布尔）

编写一个 test.lua 脚本，代码如下：

```
local   str   =    ''
local   n    =   nil

if      str     then
```

```
        print('str')
    else
        print("not str")
    end

    if      n       then
        print(n)
    else
        print("not n")
    end
```

执行该脚本：

```
    # luajit test.lua
    str
    not n
```

- number（数字）

示例如下：

```
    local a = 3
    local b = '3'
    print(type(a))    -->  输出 number，注意数字不能加引号
    print(type(b))    -->  输出 string
```

LuaJIT 支持数据类型 long long int，但 Lua 5.1 不支持，在 Lua 命令行中打印 long long int 数据会直接报错，示例如下：

```
    # luajit -i
    LuaJIT   2.1.0-beta3   --   Copyright   (C)   2005-2017   Mike   Pall.
http://luajit.org/
    JIT: ON SSE2 SSE3 SSE4.1 BMI2 fold cse dce fwd dse narrow loop abc sink
fuse
    > print(9223372036854234201LL-1)
    9223372036854234200LL

    # lua
    Lua 5.1.4  Copyright (C) 1994-2008 Lua.org, PUC-Rio
    > print(9223372036854234201LL-1)
    stdin:1: malformed number near '9223372036854234201LL'
```

- string（字符串）

字符串可以写在双引号或单引号之间，也可以写在[[]]之间。[[]]主要用于字符串较长且包含换行的情况时，示例如下：

```
# vim test.lua
str1 = "str1"
str2 = 'str2'
str3 = [[
-----------------
this
is
str3
-----------------
]]

print(str1)
print(str2)
print(str3)
```

执行该脚本：

```
# luajit test.lua
str1
str2
-----------------
this
is
str3
-----------------
```

如果一个字符串中只包含数字，那么进行算术操作时，Lua 会将其转换为 number 类型，示例如下：

```
# luajit
LuaJIT 2.1.0-beta3 -- Copyright (C) 2005-2017 Mike Pall.
http://luajit.org/
JIT: ON SSE2 SSE3 SSE4.1 BMI2 fold cse dce fwd dse narrow loop abc sink
fuse
> print("45" - 2)
43
> print("3" * "2")
6
> print("a" + 1)
stdin:1: attempt to perform arithmetic on a string value
stack traceback:
    stdin:1: in main chunk
    [C]: at 0x00404180
> print("a22" + 1)
stdin:1: attempt to perform arithmetic on a string value
```

```
stack traceback:
    stdin:1: in main chunk
    [C]: at 0x00404180
```

只包含数字的字符串能够进行加减乘除等算术运算，但若字符串中还包含其他类型，进行算术运算的话就会报错。

- table

示例如下：

```
-- 直接创建一个空 table
local n_table = {}

-- 使用赋值的方式生成一个初始 table
local n_table  = {"apple", "pear", "orange", "grape"}
```

下面是一个“大杂烩”的 table：

```
local   n_table =
{
test =  "testnginx", -- 索引为字符串
city = {"shanghai","wuhan","chengdu","beijing"},
                     -- 索引为字符串，值仍然为 table

"sada",              -- 没有直接提供索引，只有值，会使用数字索引，这个位置对应的是 1
[17]    =  360,      -- 直接使用数字作为索引，索引值是 17
12312                -- 没有直接提供索引，只有值，会使用数字索引，这个位置对应的是 2
}

print(n_table.test)      -->输出 testnginx
print(n_table.city[2])   -->输出 wuhan
print(n_table[1])        -->输出 sada
print(n_table[2])        -->输出 12312
print(n_table[17])       -->输出 360
```

注意：Lua 的 table 中的数字索引值都是从 1 开始的，和部分语言的索引值从 0 开始有所区别。table 还有更多其他特性，详见 6.10 节中的说明。

- function（函数）

示例如下：

```
# vim test.lua
function res(st)
    if st == 'test' then
        return 'this is test'
```

```
    else
        return 'this is not test'
    end
end
print(res("test"))   -- 直接作为函数执行
local a = res        -- 函数可以作为变量赋值给其他变量，local 声明了一个局部变量
print(a("123"))      -- 执行 a 函数
```

执行该脚本：

```
# luajit test.lua
this is test
this is not test
```

- thread（线程）

Lua 中最主要的线程是协同程序（coroutine），它跟线程（thread）比较相似。线程与协同程序（以下简称为协程）的区别是，线程可以在同一时间运行多个，而协程同一时间只能运行一个，协程在显式调用 yield 函数后才被挂起。

- userdata

字面理解就是用户数据，属于自定义的数据。Lua 允许将任意 C 语言数据存储在变量中，自定义数据的值不能在 Lua 中创建或修改，只能通过 C 语言处理，这就保证了程序数据的完整性。

6.5　表达式

6.5.1　算术运算符

算术运算符及其说明见表 6-2。

表 6-2　算术运算符及其说明

算术运算符	说　明
+	加法
-	减法
*	乘法
/	除法
%	取余
^	指数

示例如下：

```
# vim test.lua
print("加法 1+2=", 1+2 )
print("减法 1-2=", 1-2 )
print("乘法 1*2=", 1*2 )
print("除法 1/2=", 1/2 )
print("取余 1%2=", 1%2 )
print("指数 2^2=", 2^2 )
```

执行上面的 Lua 脚本：

```
# luajit test.lua
加法 1+2= 3
减法 1-2= -1
乘法 1*2= 2
除法 1/2= 0.5
取余 1%2= 1
指数 2^2= 4
```

6.5.2 关系运算符

关系运算符及其说明见表 6-3。

表 6-3 关系运算符及其说明

关系运算符	说　明
>	大于
<	小于
>=	大于或等于
<=	小于或等于
~=	不等于
==	等于

示例如下：

```
# vim test.lua
a = 14
b = 7

if( a == b )
then
    print("a 等于 b" )
else
    print("a 不等于 b" )
```

```
end

print('a>b',a>b)
print('a~=b',a~=b)
print('a<b',a<b)
```

执行结果如下：

```
# luajit test.lua
a 不等于 b
a>b true
a~=b true
a<b false
```

从上述执行结果可以看出关系运算符得到的结果是布尔类型，结果非 true 即 false。

6.5.3　逻辑运算符

逻辑运算符及其说明见表 6-4 所示。

表 6-4　逻辑运算符及其说明

逻辑运算符	说　　明
and	逻辑与
or	逻辑或
not	逻辑非

示例如下：

```
# vim test.lua
a = "test"
b = nil
c = ''  -- 空字符串，不是 nil

if (a and b) then
    print("a and b is true")
else
    print("a and b is false")
end

if (a or b)  then
    print("a or b is true")
else
    print("a or b is false")
end
```

```
if ( not( a and b) )  then
    print("not( a and b) is true")
else
    print("not( a and b) is false")

end

print('a and c : ',a and c)
print('a or c : ',a or c )
print('not(a and c): ',not(a and c))
```

执行结果如下：

```
a and b is false
a or b is true
not( a and b) is true
a and c :      -- 此处请注意，因为 c 为空字符串，所以输出了一个空字符串
a or c : test
not(a and c): false
```

从上述执行结果可以得出如下结论。

- a and b 的作用：当 a 为 false 时，则返回 a，否则返回 b。
- a or b 的作用：当 a 为 true 时，则返回 a，否则返回 b。
- not(a and b)的作用：和逻辑与的作用相反，当 a and b 为 true 时，则返回 false。

6.5.4 字符串连接和字符串长度计算

在 Lua 中字符串使用..进行连接，示例如下：

```
# vim test.lua
print("a" .. "b")
print(1 .. "x")
print(1 .. 2)
```

执行结果如下：

```
luajit test.lua
ab
1x
12
```

注意：如果有纯数字的字符串，在连接过程中会将数字转换为字符。

在 Lua 中，使用#返回字符串或数组的长度，示例如下：

```
# vim test.lua
print(#("a" .. "b"))
print(#"sxjk19dasdas")
print(#"")
print(#"12 12 12")
local t1 = {"a",  "c" , "b" , nil}  -- 数组，table 类型
print(#t1)
```

执行结果如下：

```
# luajit test.lua
2           -- 字符串连接后再计算
12          -- 长字符串统计
0           -- 空字符串为 0
8           -- 空格也是一个字符串
3           -- 返回数组的长度，如果数组里有 nil，则不会被计算在内
```

6.5.5　运算符优先级

运算符优先级从高到低的顺序如下。

1．^

2．not　　#　　-　　（这里的“-”是负号，不是减号）

3．*　/　%

4．+　-　　（这里的“-”是减号）

5．..

6．<　>=　==　~=

7．and

8．or

如果一段代码存在较多的运算符种类，建议使用小括号来设置优先级，这样会显得逻辑比较清晰。

6.6　变量

变量的默认值是 nil，使用变量前必须在代码中声明，目前 Lua 支持全局变量、局部变量和表中的域这 3 种变量。

6.6.1 全局变量

在声明变量时，如果前面不加 local，则为全局变量。如果在自定义变量时使用了全局变量，那么此变量会进入全局变量的命名空间，而在使用此变量时，会查询全局变量的 table（全局变量存放在一个 table 中），频繁读取 table 会影响 CPU 性能。建议尽量不要在代码中使用全局变量，而使用局部变量。

6.6.2 局部变量

在声明变量时，如果在前面加上 local，则为局部变量。使用局部变量的好处如下。

1．不会“污染”全局命名空间。

2．性能好于全局变量，不用查询全局变量的 table。

3．拥有生命周期。当局部变量在定义的作用域外使用时，此变量的生命周期就会结束，有利于垃圾回收。

下面的例子可以看出全局变量和局部变量的使用区别：

```
# vim test.lua
function test()
    local a = 'local var'
         b = 'global'
    print('test() output:',a)
end

test()  -- 执行 test 函数

print('a:',a)
print('b',b)
```

执行结果如下：

```
# luajit test.lua
test() output: local var    -- 在 test 函数的作用域内可以输出变量的值
a: nil          -- a 为局部变量，在离开 test 函数的作用域后，a 就不存在了，所以为 nil
b: global       -- b 为全局变量，在离开作用域后仍然存在值
```

6.6.3 变量赋值

通过赋值可以创建或改变一个变量的值，示例如下：

```
local a = 1
```

```
local a = a + 2
local b = a .. "1"  -- 在字符串连接中，如果有数字，则数字会自动转换为字符串

print("a:" ,a)       -- 输出为 3
print("b:" ,b)       -- 输出为 31
```

Lua 支持同时对多个变量赋值，也支持变量之间交换赋值，示例如下：

```
# vim test.lua
local a, b, c = 1, 'test', 'x'
print("a:", a)
print("b:", b)
a, b = b, a  -- 对两个变量的值进行互换
print("a:", a)
print("b:", b)
```

执行结果如下：

```
# luajit test.lua
a: 1
b: test
a: test
b: 1
```

在对多个变量进行赋值的过程中，如果被赋值的变量（等号左边的）数量大于值的数量（等号右边的），则多余的变量会被赋值为 nil，赋值顺序是从左到右。如果被赋值的变量数量小于值的数量，则多余的值会被忽略。

6.7　流程控制

6.7.1　if-else

关于 if-else 语法，示例如下：

```
# vim test.lua
local a = 70
if (a > 100) then
  print(a .. '>100')
elseif ( a <= 100 and  a >= 50)  then
  print(a,'Between 50 and 100')
else
  print(a .. '<50')
end
```

执行结果如下：

```
# luajit test.lua
70 Between 50 and 100
```

6.7.2 for 循环

关于 for 循环语法，示例如下：

```
for var=exp1,exp2,exp3 do
    something
end
```

在 for 循环语法中，var 的值从 exp1 变化到 exp2，变化值以 exp3 为步长；其中 exp3 为可选值，如果不填，则默认为 1，示例如下：

```
# vim test.lua
for i=5,1,-1 do
    print(i)
end
```

执行结果如下：

```
# luajit test.lua
5
4
3
2
1
```

泛型 for 循环通过一个迭代器函数遍历所有值，这在 Lua 中比较常见，示例如下：

```
# vim test.lua
local a = {'1', '2', '3', '4', 'test', 'test1'}

for i,v in ipairs(a) do
 print("index:" .. i , "value:" .. v)
end
```

执行结果如下（其中 index 是数组的索引值，value 是索引对应的值）：

```
# luajit test.lua
index:1 value:1
index:2 value:2
index:3 value:3
index:4 value:4
index:5 value:test
index:6 value:test1
```

6.7.3　while 循环

在使用 while 循环时，如果表达式的值为 false，则会退出循环，示例如下：

```
# vim test.lua
local a=6
while( a < 10 )
do
    a = a+1
    print("a:", a)
end
```

执行结果如下：

```
# luajit test.lua
a: 7
a: 8
a: 9
a: 10  -- 10<10 是 false，所以循环退出
```

6.7.4　break 和 return

在循环过程中，如果需要在特定条件下退出循环，可以使用 break，示例如下：

```
# vim test.lua

local a=6
while( a < 12 )
do
    a = a+1
    print("a:", a)
    if (a == 9) then
       print("it is:",a)
       break
    end
end
```

执行结果如下：

```
# luajit test.lua
a: 7
a: 8
a: 9
it is: 9  -- 当 a 等于 9 时退出循环，不再继续执行代码
```

return 命令通常在函数中使用，主要作用是返回执行结果。一旦 return 被触发，当前操作就会立即退出，不再执行函数后面的代码，那么调用该函数得到的将是 return 的返回值，示例

如下：

```
# vim test.lua

local function sum_num(a)

    local  b = a + 1
    if (b == 3)  then
        return a .. "---match"
    else
        return "no match"
    end
   print('123')

end

local num = sum_num(2)
print(num)
```

执行结果如下：

```
# luajit test.lua
2---match
```

从执行结果可以看出 print('123')这条语句并没有执行，因为在执行 return 之后，当前操作就退出了函数。

6.8 函数

在 6.4 节中对函数进行了简单介绍，在 Lua 开发中会经常使用函数，本节将对函数进行详细的讲解。

6.8.1 函数格式

使用 function 关键字来定义函数：

```
local function function_name(argument1,argument2,argument3...)
    function_body
    return function_result
end
```

针对上述代码进行分析，使用函数时需要注意如下情况。

- function：上述代码声明了一个名称为 function_name 的局部函数，如果 function 前面不

加 local 则为全局函数，但应尽量避免使用全局函数。

- (argument1,argument2,argument3...)：代表函数的参数，Lua 支持函数有多个参数或不带参数；当存在多个参数时，参数之间用逗号分隔。
- function_body：是函数的主体内容，即需要执行的语句。
- return function_result：即返回函数结果，支持返回多个值。

示例如下：

```
# vim test.lua
local function sum_num(a,b)
    local c = a - b
    local d = a + b
    return c,d
end

local c,d = sum_num(2,1)
print(c,d)
```

执行结果如下：

```
# luajit test.lua
1 3
```

6.8.2　传参方式

函数的参数是可选的，如果函数不需要使用参数，在执行函数时，加上小括号即可，示例如下：

```
# vim test.lua

local function sum_num()
    local a = 2
    local b = 1
    local c = a - b
    local d = a + b
    return c,d
end

local c,d = sum_num()  -- 执行函数
print(c,d)  -- 输出结果 1  3
```

如果函数的参数数量不确定，需要使用…来表示该函数支持多变参数，示例如下：

```
# vim test.lua
```

```
local function sum_num(...)
    local a = "a"
    for i, v in ipairs{...} do
        a = a .. v
    end
    return a
end
local a = sum_num("b","c","d","1")
print(a)
```

执行结果如下：

```
# luajit test.lua
abcd1
```

6.8.3 函数的创建位置

函数需要在调用该函数的代码上方创建，否则调用该函数就相当于调用了一个 nil 变量，示例如下：

```
# vim test.lua
-- 调换 5.7.1 节中函数的位置
local c,d = sum_num(2,1)
print(c,d)

local function sum_num(a,b)  -- 函数的创建位置在调用函数的下面
    local c = a - b
    local d = a + b
    return c,d
end
```

执行结果（很明显会报错）如下：

```
# luajit test.lua
luajit: test.lua:1: attempt to call global 'sum_num' (a nil value)
stack traceback:
    test.lua:1: in main chunk
    [C]: at 0x00404180
```

6.9 模块

Lua 5.1 版本开始支持模块，Lua 的模块是由变量、函数等组成的 table，通过模块可以轻松调用各类函数，减少代码的重复编写，提升代码的可读性。

6.9.1　模块格式

在 Lua 中创建模块很简单：创建一个 table，table 的名称就是模块名，然后将函数、变量等导入这个 table 即可，示例如下：

```
# vim md.lua
-- 定义模块的名称
local m = {}
-- 定义一个变量
m.str1 = "a"
-- 定义一个私有函数，即局部函数
local function func_local()
    print("Are you ok!")
end
-- 定义一个全局函数
function m.func()
    print("I am here !")
    func_local()
end
return m
```

6.9.2　加载模块

在创建模块后，通过 require 将模块加载到代码中，如下所示：

```
require("文件名")
```

在使用模块时需要注意如下情况。

- 当加载模块时，文件名不需要写.lua 后缀。
- 当加载模块时，会查询文件的位置，如果加载了一个不存在的模块，则会抛出错误信息，示例如下：

```
# luajit test.lua
luajit: test.lua:1: module 'mdx' not found:
    no field package.preload['mdx']
    no file './mdx.lua'
    no file '/usr/local/share/luajit-2.1.0/mdx.lua'
    no file '/usr/local/share/lua/5.1/mdx.lua'
    no file '/usr/local/share/lua/5.1/mdx/init.lua'
    no file './mdx.so'
    no file '/usr/local/lib/lua/5.1/mdx.so'
    no file '/usr/local/lib/lua/5.1/loadall.so'
```

```
stack traceback:
    [C]: in function 'require'
    test.lua:1: in main chunk
    [C]: at 0x00404180
```

根据上述错误信息，可以间接确认查找文件是按照由上至下的顺序进行的，且后缀名会发生变化。

下面来试着加载一下 6.9.1 节创建的 md.lua 模块：

```
# vim test.lua
local m = require("md")  -- 加载 md.lua 模块
print(m.str1)            -- 执行模块中的一条语句，输出变量
m.func()                 -- 执行模块中的一个函数
```

执行结果如下：

```
# luajit test.lua
a
I am here !  -- m.func() 输出了 2 行，下面一行是执行局部函数 func_local 的结果
Are you ok!
```

局部函数又叫作私有函数，它不能直接被调用，只能在模块内部被调用，如果从外部调用，则会报错：

```
# vim test.lua f
local m = require("md")
m.func_local()
```

执行结果会返回报错信息，如下所示：

```
# luajit test.lua
luajit: test.lua:4: attempt to call field 'func_local' (a nil value)
stack traceback:
    test.lua:4: in main chunk
    [C]: at 0x00404180
```

6.10 Lua 常见操作

Lua 还有很多语法特性，本节先来讲解几种常见的操作。

6.10.1 操作 table

首先，介绍一下 table 常见的操作，见表 6-5。

表 6-5　table 常见的操作

常见操作	使用说明
local tabel = {}	创建一个空的 table
table.insert(table,value)	默认插入到 table 的最后位置
table.insert(table,pos,value)	在指定的索引位置（pos）插入值，Lua 中的索引值是从 1 开始的
table.getn(table)	获取 table 的长度，与使用#获取长度的方式类似
table.concat(table, sep, start, end)	将 table 用 sep 连接符进行连接，sep 默认是空字符串； start 是开始的索引位置，end 是结束的索引位置； start 和 end 是可选参数，如果不填写 start 和 end，则整个 table 将会被连接起来，连接后的值是字符串格式的
table.maxn (table)	获取 table 的最大索引值
table.remove (table)	默认删除表的最后一个元素
table.remove (table, pos)	删除指定位置的值，pos 是索引值
table.sort(table)	对表进行升序排列
table.sort(table, func)	使用比较函数 func 对表进行排序，如 func 可以是 local func_sort = function(a, b) return b < a end

示例如下：

```
local a = {1, 8}
print(table.getn(a))      -- 长度为 2
table.insert(a, 1, 5)     -- 在索引位置 1，插入值为 5，即 table 的第 1 个数字会变成 5
table.insert(a,123124)    -- 在末尾插入 123124
print(table.concat(a, ","))  -- 以逗号连接 a 的所有值，并输出 5,1,8,123124
print(table.getn(a))          -- 长度为 4
print(table.maxn(a))          -- 最大的索引值为 4
print(#a)                     -- 长度为 4
--table.remove(a,2)
table.sort(a)                 -- 升序排列，a 将变为排序后的结果
print(table.concat(a, ","))   -- 输出结果为 1,5,8,123124

                              -- 声明一个函数，这里是一个倒序排序
local func_sort = function(a, b) return b < a end
table.sort(a,func_sort)
print(table.concat(a, ",",1,3))   -- 输出索引从 1 到 3 的值为 123124,8,5
```

6.10.2　定义字符串

一般情况下，使用单引号''或双引号" "来定义一个字符串。如果字符串的长度太长，或者需要一个不被转义的特殊字符的字符串，可以采用[[]]，包含在[[]]内部的引号、回车符等将不会被转义，且支持多行字符合并。

如果字符串中有[]作为字符来使用，则需要用到[=[]=]，示例如下：

```
local a = [["a","d","s"]]
```

```
print(a)
-- 一级长括号
local b = [=[[[0-9]+
"x a"\n \\  ]]] [[]] [[]]  ]]=]
print(b)
```

输出内容：

```
"a","d","s"
[[0-9]+
"x a"\n \\  ]]] [[]] [[]]  ]
```

6.10.3 字符串连接

在 Lua 中，可以使用..来进行字符串连接，但如果有大量的字符串要连接起来，使用..会损耗一定的性能。此时，推荐使用 table.concat，以提升性能和灵活度，示例如下：

```
local tabel_s = {};
for i = 1,10, 1 do
    tabel_s [i] = "a";
end
print(table.concat(tabel_s,","))  -- 输出结果为 a,a,a,a,a,a,a,a,a,a
```

6.11 引入 Lua 的插曲

笔者第一次使用 Lua，并不是因为一个很大的项目需求。那是 2013 年的时候，当时有一个需求要对某个 URL 设置 a、b 两个版本的缓存（如图 6-8 所示），需要根据用户 Cookie 中的最后一次访问时间进行缓存配置，如果大于某个时间就使用 a 版本的页面，小于或等于某个时间，就使用 b 版本的页面，即同一个 URL 要根据 Cookie 的值设置两个版本的页面。

如果把对 Cookie 的判断放到后端服务器上，那么，每次请求都会在后端服务器上进行计算，前端缓存系统将无法起作用。如果放到 Nginx 中进行判断，则需要在大于某个时间时，新增一个 version 等于 a 标识的请求头；在小于或等于某个时间时，新增一个 version 等于 b 标识的请求头；缓存系统存放的 key 使用 URL+version 请求头，在请求代理到后端后，需要对请求头进行判断并返回对应的结果给缓存系统。

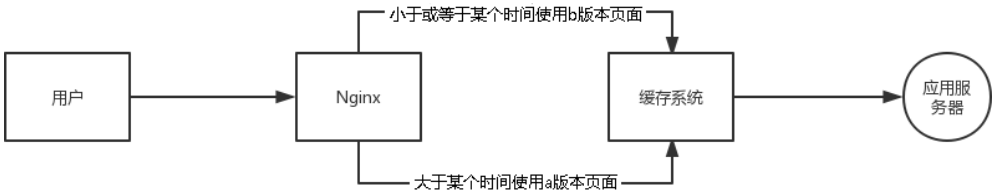


图 6-1 对某个 URL 设置 a、b 两个版本的缓存

可是 Nginx 不支持进行大于、小于判断，考虑到使用 Nginx 进行逻辑判断的复杂度会随着业务量的增加而越来越大，笔者决定引入 Lua，引入 Lua 后，代码就变得简单多了，如下所示：

```
location  = /ab {
            -- Lua 语法的开始位置
        rewrite_by_lua '
            -- 导入第三方模块
            local ngx = require "ngx"
            -- 获取用户 Cookie 的 frist_time，如果没有获取到，则赋值为 0
            local frist_time = tonumber(ngx.var.cookie_frist_time) or 0
            if frist_time > 12392 then
            -- 大于 12392 就添加一个 vesion 为 a 的请求头
                ngx.req.set_header("Version", "a")
            else
            -- 小于或等于 12392 就添加一个 vesion 为 b 的请求头
                ngx.req.set_header("Version", "b")

            end
        ';
            -- Lua 语法的结束位置
        proxy_pass http://cache_servers;
}
```

未使用过 Nginx+Lua 进行开发的读者可能会对上面的配置不太理解，但是可以先了解一下此代码的格式（非常清晰明了），后面将会对其进行详细讲解。

6.12　小结

本章对 Lua 的基础语法进行了讲解，后续章节会根据实际需要对 Lua 进行更深入的介绍，届时，读者将会发现很多常用指令都来自 Lua-Nginx-Module。

第 7 章
Lua-Nginx-Module 常用指令

第 6 章介绍了 Lua 的基础语法，本章将会讲解基于 LuaJIT 的 Lua-Nginx-Module，它作为 Nginx 的第三方开源软件，拥有十分丰富的功能，可以轻松完成高并发的业务需求。

注意：本书使用的 Lua-Nginx-Module 版本是 0.10.13。Nginx API for Lua 将被简称为 Lua API，而 Lua-Nginx-Module 则被简称为 Ngx_Lua。后面章节中涉及的 Lua API 大部分是包含参数的。如果参数以“?”结尾，代表这个参数是可选的，例如在指令 ngx.req.get_headers (max_headers?, raw?)中，max_headers 和 raw 是可选的。

7.1 Nginx 和 OpenResty

首先认识一下 OpenResty，它是一个基于 Nginx 和 Lua 开发的高性能的 Web 平台，包含大量成熟的第三方库，可快速搭建出高性能的 Web 服务器，支持常用的反向代理、网关系统、Web 应用等。

如果在 Nginx 上使用 Ngx_Lua，需要先进行编译；而 OpenResty 已经包含此模块，不需要再进行编译了。读者可以自由选择使用 Nginx 或 OpenResty 来搭建服务。如果无法抉择，可参考如下场景。

1. 使用 Nginx 编译 Ngx_Lua 的场景

HTTP 代理服务器：复杂度较小，只需部分组件即可，且代理服务器一般由运维人员进行维护。使用 Nginx 的稳定版进行编译，在性能方面会更有保障；而 OpenResty 则基于 Nginx 的主线版，可能会不定期更新。

2．OpenResty 的使用场景

API 服务：业务需求多，需要大量组件。

网关系统：需要大量组件和指令来实现动态组件功能。

Web 应用服务器：业务服务、页面服务等，如详情页业务的开发。

使用 Nginx 编写的 Lua 代码都可以直接迁移到 OpenResty 上，反之却不一定可行，毕竟 OpenResty 的组件更多。

7.2　安装 Ngx_Lua

请先安装 LuaJIT 2.1.0-beta3（详见 6.2 节），并需要编译 ngx_devel_kit 模块。

下面是在 Nginx 上的安装方式（OpenResty 自带此模块，不必安装和编译）：

```
# wget 'http://nginx.org/download/nginx-1.12.2.tar.gz'
# git clone https://github.com/simplresty/ngx_devel_kit.git
# git clone https://github.com/openresty/lua-nginx-module.git
# tar -xzvf nginx-1.12.2.tar.gz
# cd nginx-1.12.2/
# ./configure --prefix=/usr/local/nginx_1.12.2 \
     --add-module=../ngx_devel_kit \
     --add-module=../lua-nginx-module
     --with-ld-opt="-Wl,-rpath,$LUAJIT_LIB"

# make && make install
```

并不是每个 Nginx 版本都支持最新的 Ngx_Lua，目前已知支持最新 Ngx_Lua 的 Nginx 版本如下：

```
1.13.x (last tested: 1.13.6)
1.12.x
1.11.x (last tested: 1.11.2)
1.10.x
1.9.x (last tested: 1.9.15)
1.8.x
1.7.x (last tested: 1.7.10)
1.6.x
```

如果需要获取最新版本的支持信息，请参考 https://github.com/openresty/lua-nginx-module#nginx-compatibility。

7.3 牢记 Context

Ngx_Lua API 指令和 Nginx 的指令一样，都存在配置环境的约束问题，因此在使用过程中要确保指令的环境符合预期。

指令：ngx.var.VARIABLE

语法：ngx.var.VAR_NAME

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、ody_filter_by_lua*、log_by_lua*

Context 即配置环境，第一次接触 Ngx_Lua 的读者看到这样的配置环境可能会觉得难以理解，因为这还涉及 Ngx_Lua 的执行阶段（后面会有介绍）。

7.4 Hello World

首先，还是来一条经典语句“Hello, World!”，在 Nginx 配置中加入一个 server：

```
server {
    listen       80;
    server_name  testnginx.com;
    charset koi8-r;
    location = /test {
     #设置文件使用默认的 MIME-Type,将会增加一个 Content-Type:text/plain 的响应头
     default_type 'text/plain';
     -- content_by_lua_block 执行阶段
     content_by_lua_block {
         ngx.say('Hello,World!')
     }
    }
}
```

访问该 server，输出如下：

```
#  curl -I http://testnginx.com/test
Hello,World!
```

ngx.say 将数据作为响应体输出，返回给客户端，并在末尾加上一个回车符。

代码中用到了 content_by_lua_block 这个指令块，它的主要作用是在 HTTP 的内容处理阶段生成数据，详见 8.6 节。

7.5　避免 I/O 阻塞

当 Nginx 和 Lua 进行读取磁盘操作时会对 Nginx 的事件循环造成阻塞，所以在请求中应尽量避免操作磁盘，特别是当文件较大时更应如此。

如果 Lua 使用网络 I/O，为了避免出现阻塞的情况，请使用基于 Lua API 开发的指令，并使用子请求（将在 7.13 节介绍）来发送网络 I/O 和磁盘 I/O。如果需要频繁读取磁盘，请分离磁盘 I/O 的任务和网络 I/O 的任务，避免它们相互影响。

7.6　定义模块搜索路径

在开发过程中，常常需要编写自定义的模块，或者引入第三方的 Lua 或 C 模块。通过下面的配置可以定义相关模块的路径，以方便快速查找。

7.6.1　定义 Lua 模块的搜索路径

lua_package_path 用来设置默认的 Lua 模块的搜索路径，并配置在 http 阶段。它支持配置相对路径和绝对路径，其中相对路径是在 Nginx 启动时由-p PATH 决定的。如果在启动 Nginx 时没有配置-p PATH，就会使用编译时--prefix 的值，此值一般存放在 Nginx 的$prefix（也可以用${prefix}来表示）变量中。使用 lua_package_path 设置 Lua 模块搜索路径的示例如下：

```
http {
    -- lua_package_path 在配置中只能出现一次，使用下面的任何一种方法都可以
    lua_package_path "/usr/local/nginx_1.12.2/conf/lua_modules/?.lua;;";
    lua_package_path "conf/lua_modules/?.lua;;";
    lua_package_path "${prefix}conf/lua_modules/?.lua;;";
```

上述配置中的 3 种配置方式都指向同一个位置：第 1 种配置方式采用的是绝对路径；第 2 种配置方式采用的是相对路径，Nginx 编译时用 --prefix=/usr/local/nginx_1.12.2；第 3 种配置方式采用的也是相对路径，Nginx 编译时用--prefix=/usr/local/nginx_1.12.2 或-p PATH 指定的位置。

第 1 种配置方式的缺点在于写出了具体文件搜索路径，迁移代码时会比较麻烦。第 2 种配置方式的缺点在于无法和-p PATH 一起使用，如果-p 换了位置就会导致这个配置无效。对于第 3 种配置方式，如果-p 的位置换了，${prefix}的值会跟着变换，使用起来比较灵活。所以建议使用第 3 种配置方式来配置。

lua_package_path 可以支持设置多个搜索路径，多个搜索路径之间使用分号分隔就可以了，示例如下：

```
lua_package_path "${prefix}conf/lua_modules/?.lua;/opt/lua/?.lua;;";
```

注意：上述配置中搜索路径的最后出现了两个半角分号“;;”，代表的是 LuaJIT 安装时的原始搜索路径。如果在前面的搜索路径里面无法搜索到需要的模块，就会依次搜索后面的路径。

7.6.2 定义 C 模块的搜索路径

lua_package_cpath：用来设置 C 模块的搜索路径，并配置在 http 阶段。使用方式和 lua_package_path 一样，示例如下：

```
lua_package_cpath "${prefix}conf/c_md/?.so;/opt/c/?.so;;";
```

7.7 读/写 Nginx 的内置变量

第 2 章介绍过 Nginx 的内置变量，如果需要读取 Nginx 的内置变量可以使用 ngx.var.VARIABLE。

语法：ngx.var.VAR_NAME

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、og_by_lua*

含义：读/写 Nginx 的变量值。例如 HTTP 请求头、Nginx 创建的变量、URL 参数，甚至 Nginx 通过正则表达式捕获的$1、$2 等值（获取方式是 ngx.var[1]、ngx.var[2]，依此类推）。

示例如下：

```
server {
    listen       80;
    server_name  testnginx.com;
    location ~ ^/([a-z]+)/var.html {
        set $a '';
        set $b '';
        set $c '';
        set $d '';
        rewrite_by_lua_block {
           local ngx = require "ngx"
           -- 将 1 赋值给变量 a
           ngx.var.a = '1'
           -- 获取 HTTP 请求头中 User_Agent 的值并赋值给变量 b
           ngx.var.b = ngx.var.http_user_agent
           -- 获取参数 test 的值赋值给变量 c
```

```
            ngx.var.c = ngx.var.arg_test
            -- 获取 location 中正则表达式捕获的$1 的值并赋值给变量 d
            ngx.var.d = ngx.var[1]
        }
        echo $a;
        echo $b;
        echo $c;
        echo $d;
    }
```

执行结果如下：

```
# curl -i 'http://testnginx.com/nginx/var.html?test=12132&a=2&b=c&dd'
HTTP/1.1 200 OK
Server: nginx/1.12.2
Date: Thu, 07 Jun 2018 07:22:32 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: keep-alive
1
curl/7.19.7 (x86_64-redhat-linux-gnu) libcurl/7.19.7 NSS/3.19.1 Basic
ECC zlib/1.2.3 libidn/1.18 libssh2/1.4.2
12132
nginx
```

如果是未定义的 Nginx 变量，就无法直接在 Lua 中进行读取。况且有些变量只能读取，无法进行修改，如$query_string、$arg_PARAMETER 和$http_NAME。

7.8　控制请求头

在 4.1 节中讲了 Nginx 中控制请求头的指令，在 Lua API 中也有类似的指令。

7.8.1　添加请求头

指令：ngx.req.set_header

语法：ngx.req.set_header(header_name, header_value)

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*

含义：添加或修改当前 HTTP 的请求头。如果请求头已经存在，则会被替换成新的值。通过此方式设置的请求头会被继承到子请求中。

例如要设置一个名为 Test_Ngx_Ver、值为 1.12.2 的请求头，则代码如下：

```
ngx.req.set_header("Test_Ngx_Ver", "1.12.2")
```

ngx.req.set_header 支持给同一个请求头设置多个值，用数组的方式添加：

```
ngx.req.set_header("Test", {"1", "2"})
```

多个值的输出结果：

```
Test: 1
Test: 2
```

7.8.2 清除请求头

指令：ngx.req.clear_header

语法：ngx.req.clear_header(header_name)

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*

含义：清除当前请求中指定的请求头。清除后，如果存在未执行的子请求，则子请求会继承清除后的请求头。

示例如下：

```
ngx.req.clear_header("Test_Ngx_Ver")
```

还有一种清除请求头的方式：

```
ngx.req.set_header("Test_Ngx_Ver", nil)
```

7.8.3 获取请求头

指令：ngx.req.get_headers

语法：headers = ngx.req.get_headers(max_headers?, raw?)

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*

含义：获取当前请求的全部请求头，并返回一个 Lua 的 table 类型的数据。

示例如下：

```
server {
    listen       80;
    server_name  testnginx.com;
```

```
        location  / {

           content_by_lua_block {
              local ngx = require "ngx";
              local h = ngx.req.get_headers()
              for k, v in pairs(h) do
                  ngx.say('Header name: ',k,' value:',v)
              end
              -- 因为是 table，所以可以使用下面的方式读取单个响应头的值
              ngx.say(h["host"])
           }
        }
    }
```

输出结果如下：

```
    # curl -i 'http://testnginx.com/test?=12132&a=2&b=c&dd'
    HTTP/1.1 200 OK
    Server: nginx/1.12.2
    Date: Fri, 08 Jun 2018 07:46:38 GMT
    Content-Type: application/octet-stream
    Transfer-Encoding: chunked
    Connection: keep-alive

    Header name:host value: testnginx.com
    Header name:accept value: */*
    Header  name:user-agent  value:  curl/7.19.7  (x86_64-redhat-linux-gnu)
libcurl/7.19.7 NSS/3.19.1 Basic ECC zlib/1.2.3 libidn/1.18 libssh2/1.4.2
    testnginx.com
```

7.9　控制响应头

HTTP 响应头需要配置很多重要的信息，例如添加 CDN 缓存时间、操作 set-cookie、标记业务数据类型等。利用 Lua 的 API 可以轻松完成这些配置，并且它有丰富的模块可供选择。

7.9.1　获取响应头

指令：ngx.resp.get_headers

语法：headers = ngx.resp.get_headers(max_headers?, raw?)

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_

by_lua*、body_filter_by_lua*、log_by_lua*、balancer_by_lua*

含义：读取当前请求的响应头，并返回一个 Lua 的 table 类型的数据。

示例如下：

```
server {
    listen       80;
    server_name  testnginx.com;
     location  / {
        content_by_lua_block {
           local ngx = require "ngx";
           local h = ngx.resp.get_headers()
           for k, v in pairs(h) do
               ngx.say('Header name:',k,' value: ',v)
           end
           -- 因为是 table，所以可以使用下面的方式读取单个响应头的值
           ngx.say(h["content-type"])
        }
    }
}
```

执行结果如下：

```
# curl -i 'ttp://testnginx.com/test?=12132&a=2&b=c&dd'
HTTP/1.1 200 OK
Server: nginx/1.12.2
Date: Fri, 08 Jun 2018 07:36:35 GMT
Content-Type: application/octet-stream
Transfer-Encoding: chunked
Connection: keep-alive

Header name:content-type value: application/octet-stream
Header name:connection value: keep-alive
application/octet-stream
```

7.9.2 修改响应头

指令：ngx.header.HEADER

语法：ngx.header.HEADER = VALUE

语法：value = ngx.header.HEADER

环境：rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、og_by_lua*

含义：对响应头进行修改、清除、添加等操作。

此 API 在输出响应头时，默认会将“_”替换成“-”，示例如下：

```
server {
    listen       80;
    server_name  testnginx.com;

    location / {
        content_by_lua_block {
            local ngx = require "ngx"
            ngx.header.content_type = 'text/plain';
            -- 在代码里面是“_”，输出时就变成“-”了
            ngx.header.Test_Nginx = 'Lua';
            -- 下面的代码等同于 ngx.header.A_Ver = 'aaa'
            ngx.header["A_Ver"] = 'aaa';
            -- 读取响应头，并赋值给变量 a
            local a = ngx.header.Test_Nginx;
        }

    }
}
```

执行代码，“_”都被替换成了“-”，如下所示：

```
# curl -i 'http://testnginx.com/?test=12132&a=2&b=c&dd'
HTTP/1.1 200 OK
Server: nginx/1.12.2
Date: Fri, 08 Jun 2018 03:18:16 GMT
Content-Type: application/octet-stream
Transfer-Encoding: chunked
Connection: keep-alive
Test-Nginx: Lua
A-Ver: aaa
```

有时需要在一个响应头中存放多个值。例如，当访问/test 路径时，需要为 set-cookie 设置两个 Cookie：

```
location = /test {
    content_by_lua_block {
       local ngx = require "ngx"
       -- 以逗号分隔两个 Cookie
       ngx.header['Set-Cookie']   =   {'test1=1;   path=/test',   'test2=2;
path=/test'}
```

```
        }
    }
```

输出结果如下：

```
# curl -i 'http://testnginx.com/test?=12132&a=2&b=c&dd'
HTTP/1.1 200 OK
Server: nginx/1.12.2
Date: Fri, 08 Jun 2018 03:21:59 GMT
Content-Type: application/octet-stream
Transfer-Encoding: chunked
Connection: keep-alive
Set-Cookie: test1=1; path=/test
Set-Cookie: test2=2; path=/test
```

7.9.3 清除响应头

如需清除一个响应头，将它赋值为 nil 即可，如下所示：

```
ngx.header["X-Test"] = nil;
```

7.10 读取请求体

$request_body 表示请求体被读取到内存中的数据，一般由 proxy_pass、fastcgi_pass、uwsgi_pass 和 scgi_pass 等指令进行处理。由于 Nginx 默认不读取请求体的数据，因此当 Lua 通过 ngx.var.request_body 的方式获取请求体时会发现数据为空。那么，该如何获得请求体的数据呢？下面将介绍几种可行的方式。

7.10.1 强制获取请求体

指令：lua_need_request_body

语法：lua_need_request_body <on|off>

默认值：off

环境：http、server、location、location if

含义：默认为 off，即不读取请求体。如果设置为 on，则表示强制读取请求体，此时，可以通过 ngx.var.request_body 来获取请求体的数据。但需要注意一种情况，$request_body 存在于内存中，如果它的字节大小超过 Nginx 配置的 client_body_buffer_size 的值，Nginx 就会把请求体存放到临时文件中。此时数据就不在内存中了，这会导致$request_body 为空。所以需要设

置 client_body_buffer_size 和 client_max_body_size 的值相同，避免出现这种情况。

这种配置方式不够灵活，Ngx_Lua 官网也不推荐使用此方法。下面将介绍一种更合适的方式去获取请求体的数据。

7.10.2　用同步非阻塞方式获取请求体

指令：ngx.req.read_body

语法：ngx.req.read_body()

环境：rewrite_by_lua*、access_by_lua*、content_by_lua*

含义：同步读取客户端请求体，且不会阻塞 Nginx 的事件循环。使用此指令后，就可以通过 ngx.req.get_body_data 来获取请求体的数据了。但如果使用临时文件来存放请求体，就需要先使用函数 ngx.req.get_body_file 来获取临时文件名，再读取临时文件中的请求体数据。

指令：ngx.req.get_body_data

语法：data = ngx.req.get_body_data()

环境：rewrite_by_lua*、access_by_lua*、content_by_lua*、log_by_lua*

含义：执行 ngx.req.read_body 指令后，可以使用本指令在内存中获取请求体数据，结果会返回一个 Lua 的字符串类型的数据。如果要获取 Lua 的 table 类型的数据，则需要使用 ngx.req.get_post_args。

指令：ngx.req.get_post_args

语法： args, err = ngx.req.get_post_args(max_args?)

环境：rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*

含义：在执行 ngx.req.read_body 指令后，可以使用本指令读取包含当前请求在内的所有 POST 请求的查询参数，返回一个 Lua 的 table 类型的数据。max_args 参数的作用是限制参数的数量。为了服务的安全，最多支持使用 100 个参数（包括重复的参数），超过限制的参数会被忽略。如果 max_args 为 0，则表示关闭此限制；但为了避免被无穷多的参数攻击，不要将 max_args 设置为 0。如果最多支持使用 10 个参数，则应配置为 ngx.req.get_post_args(10)。

指令：ngx.req.get_body_file

语法：file_name = ngx.req.get_body_file()

环境：rewrite_by_lua*、access_by_lua*、content_by_lua*

含义：在执行 ngx.req.read_body 指令后，可以使用本指令获取存放请求体的临时文件名（绝对路径）。如果请求体被存放在内存中，获取的值就是 nil。通过本指令获取的文件是只读的，不可以被修改，且会在被 Nginx 读取后被删除。

7.10.3 使用场景示例

下面将对这些指令的使用方式和使用场景进行展示。

1. 获取 string 类型的请求体

要获取 string 类型的请求体，可以使用如下配置：

```
   server {
       listen       80;
       server_name  testnginx.com;
       location / {
         client_max_body_size 10k;
         client_body_buffer_size 1k;

         content_by_lua_block {
          local ngx = require "ngx"
            -- 开启读取请求体模式
            ngx.req.read_body()
            -- 获取内存中的请求体
            local data = ngx.req.get_body_data()
            if data then
                ngx.print('ngx.req.get_body_data: ',data, ' ---- type is ',
type(data))
                return
            else
            -- 如果没有获取到内存中的请求体数据，则到临时文件中读取
                local file = ngx.req.get_body_file()
                if file then
                    ngx.say("body is in file ", file)
                else
                    ngx.say("no body found")
                end
            end
         }
       }
```

配置好后，重载 Nginx 配置（重载是指使用 HUP 信号或 reload 命令来重新加载配置）。先

用一个小于 1KB 的请求体（在 Nginx 配置中设置 client_body_buffer_size 为 1k）执行请求，输出的是 string 字符串类型，如下所示：

```
# curl -i http://testnginx.com/ -d 'test=12132&a=2&b=c&dd'
HTTP/1.1 200 OK
Server: nginx/1.12.2
Date: Wed, 06 Jun 2018 11:03:35 GMT
Content-Type: application/octet-stream
Transfer-Encoding: chunked
Connection: keep-alive

ngx.req.get_body_data: test=12132&a=2&b=c&dd ---- type is string
```

2．获取 table 类型的请求体

要获取 table 类型的请求体，可以使用如下配置：

```
server {
    listen       80;
    server_name  testnginx.com;

    location / {
      client_max_body_size 10k;
      client_body_buffer_size 1k;

      content_by_lua_block {
         -- 开启读取请求体模式
         ngx.req.read_body()
         -- 获取内存中的请求体，返回的结果是 Lua 的 table 类型的数据
         local args, err = ngx.req.get_post_args()
         if args then
            for k, v in pairs(args) do
                if type(v) == "table" then
                    -- 如果存在相同的参数名，就会将相同的参数并列在一起，以逗号分隔
                    ngx.say(k, ": ", table.concat(v, ", "))
                else
                    ngx.say(k, ": ", v)
                end
             end
         else
             -- 如果没有获取到内存中的请求体数据，则到临时文件中读取
             local file = ngx.req.get_body_file()
             if file then
                 ngx.say("body is in file ", file)
             else
```

```
                    ngx.say("no body found")
                end
            end
        }
    }
}
```

发送测试请求，其中 a 参数有 2 个，c 参数值为空，d 参数连等号都没有。执行结果如下所示：

```
#  curl -i http://testnginx.com/ -d 'test=12132&a=2&b=c&dd=1&a=354&c=&d'

b: c
dd: 1
d: true
c:
test: 12132
a: 2, 354
```

可以看到参数 a 的两个值并列显示，并以逗号分隔；参数 c 显示为空；参数 d 的结果为布尔值 true。

3．获取临时文件中的请求体

如果使用一个大小在 1KB~10KB 之间的请求体，会发生什么呢？测试执行结果如下：

```
# curl -i http://testnginx.com/ -d 'test=12132&a=2&b=kls204120312saldkk12
easjdiasasd3ej12i3j12io3jeioq2jesksls204120312saldkk12easjdiasasd3ej12i3j12io
3jeioq2jesksls204120312saldkk12easjdiasasd3ej12i3j12io3jeioq2jesksls204120312s
aldkk12easjdiasasd3ej12i3j12io3jeioq2jesksls204120312saldkk12easjdiasasd3ej12
i3j12io3jeioq2jesksls204120312saldkk12easjdiasasd3ej12i3j12io3jeioq2je2041203
12saldkk12easjdiasasd3ej12i3j12io3jeioq2jesksls204120312saldkk12easjdiasasd3e
j12i3j12io3jeioq2jesksls204120312saldkk12easjdiasasd3ej12i3j12io3jeioq2jesksls
204120312saldkk12easjdiasasd3ej12i3j12io3jeioq2jesksls204120312saldkk12easjdi
asasd3ej12i3j12io3jeioq2jesksls204120312saldkk12easjdiasasd3ej12i3j12io3jeioq
2je204120312saldkk12easjdiasasd3ej12i3j12io3jeioq2jesksls204120312saldkk12eas
jdiasasd3ej12i3j12io3jeioq2jesksls204120312saldkk12easjdiasasd3ej12i3j12io3je
ioq2jesksls204120312saldkk12easjdiasasd3ej12i3j12io3jeioq2jesksls204120312sald
kk12easjdiasasd3ej12i3j12io3jeioq2jesksls204120312saldkk12easjdiasasd3ej12i3j
12io3jeioq2je204120312saldkk12easjdiasasd3ej12i3j12io3jeioq2jesksls204120312s
aldkk12easjdiasasd3ej12i3j12io3jeioq2jesksls204120312saldkk12easjdiasasd3ej12
i3j12io3jeioq2jesk20312saldkk12easjdiasasd3ej12i3j12io3jeioq2jesksls204120312
saldkk12easjdiasasd3ej12i3j12io3jeioq2jesksls204120312saldkk12easjdiasasd3ej1
2i3j12io3jeioq2je204120312saldkk12easjdiasasd3ej12i3j12io3jeioq2jesksls204120
312saldkk12easjdiasasd3ej11'
```

```
HTTP/1.1 100 Continue

HTTP/1.1 200 OK
Server: nginx/1.12.2
Date: Wed, 06 Jun 2018 10:14:32 GMT
Content-Type: application/octet-stream
Transfer-Encoding: chunked
Connection: keep-alive

body is in file /usr/local/nginx_1.12.2/client_body_temp/0000000051
```

因为请求体数据的大小大于 client_body_buffer_size 的值，所以使用了临时文件存储请求体的数据。因此，需要先获取存放数据的临时文件名，再读取请求体数据。

注意：读取临时文件中的请求体数据是不被推荐的，因此本书不对相关操作进行讲解，有兴趣的读者可以使用 io.open 完成读取。

7.10.4　使用建议

在实际应用中，关于读取请求体，有如下几条建议。

- 尽量不要使用 lua_need_request_body 获取请求体。
- 获取请求体前，必须执行 ngx.req.read_body()。
- 获取请求体数据时尽量不要使用硬盘上的临时文件，否则会对性能有很大影响；务必确认请求体数据的字节大小，并确保 client_body_buffer_size 和 client_max_body_size 的值一致，这样只需到内存中读取数据就可以了。它既提高了 Nginx 自身的吞吐能力，也提升了 Lua 的读取性能。
- 如果请求体存放在临时文件中，Nginx 会在处理完请求后自动清理临时文件。
- 使用 ngx.req.get_post_args 可以对请求的参数进行灵活控制，但不能关闭限制，以避免被恶意攻击。

7.11　输出响应体

在 Lua 中，响应体的输出可以使用 ngx.print 和 ngx.say 这两个指令完成。

7.11.1　异步发送响应体

指令：ngx.print

语法：ok, err = ngx.print(...)

环境：rewrite_by_lua*、access_by_lua*、content_by_lua*

含义：用来输出内容，输出的内容会和其他的输出合并，然后发送给客户端。如果响应头还未发送，发送前会优先将响应头发送出去。

示例如下：

```
location  / {

    content_by_lua_block {
        local ngx = require "ngx";
        local h = ngx.req.get_headers()
        for k, v in pairs(h) do
            ngx.print('Header name: ',k,' value: ',v)
        end

    }
}
```

执行结果如下（所有的数据会合并到一起进行发送）：

```
# curl -i 'http://testnginx.com/test?=12132&a=2&b=c&dd'
HTTP/1.1 200 OK
Server: nginx/1.12.2
Date: Fri, 08 Jun 2018 08:11:40 GMT
Content-Type: application/octet-stream
Transfer-Encoding: chunked
Connection: keep-alive

Header name:host value: testnginx.comHeader name:accept value: */*Header name:user-agent value: curl/7.19.7 (x86_64-redhat-linux-gnu) libcurl/7.19.7 NSS/3.19.1 Basic ECC zlib/1.2.3 libidn/1.18 libssh2/1.4.
```

指令：ngx.say

语法：ok, err = ngx.say(...)

环境：rewrite_by_lua*、access_by_lua*、content_by_lua*

含义：其功能和 ngx.print 一样，只是在输出结果中多了 1 个回车符。

7.11.2 同步发送响应体

ngx.print 和 ngx.say 为异步调用，执行后并不会立即输出响应体，可以通过执行 ngx.flush(true)来实现同步输出响应体的功能。

指令：ngx.flush

语法：ok, err = ngx.flush(wait?)

环境：rewrite_by_lua*、access_by_lua*、content_by_lua*

含义：在默认情况下会发起一个异步调用，即不等后续的数据到达缓冲区就会直接将内容输出到客户端。如果将 wait 的参数值设置为 true，则表示同步执行，即会等内容全部输出到缓冲区后再输出到客户端。

```
server {
    listen       80;
    server_name  testnginx.com;
    default_type 'text/plain';
    location /test1 {
        content_by_lua_block {
           ngx.say("test ")
           ngx.say("nginx ")
           ngx.sleep(3)
           ngx.say("ok!")
           ngx.say("666!")
        }
    }

    location /test2 {
        content_by_lua_block {
           ngx.say("test ")
           ngx.say("nginx ")
           ngx.flush(true)
           ngx.sleep(3)
           ngx.say("ok!")
           ngx.say("666!")
        }
    }

}
```

访问/test1 和 /test2 后，从执行结果可以看出，带有 ngx.flush(true) 指令的内容会先输出“test nginx”，然后，等待大约 3s 后再输出“ok! 666!”。如果没有配置 ngx.flush(true)指令，请求会在等待 3s 后输出完整的一句话。

注意：指令 ngx.flush 不支持 HTTP 1.0，可以使用如下方式进行测试。

```
# curl -i 'http://testnginx.com/test2'  --http1.0
```

7.12 正则表达式

虽然 Lua 支持正则匹配且功能齐全，但在 Nginx 上推荐使用 Ngx-Lua 提供的指令。

7.12.1 单一捕获

指令：ngx.re.match

语法：captures, err = ngx.re.match(subject, regex, options?, ctx?, res_table?)

环境：init_worker_by_lua*、set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、balancer_by_lua*、ssl_certificate_by_lua*、ssl_session_fetch_by_lua*、ssl_session_store_by_lua*

含义：使用 Perl 兼容的正则表达式来匹配 subject 参数，只返回匹配到的第一个结果。如果匹配失败，则返回 nil；如果有异常，则返回 nil 和一个描述错误信息的 err。

示例如下：

```
location / {
    content_by_lua_block {
        local ngx = require "ngx";
        -- 匹配多个数字+aaa 的正则表达式
    local m, err = ngx.re.match(ngx.var.uri, "([0-9]+)(aaa)");
        if m then
           -- 匹配成功后输出的信息
           ngx.say(ngx.var.uri,'---match success---','its type:',type(m))
           ngx.say(ngx.var.uri,'---m[0]---', m[0])
           ngx.say(ngx.var.uri,'---m[1]---', m[1])
           ngx.say(ngx.var.uri,'---m[2]---', m[2])
        else
           if err then
               ngx.log(ngx.ERR, "error: ", err)
               return
           end
           ngx.say("match not found")
        end

    }
}
```

执行结果如下：

```
# curl  'http://testnginx.com/test/a123aaa/b456aaa/c'
/test/a123aaa/b456aaa/c---match success---its type: table
/test/a123aaa/b456aaa/c---m[0]---123aaa
/test/a123aaa/b456aaa/c---m[1]---123
/test/a123aaa/b456aaa/c---m[2]---aaa
```

从执行结果可以看出以下几点。

1．ngx.re.match 只返回匹配到的第一个结果，所以后面的 456aaa 并没有被输出。

2．ngx.re.match 返回的结果是 table 类型的。

3．ngx.re.match 匹配成功后，m[0] 的值是匹配到的完整数据，而 m[1]、m[2] 是被包含在括号内的单个匹配结果。

7.12.2　全部捕获

ngx.re.match 只返回第一次匹配成功的数据。如果想获取所有符合正则表达式的数据，可以使用 ngx.re.gmatch。

指令：ngx.re.gmatch

语法：iterator, err = ngx.re.gmatch(subject, regex, options?)

环境：init_worker_by_lua* 、set_by_lua* 、rewrite_by_lua* 、access_by_lua 、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*，ngx.timer.*、balancer_by_lua*、ssl_certificate_by_lua*、ssl_session_fetch_by_lua*、ssl_session_store_by_lua*

含义：和 ngx.re.match 功能相似，但返回的是一个 Lua 迭代器，可以通过迭代的方式获取匹配到的全部数据。

```
location / {
   content_by_lua_block {
      local ngx = require "ngx";
      -- 参数 i 表示忽略大小写
      local m_table, err = ngx.re.gmatch(ngx.var.uri, "([0-9]+)(aaa)",
"i");
      if not m_table then
          ngx.log(ngx.ERR,  err)
          return
      end
      while true do
         local m, err = m_table()
         if err then
            ngx.log(ngx.ERR,  err)
```

```
            return
        end
        if not m then
            break
        end
        ngx.say(m[0])
        ngx.say(m[1])
    end

  }
}
```

执行结果如下：

```
# curl  'http://testnginx.com/test/a123aaa/b456AAA/c'
123aaa
123
456AAA
456
```

ngx.re.match 和 ngx.re.gmatch 都有一个 options 参数，用来控制匹配的执行方式，options 常用参数说明见表 7-1。

表 7-1　options 常用参数说明

参　数	说　明
i	忽略大小写
o	仅编译一次，启用 worker 进程级别的正则表达式进行缓存
m	多行模式（类似于 Perl 的/m 修饰符）
j	启用 PCRE JIT 编译，要求 PCRE（Perl Compatible Regular Expressions，一个用 C 语言编写的正则表达式函数库）版本在 8.21 以上，且必须在 Nginx 编译时指定--enable-jit 参数。为了使性能更佳，此参数应与参数 o 一起使用

7.12.3　更高效的匹配和捕获

ngx.re.match 和 ngx.re.gmatch 在使用过程中都会生成 Lua table，如果只需确认正则表达式是否可以匹配成功，推荐使用如下指令。

指令：ngx.re.find

语法：from, to, err = ngx.re.find(subject, regex, options?, ctx?, nth?)

环境：init_worker_by_lua*、set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、balancer_by_

lua*、ssl_certificate_by_lua*、ssl_session_fetch_by _lua*、ssl_session_store_by_lua*

含义：与 ngx.re.match 类似，但只返回匹配结果的开始位置索引和结束位置索引。

因为 ngx.re.find 不会创建 table 来存储数据，所以性能上比 ngx.re.match 和 ngx.re.gmatch 要好很多。此时，如需捕获匹配到的数据，可以使用 Lua 的函数 string.sub。

```
   location / {
      content_by_lua_block {
         local ngx = require "ngx";
         local uri = ngx.var.uri
         -- 使用 o、j 两个参数进行匹配，以提升性能
         local    find_begin,find_end,err    =    ngx.re.find(uri,    "([0-
9]+)(aaa)","oj");
         if find_begin then
             ngx.say('begin: ',find_begin)
             ngx.say('end: ',find_end)
          -- 利用 Lua 的 string.sub 函数来获取数据
             ngx.say('find it: ' ,string.sub(uri, find_begin,find_end))
             return
         end
       }
   }
```

执行结果如下：

```
   # curl  'http://testnginx.com/test/a123aaa/b456AAAa/c'
   begin:8
   end:13
   find it: 123aaa
```

ngx.re.match、ngx.re.gmatch 和 ngx.re.find 都支持 ctx 参数，有关 ctx 参数的说明如下。

- ctx 是 Lua table 类型的，是可选的第 4 个参数，但若用到第 5 个参数 nth，那么，此位置需要用 nil 作为占位符。
- 当 ctx 有值（键是 pos，如 pos=1）时，ngx.re.find 将从 pos 位置开始进行匹配（位置的下标从 1 开始）。
- 无论 ctx 表中是否有值，ngx.re.find 都会在正则表达式匹配成功后，将 ctx 值设置为所匹配字符串之后的位置；若匹配失败，ctx 表将保持原有的状态。

nth 是 ngx.re.find 的第 5 个参数，是在 Lua-Nginx-Module 0.9.3 版本之后新增加的参数，它的作用和 ngx.re.match 中的 m[1]、m[2]类似。当 nth 等于 1 时，获取的结果等同于 ngx.re.match 中的 m[1]，示例如下：

```
   location / {
      content_by_lua_block {
         local ngx = require "ngx";
         local uri = ngx.var.uri

         -- 从 uri 位置为 10 的地方开始进行匹配，下标默认从 1 开始
         -- 只匹配 nth 是 1 的数据，即([0-9]+)的值
         local ctx = { pos = 10 }
         local    find_begin,find_end,err    =    ngx.re.find(uri,    "([0-
9]+)(aaa)","oji",ctx,1);
         if find_begin then
             ngx.say('begin: ',find_begin)
             ngx.say('end: ',find_end)
             ngx.say('find it: ' ,string.sub(uri, find_begin,find_end))
             return
         end
       }
   }
```

执行结果如下：

```
   # curl  'http://testnginx.com/test/a123aaa/b456AAAa/c'
   begin:10
   end:10
   find it: 3
```

因为 ctx 的位置是 10，所以 uri 前面的“/test/a12”这 9 个字符被忽略了，匹配到的只有 3aaa，又因为 nth 为 1，所以捕获到的值是 3。

7.12.4 替换数据

Lua API 也支持匹配对应数据并对其进行替换的指令。

指令：ngx.re.sub

语法：newstr, n, err = ngx.re.sub(subject, regex, replace, options?)

环境：init_worker_by_lua*、set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、balancer_by_lua*、ssl_certificate_by_lua*、ssl_session_fetch_by_ lua*、ssl_session_store_by_lua*

含义：若 subject 中含有参数 regex 的值，则将之替换为参数 replace 的值。options 为可选参数。替换后的内容将赋值给 newstr，n 表示匹配到的次数。

示例如下：

```
location / {
    content_by_lua_block {
        local ngx = require "ngx";
        local uri = ngx.var.uri
        local n_str, n, err = ngx.re.sub(uri,"([0-9]+)",'zzzz')
        if n_str then
            ngx.say(uri)
            ngx.say(n_str)
            ngx.say(n)
        else
            ngx.log(ngx.ERR, "error: ", err)
            return
        end
    }
}
```

执行结果如下：

```
# curl   'http://testnginx.com/test188/x2/1231'
/test188/x2/1231
/testzzzz/x2/1231
1
```

从结果可以看出，只在第一次匹配成功时进行了替换操作，并且只替换了 1 次，所以 n 的结果是 1。如果要替换匹配到的全部结果可以使用 ngx.re.gsub，示例如下：

```
local n_str, n, err = ngx.re.gsub(uri,"([0-9]+)",'zzzz')
```

从执行结果可知，替换了 3 次，如下所示：

```
# curl   'http://testnginx.com/test188/x2/1231'
/test188/x2/1231
/testzzzz/xzzzz/zzzz
3
```

7.12.5　转义符号

正则表达式包括\d、\s、\w 等匹配方式，但在 Ngx_Lua 中使用时，反斜线 \ 会被 Lua 处理掉，从而导致匹配异常。所以需要对带有 \ 的字符进行转义，转义方式和其他语言有些区别，转义后的格式为\\\\d、\\\\s、\\\\w，因为反斜线会被 Nginx 和 Lua 各处理一次，所以\\\\会先变成\\，再变成\。

还可以通过[[]]的方式将正则表达式直接传入匹配指令中，以避免被转义，如下所示：

```
local find_regex = [[\d+]]
local m = ngx.re.match("xxx,43", find_regex)
ngx.say(m[0])    -- 输出 43
```

通常建议使用[[]]的方式。

7.13 子请求

Nginx 一般分两种请求类型，一种是主请求；一种是子请求，即 subrequest。主请求从 Nginx 的外部进行访问，而子请求则在 Nginx 内部进行访问。子请求不是 HTTP 请求，不会增加网络开销。它的主要作用是将一个主请求分解为多个子请求，用子请求去访问指定的 location 服务，最后汇总到一起完成主请求的任务。

Nginx 的请求方法有很多种，如 GET、POST、 PUT 、DELETE 等，同样，子请求也支持这些请求方法。

7.13.1 请求方法

Lua API 中提供了多个指令来实现子请求，Lua API 常见的请求方法说明见表 7-2。

表 7-2 Lua API 常见的请求方法说明

Nginx 的请求方法	Lua API 中的请求方法	说　明
GET	ngx.HTTP_GET	GET 请求，主要用于获取数据
HEAD	ngx.HTTP_HEAD	和 GET 请求一样，但是没有响应体
PUT	ngx.HTTP_PUT	向服务器传送数据并替换指定文档的内容
DELETE	ngx.HTTP_DELETE	删除指定的资源
POST	ngx.HTTP_POST	将请求体提交到服务器上
OPTIONS	ngx.HTTP_OPTIONS	允许客户端请求查看服务器的性能

7.13.2 单一子请求

指令：ngx.location.capture

语法：res = ngx.location.capture(uri, options?)

环境：rewrite_by_lua*、access_by_lua*、content_by_lua*

含义：发出同步但不阻塞 Nginx 的子请求。可以用来访问指定的 location，但不支持访问命名 location（如@abc 就是命名 location）。location 中可以有静态文件，如 ngx_proxy、

ngx_fastcgi、ngx_memc、ngx_postgres、ngx_drizzle，甚至 Ngx_Lua 和 Nginx 的 c 模块。

子请求总是会把整个请求体缓存到内存中，如果要处理一个较大的子请求，使用 cosockets 是最好的选择（cosockets 是与 ngx.socket.tcp 有关的 API）。

子请求一般在内部进行访问，建议在被子请求访问的 location 上配置 internal，即只允许内部访问。

子请求返回的结果 res，它是一个 table 类型的数据，包含 4 个元素：res.status、res.header、res.body 和 res.truncated。res 的元素名及其用途见表 7-3。

表 7-3　res 的元素名及其用途

元 素 名	用　途
res.status	存储子请求的 HTTP 返回状态
res.header	存储子请求返回的所有响应头，table 类型
res.body	保存子请求的响应体，数据有可能会被截断，table 类型
res.truncated	记录请求是否被截断，如果出现截断，一般是由于客户端过早断开连接或子请求超时导致的，布尔类型

ngx.location.capture 的第 2 个参数 options 是可选参数，也可以包含多个参数，示例如下：

```
server {
    listen       80;
    server_name  testnginx.com;
    default_type 'text/plain';

    location = /main {
        set $m 'hello';
        content_by_lua_block {
            local ngx = require "ngx";
            -- 发起子请求，访问/test，请求方式是 GET，请求体是 test nginx
            -- 子请求的 URL 参数是 a=1&b=2，并使用 copy_all_vars
            -- 将主请求的 Nginx 变量（$m）全部复制到子请求中
              local res = ngx.location.capture(
              '/test',  { method = ngx.HTTP_GET , body = 'test nginx',
               args = { a = 1, b = 2 },copy_all_vars = true }
            )
            ngx.say(res.status)
            ngx.say(res.body)
            ngx.say(type(res.header))
            ngx.say(type(res.truncated))
        }
    }
```

```
        location = /test
    {
        # 只能在 Nginx 内部进行访问
            internal;
            content_by_lua_block {
                local ngx = require "ngx";
                -- 获取请求体，在这里是获取主请求的请求体
                ngx.req.read_body()
                local body_args = ngx.req.get_body_data()
                -- 输出请求的参数，获取主请求的 m 变量的值，并与 world 进行字符串拼接
                ngx.print('request_body: ' ,body_args, ' capture_args: ',
ngx.var.args, '--- copy_all_vars : ', ngx.var.m .. 'world!')
            }
        }
    }
```

执行结果如下：

```
    # curl   'http://testnginx.com/main'
    200
    request_body:test   nginx   capture_args:a=1&b=2---      copy_all_vars   :
helloworld!
    table
    boolean
```

从示例中可以看出如下几点。

- ngx.location.capture 的第 2 个参数 options 可以包含多个 table 类型的参数。
- 子请求的请求方法由参数 method 进行配置，示例中的请求方法为 GET。
- 子请求通过参数 body 可以定义新的请求体。
- 子请求通过参数 args 可以配置新的 URL 的 args，args 是 table 类型的。
- copy_all_vars = true 的作用是将主请求的全部变量传递给子请求，如果没有此配置就不会传递过去。
- 从子请求的返回结果可以获取状态码、响应体、响应头、结果是否被截断。

根据上面的介绍可知，下面两种方式是等价的：

```
    local res = ngx.location.capture('/test?a=1&b=2')
    local res = ngx.location.capture('/test , args = { a = 1, b = '2' }')
```

ngx.location.capture 还支持更丰富的参数操作，具体如下。

- vars 参数，table 类型，可以设置子请求中的变量值，前提是该变量在 Nginx 中被声明

过。如果配置 copy_all_vars = true，且 vars 里有和主请求相同的变量，则会使用 vars 中变量的值；如果 vars 里是新变量，就会和主请求的变量一起传递过去。

- share_all_vars 参数，用来共享主请求和子请求的变量，如果在子请求中修改了共享变量的值，主请求的变量值也会被改变。不推荐使用此参数，因为可能会导致很多意外问题的出现。
- always_forward_body 参数，默认值为 false，此时，如果不设置 body 参数，且请求方法是 PUT 或 POST，则主请求的请求体可以传给子请求。如果把 always_forward_body 设置为 true，且不设置 body 参数，无论请求方法是什么，主请求的请求体都会传递给子请求。
- ctx 参数，指定一个 table 作为子请求的 ngx.ctx 表，它可以使主请求和子请求共享请求头的上下文环境。

关于参数 vars 的使用方式，示例如下：

```
location = /main {
    set $m 'hello';
    set $mm '';
    content_by_lua_block {
        local ngx = require "ngx";
        local res = ngx.location.capture(
            '/test',
            { method = ngx.HTTP_POST ,
            vars = {mm = 'MMMMM',m = 'hhhh'}}
        )
        ngx.say(res.body)
    }
}
location = /test {
    content_by_lua_block {
        local ngx = require "ngx";
        ngx.print(ngx.var.m .. ngx.var.mm )
    }
}
```

执行结果如下：

```
# curl   'http://testnginx.com/main'
hhhhMMMMM
```

主请求的变量在子请求中被修改了，并传给了子请求指定的/test。

注意：使用 ngx.location.capture 发送子请求时，默认会将主请求的请求头全部传入子请求中，这可能会带来一些不必要的麻烦。例如，如果浏览器发送的压缩头 Accept-Encoding:gzip 被传入子请求中，且子请求是 ngx_proxy 的标准模块，则请求的结果会被压缩后再返回，导致 Lua 无法读取子请求返回的数据。因此应将子请求的 proxy_pass_request_headers 设置为 off，避免把请求头传递给后端服务器。

7.13.3 并发子请求

有时需要发送多条子请求去获取信息，这时，就要用到并发操作了。

指令：ngx.location.capture_multi

语法：res1, res2, ... = ngx.location.capture_multi({ {uri, options?}, {uri, options?}, ... })

环境：rewrite_by_lua*、access_by_lua*、content_by_lua*

含义：与 ngx.location.capture 相似，但可以支持多个子请求并行访问，并按配置顺序返回数据。返回的数据也是多个结果集。

示例如下：

```
   server {
       listen       80;
       server_name  testnginx.com;
       default_type 'text/plain';
       location = /main {
           set $m 'hello';
           set $mm '';
           content_by_lua_block {
               local ngx = require "ngx";
               -- 发送两个子请求，会返回两个结果集
               local res1, res2 = ngx.location.capture_multi{
                   { "/test1?a=1&b=2" },
                   {  "/test2",{  method  =  ngx.HTTP_POST},body  =  "test
nginx" },
               }
               -- 返回的 body 的方式和 ngx.location.capture 一样
               if res1.status == ngx.HTTP_OK then
                   ngx.say(res1.body)
               end

               if res2.status == ngx.HTTP_OK then
                   ngx.say(res2.body)
```

```
            end
        }
    }
    location = /test1 {
        echo 'test1';
    }
    location = /test2 {
        echo 'test2';
    }

}
```

执行结果如下：

```
# curl   'http://testnginx.com/main'
test1

test2
```

主请求需要等到所有的子请求都返回后才会结束子请求的执行，最慢的子请求的执行时间就是整体的消耗时间，所以在实际业务中需要对子请求的超时时间做好限制。

注意：Nginx 对子请求有并发数量限制，目前 Nginx 1.1 以上的版本限制子请求并发数量为 200 个，老版本是 50 个。

7.14　获取 Nginx 的环境变量

通过 Lua API 可以获取 Nginx 的环境变量，用来提升某些业务处理流程，例如有些定时任务只需要在一个 worker 进程上执行，不需要执行多次，因此可以获取环境变量中 worker 的 ID，在指定的 ID 上执行任务即可；或者获取 Nginx 的 worker 进程是否正在 shutdown，以决定是否对数据进行备份操作。

7.14.1　获取环境所在的模块

指令：ngx.config.subsystem

语法：subsystem = ngx.config.subsystem

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、init_by_lua*、init_worker_by_lua*

含义：获取当前请求的 Nginx 子环境（http 或 stream）。如果在 http 模块下，就返回字符

串 http；如果在 stream 模块下，则返回字符串 stream。

7.14.2 确认调试模式

指令：ngx.config.debug

语法：debug = ngx.config.debug

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、init_by_lua*、init_worker_by_lua*

含义：判断请求是否在 Debug 模式下执行。例如，当需要在 Debug 模式下打印某些数据或执行某些代码时，可以通过这个判断，区分线下测试环境和线上环境。

7.14.3 获取 prefix 路径

指令：ngx.config.prefix

语法：prefix = ngx.config.prefix()

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、init_by_lua*、init_worker_by_lua*

含义：获取编译 Nginx 时--prefix=的路径，如果启动 Nginx 时使用了参数-p，就以参数-p 的值为准。

7.14.4 获取 Nginx 的版本号

指令：ngx.config.nginx_version

语法：ver = ngx.config.nginx_version

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、init_by_lua*、init_worker_by_lua*

含义：获取 Nginx 的版本号，如本书使用的 Nginx 版本号是 1.12.2。

7.14.5 获取 configure 信息

指令：ngx.config.nginx_configure

语法：str = ngx.config.nginx_configure()

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_

by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、init_by_lua*

含义：获取编译 Nginx 时./configure 命令的信息，返回的是一个字符串。

7.14.6　获取 Ngx_Lua 的版本号

指令：ngx.config.ngx_lua_version

语法：ver = ngx.config.ngx_lua_version

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、init_by_lua*

含义：获取 Ngx_Lua 模块的版本号。可以用来检查 Ngx_Lua 的版本。例如，当开发某个功能需要使用指定的版本时，可以在代码中进行判断，如果不是指定的版本，可以输出警告信息。

7.14.7　判断 worker 进程是否退出

指令：ngx.worker.exiting

语法：exiting = ngx.worker.exiting()

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、init_by_lua*、init_worker_by_lua*

含义：判断 Nginx 的 worker 进程是否退出。

7.14.8　获取 worker 进程的 ID

指令：ngx.worker.id

语法：count = ngx.worker.id()

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、init_by_lua*

含义：获取当前执行的 worker 进程的 ID。worker 进程的 ID 从 0 开始，依次递增，最大值是 worker 总数的值减 1。

7.14.9　获取 worker 进程的数量

指令：ngx.worker.count

语法：count = ngx.worker.count()

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、init_by_lua*、init_worker_by_lua*

含义：获取当前 Nginx worker 进程的数量，即 Nginx 配置中 worker_processes 的值。

7.15 定时任务

可以使用 Nginx 执行定时任务，例如，定期获取 MySQL 数据库中的数据并存放到共享内存中，定时监听某个配置是否发生了改变（如果发生改变就重载 Nginx），定时将日志远程传输到集中存储上等。

在 Lua-Nginx_Module 0.10.9 版本之前，常使用 ngx.timer.at 来启动定时任务。Lua-Nginx_Module 0.10.9 新增了 ngx.timer.every，启动定时任务更加方便了。本章中的定时任务都使用 ngx.timer.every 来创建，后续介绍也会以此命令为主。

7.15.1 创建定时任务

指令：ngx.timer.every

语法：hdl, err = ngx.timer.every(delay, callback, user_arg1, user_arg2, ...)

环境：init_worker_by_lua*、set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、balancer_by_lua*、ssl_certificate_by_lua*、ssl_session_fetch_by_lua*、ssl_session_store_by_lua*

含义：创建一个定时任务，delay 指延迟时间，表示每隔多久执行一次，支持配置 0.001s，不支持配置 0s；callback 是需要执行的 Lua 函数，当 Nginx 退出时，定时任务会被关闭。

```
   init_worker_by_lua_block {
       local delay = 3;
       local ngx = require "ngx";
       local check
       check = function(premature)
           if not premature then
            -- 输出当前 worker 进程的 PID 和 ID
                ngx.log(ngx.ERR,   '   ngx.worker.pid:   ',ngx.worker.pid(),'
ngx.worker.id: ',ngx.worker.id(),"------test nginx !!!")
           end
       end
       -- 每隔 3s 执行一次 check 函数
```

```
        local ok, err = ngx.timer.every(delay, check)
        if not ok then
             ngx.log(ngx.ERR, "failed to create timer: ", err)
             return
        end
    }
```

重载 Nginx 配置后，定时任务会在启动 worker 进程时就被触发执行，请观察图 7-1 所示的定时任务输出的日志。

```
[error] 22118#22118: *1240 [lua] init_worker_by_lua:7:  ngx.worker.pid: 22118 ngx.worker.id: 0------test nginx !!!, context: ngx.timer
[error] 22119#22119: *1241 [lua] init_worker_by_lua:7:  ngx.worker.pid: 22119 ngx.worker.id: 1------test nginx !!!, context: ngx.timer
[error] 22120#22120: *1242 [lua] init_worker_by_lua:7:  ngx.worker.pid: 22120 ngx.worker.id: 2------test nginx !!!, context: ngx.timer
[error] 22119#22119: *1243 [lua] init_worker_by_lua:7:  ngx.worker.pid: 22119 ngx.worker.id: 1------test nginx !!!, context: ngx.timer
[error] 22120#22120: *1244 [lua] init_worker_by_lua:7:  ngx.worker.pid: 22120 ngx.worker.id: 2------test nginx !!!, context: ngx.timer
[error] 22118#22118: *1245 [lua] init_worker_by_lua:7:  ngx.worker.pid: 22118 ngx.worker.id: 0------test nginx !!!, context: ngx.timer
[error] 22119#22119: *1246 [lua] init_worker_by_lua:7:  ngx.worker.pid: 22119 ngx.worker.id: 1------test nginx !!!, context: ngx.timer
[error] 22120#22120: *1247 [lua] init_worker_by_lua:7:  ngx.worker.pid: 22120 ngx.worker.id: 2------test nginx !!!, context: ngx.timer
[error] 22118#22118: *1248 [lua] init_worker_by_lua:7:  ngx.worker.pid: 22118 ngx.worker.id: 0------test nginx !!!, context: ngx.timer
[error] 22119#22119: *1249 [lua] init_worker_by_lua:7:  ngx.worker.pid: 22119 ngx.worker.id: 1------test nginx !!!, context: ngx.timer
[error] 22120#22120: *1250 [lua] init_worker_by_lua:7:  ngx.worker.pid: 22120 ngx.worker.id: 2------test nginx !!!, context: ngx.timer
[error] 22118#22118: *1251 [lua] init_worker_by_lua:7:  ngx.worker.pid: 22118 ngx.worker.id: 0------test nginx !!!, context: ngx.timer
[error] 22119#22119: *1252 [lua] init_worker_by_lua:7:  ngx.worker.pid: 22119 ngx.worker.id: 1------test nginx !!!, context: ngx.timer
[error] 22120#22120: *1253 [lua] init_worker_by_lua:7:  ngx.worker.pid: 22120 ngx.worker.id: 2------test nginx !!!, context: ngx.timer
[error] 22118#22118: *1254 [lua] init_worker_by_lua:7:  ngx.worker.pid: 22118 ngx.worker.id: 0------test nginx !!!, context: ngx.timer
[error] 22119#22119: *1255 [lua] init_worker_by_lua:7:  ngx.worker.pid: 22119 ngx.worker.id: 1------test nginx !!!, context: ngx.timer
[error] 22120#22120: *1256 [lua] init_worker_by_lua:7:  ngx.worker.pid: 22120 ngx.worker.id: 2------test nginx !!!, context: ngx.timer
[error] 22118#22118: *1257 [lua] init_worker_by_lua:7:  ngx.worker.pid: 22118 ngx.worker.id: 0------test nginx !!!, context: ngx.timer
[error] 22120#22120: *1258 [lua] init_worker_by_lua:7:  ngx.worker.pid: 22120 ngx.worker.id: 2------test nginx !!!, context: ngx.timer
```

图 7-1　定时任务输出的日志

从图 7-1 可以发现如下规则。

- 每个 worker 进程都在执行输出操作。
- 都是每 3s 执行一次。
- 如果没有从 Nginx 外部进行访问的请求，定时任务会继续执行下去。

参数 user_arg1、user_arg2 用来给定时任务传递参数，配置示例如下：

```
    init_worker_by_lua_block {
        local delay = 3;
        local ngx = require "ngx";
        local check
        -- 新增一个 u_arg1 参数，是对下面定时任务的'test nginx'进行填充
        check = function(premature,u_arg1)
            if not premature then
                ngx.log(ngx.ERR,    '    ngx.worker.pid:    ',ngx.worker.pid(),'
ngx.worker.id: ',ngx.worker.id(),'------', u_arg1)
            end
        end
```

```
        -- 新增参数'test nginx'
        local ok, err = ngx.timer.every(delay, check,'test nginx')
        if not ok then
            ngx.log(ngx.ERR, "failed to create timer: ", err)
            return
        end
    }
```

7.15.2 性能优化

在前面的配置示例中，Nginx 启动了 3 个 worker 进程，所以每 3s 会执行 3 次 worker 进程。但有时只需执行一次即可，例如当向共享内存中存放数据时，因为数据是所有 worker 进程共享的，所以执行一次就足够了。且被启动的 worker 进程越多，后端的并发就越多，这会增加后端服务器的负载压力，那么应该怎么减少 worker 进程重复执行的次数呢？

既然根据输出的日志可以获得每个 worker 进程的 ID，那么，只需利用 ID 指定一个 worker 进程来执行定时任务就可以了，示例如下：

```
    init_worker_by_lua_block {
         local delay = 3;
         local ngx = require "ngx";
         local check
         check = function(premature)
             if not premature then
                 ngx.log(ngx.ERR,   '   ngx.worker.pid:   ',ngx.worker.pid(),'
ngx.worker.id: ',ngx.worker.id(),"------test nginx !!!")
             end
         end

        -- 如果 worker 进程的 ID 为 0 就执行定时任务
        if 0 == ngx.worker.id() then
             local ok, err = ngx.timer.every(delay, check)
             if not ok then
                 ngx.log(ngx.ERR, "failed to create timer: ", err)
                 return
             end
         end

    }
```

观察日志，会发现每 3s worker 进程只执行一次。

如果 worker 进程意外终止，Nginx 的 master 进程会保证在 worker 进程意外终止后重启新

的 worker 进程，ID 保持不变。

如果要求定时任务只在 Nginx 重载时执行一次，可以使用如下方式：

```
local ok, err = ngx.timer.at(0,func)
```

这表示立即执行 func 函数，且由于没有回调 ngx.timer.at 的指令，只会执行一次。

注意：关于定时任务，需要在在 init_worker_by_lua*的执行阶段中执行（详见 8.2 节）。

定时任务在 Nginx 后台运行，不直接和客户端请求打交道，因此不会直接影响请求的响应时间，但这并不代表它不会干扰请求的响应时间，如果在同一时间内有大量定时任务执行，也会降低 Nginx 的整体性能。此时，可以使用如下指令对正在运行的定时任务进行控制。

指令：lua_max_running_timers

语法：lua_max_running_timers <count>

默认值：lua_max_running_timers 256

环境：http

含义：设置被允许的 running timers（正在执行回调函数的计时器）的最大数量，如果超过这个数量，就会抛出“N lua_max_running_timers are not enough”，其中 N 是变量，指的是当前正在运行的 running timers 的数量。

指令：lua_max_pending_timers

语法：lua_max_pending_timers <count>

默认值：lua_max_pending_timers 1024

环境：http

含义：设置允许使用的 pending timers（执行挂起的定时器）的最大数量，如果在定时任务中超过这个限制，则会报“too many pending timers”错误。

7.15.3　禁用的 Lua API

ngx.timer.every 支持用户操作共享内存、读取数据库数据、获取系统时间等，但在定时任务中有些 API 是被明确禁止的，例如下面的 API。

- 子请求 ngx.location.capture。
- 向客户端输出的 Lua API（如 ngx.say、ngx.print 和 ngx.flush）。
- 以 ngx.req.开头的 Lua API。

7.16 常用指令

Ngx_Lua 提供了大量的 Lua API 指令来实现各种功能，本节将会介绍一些常用的指令。

7.16.1 请求重定向

在 Nginx 中通过 rewrite 对请求进行重定向，而在 Ngx_Lua 里可以使用 ngx.redirect、ngx.req.set_uri 来完成重定向，并且 Ngx_Lua 还提供了一个具有强大的扩展能力的 ngx.exec 指令。

指令：ngx.redirect

语法：ngx.redirect(uri, status?)

环境：rewrite_by_lua*、access_by_lua*、content_by_lua*

含义：发出一个 HTTP 状态码为 301 或 302 的重定向请求到指定的 URI。

参数 status 的可选值有 301、302、303、307 和 308，默认值是 302。下面是 ngx.redirect 重定向和 rewrite 重定向的对比：

```
location / {
    # 等同于 rewrite ^/ http://testnginx.com/test? redirect;
    rewrite_by_lua_block {
        return ngx.redirect("/test")
    }
}
```

上述配置使用了默认状态码 302。如果在跳转过程中需要保留请求的参数，可做如下配置：

```
location / {
    # 等同于 rewrite ^/ http://testnginx.com/test permanent;
    rewrite_by_lua_block {
        local ngx = require "ngx";
        return ngx.redirect("/test?" ..  ngx.var.args  ,301)
    }
}
```

也可以自定义参数，如下所示：

```
return ngx.redirect("/test?test=1&a=2" ,301)
```

支持跳转到其他域名，如 http://abc.testnginx.com：

```
return ngx.redirect("http://abc.testnginx.com",301)
```

注意：跳转时都需要加 return 指令，其作用是为了强调跳转操作，官方推荐使用这种方式。

指令：ngx.req.set_uri

语法：ngx.req.set_uri (uri, jump?)

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*

含义：用参数 uri 来重写当前的 URL，和 Nginx 的 rewrite 内部重定向功能相似。例如，rewrite 的指令 rewrite ^ /test last; 与 ngx.req.set_uri("/test", true)功能相似，而 rewrite ^ /test break; 则与 ngx.req.set_uri("/foo", false)功能相似。

如果需要在跳转过程中修改参数，可以使用 ngx.req.set_uri_args 来完成新的参数配置，操作如下：

```
ngx.req.set_uri_args("a=1&b=2&c=3")
ngx.req.set_uri("/test", true)
```

指令：ngx.exec

语法：ngx.exec(uri, args?)

环境：rewrite_by_lua、access_by_lua*、content_by_lua*

含义：使用 uri、args 参数来完成内部重定向，类似于 echo-nginx-module 的 echo_exec 指令。

示例如下：

```
server {
    listen       80;
    server_name  testnginx.com;
    default_type 'text/plain';
    location / {
        content_by_lua_block {
            return ngx.exec('/test');
        }
    }
    location /test {
        content_by_lua_block {
            ngx.say(ngx.var.args);
        }
    }
}
```

下面是 ngx_exec 指令常用的几种参数设置的示例。

- 保留之前的参数，如“ngx.exec('/test',ngx.var.args);”。

- 保留之前的参数，并新增参数，如“ngx.exec('/test' ,ngx.var.args .. 'd=4');”。
- 去掉之前的参数，并新增参数，如“ngx.exec('/test' , 'd=4');”。

注意：ngx_exec 是一个内部重定向指令，不涉及外部的 HTTP 请求。在使用中推荐采用 return ngx.exec(…)的方式。

7.16.2 日志记录

在使用 Lua 进行开发的过程中，需要使用日志来输出异常和调试信息，在 Lua API 中可以使用 ngx.log 来记录日志。

指令：ngx.log

语法：ngx.log(log_level, …)

环境：init_by_lua*、init_worker_by_lua*、set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、balancer_by_lua*、ssl_certificate_by_lua*、ssl_session_fetch_by_lua*、ssl_session_store_by_lua*

含义：根据 log_level 的等级，将内容记录到 error.log 的日志文件中。

log_level 的级别及其说明见表 7-4（和 Nginx 的 error.log 日志级别是一致的）。

表 7-4 log_level 的级别及其说明

级　别	说　明
ngx.STDERR	标准输出
ngx.EMERG	紧急错误，需要引起重视
ngx.ALERT	警告，需要引起重视
ngx.CRIT	严重问题，需要引起重视
ngx.ERR	错误，需要引起重视
ngx.WARN	告警，一般可以忽略
ngx.NOTICE	通知性信息
ngx.INFO	各种打印信息，一般不会在线上环境中使用
ngx.DEBUG	调试日志，开发环境可以使用

示例如下：

```
server {
    listen       80;
    server_name  testnginx.com;
    default_type 'text/plain';
```

```
        location / {
            content_by_lua_block {
                ngx.say("test ")
                ngx.say("nginx ")
                ngx.log(ngx.ALERT,'Log Test Nginx')
                ngx.log(ngx.STDERR,'Log Test Nginx')
                ngx.log(ngx.EMERG,'Log Test Nginx')
                ngx.log(ngx.ALERT,'Log Test Nginx')
                ngx.log(ngx.CRIT,'Log Test Nginx')
                ngx.log(ngx.ERR,'Log Test Nginx')
                ngx.log(ngx.WARN,'Log Test Nginx')
                ngx.log(ngx.NOTICE,'Log Test Nginx')
                ngx.log(ngx.INFO,'Log Test Nginx')
                ngx.log(ngx.DEBUG,'Log Test Nginx')
            }
        }

    }
```

执行结果如下：

```
    curl -i 'http://testnginx.com/'
```

查看 error.log 日志，该日志默认在 logs/error.log 文件中，示例如下：

```
    2018/06/11   11:18:26   [alert]   1180#1180:   *34   [lua]   content_by_lua
(nginx.conf:66):4:   Log   Test   Nginx,   client:   10.19.48.161,   server:
testnginx.com, request: "GET / HTTP/1.1", host: "testnginx.com"
    2018/06/11   11:18:26   []   1180#1180:   *34   [lua]   content_by_lua
(nginx.conf:66):5:   Log   Test   Nginx,   client:   10.19.48.161,   server:
testnginx.com, request: "GET / HTTP/1.1", host: "testnginx.com"
    2018/06/11   11:18:26   [emerg]   1180#1180:   *34   [lua]   content_by_lua
(nginx.conf:66):6:   Log   Test   Nginx,   client:   10.19.48.161,   server:
testnginx.com, request: "GET / HTTP/1.1", host: "testnginx.com"
    2018/06/11   11:18:26   [alert]   1180#1180:   *34   [lua]   content_by_lua
(nginx.conf:66):7:   Log   Test   Nginx,   client:   10.19.48.161,   server:
testnginx.com, request: "GET / HTTP/1.1", host: "testnginx.com"
    2018/06/11   11:18:26   [crit]   1180#1180:   *34   [lua]   content_by_lua
(nginx.conf:66):8:   Log   Test   Nginx,   client:   10.19.48.161,   server:
testnginx.com, request: "GET / HTTP/1.1", host: "testnginx.com"
    2018/06/11   11:18:26   [error]   1180#1180:   *34   [lua]   content_by_lua
(nginx.conf:66):9:   Log   Test   Nginx,   client:   10.19.48.161,   server:
testnginx.com, request: "GET / HTTP/1.1", host: "testnginx.com"
```

观察 error.log 日志可发现，它并没有输出所有级别的日志，这是因为 Nginx 中 error.log 的日志级别会影响 Lua 中日志的级别。如果将 error.log 的级别修改如下：

```
error_log /usr/local/nginx_1.12.2/logs/error.log info;
```

这样 Lua 的日志就可以打印到 info 级别了，如果需要 debug 级别的日志，重新编译 Nginx 并开启 Debug 模式即可。

ngx.log 支持多个字符串合并输出，字符串之间以逗号分隔，示例如下：

```
ngx.log(ngx.ERR, 'Log Test Nginx', 'a', 'b', 'c')
```

ngx.log 单条日志可输出的最大字节数受 Nginx 的限制，默认最多是 2048 字节，即 2KB。

Lua 提供了 print 命令来简化 info 级别日志的输出。下面两条语句的作用是一样的：

```
print("Log Test Nginx ")
ngx.log(ngx.INFO, 'Log Test Nginx')
```

注意：ngx.print 和 print 是两条命令，不要混淆了。

7.16.3 请求中断处理

在 Lua 中，可以对请求进行中断处理，有两种情况。

- 中断整个请求，则请求不再继续执行，直接返回到客户端。
- 中断当前的执行阶段，请求会继续执行下一个阶段，并继续响应请求。

它们都是通过 ngx.exit 指令完成的。

指令：ngx.exit

语法：ngx.exit(status)

环境：rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、ngx.timer.*、balancer_by_lua*、ssl_certificate_by_lua*、ssl_session_fetch_by_lua*、ssl_session_store_by_lua*

含义：参数 status 的值是 HTTP 的状态码。当参数 status≥200 时，请求会被中断，并将 status 的值作为状态码返回给 Nginx；当参数 status=0 时，请求会中断当前的执行阶段，继续执行下一个阶段（前提是还有下一个阶段）。

环境：init_by_lua*、set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_ lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、balancer_by_ lua*、ssl_certificate_by_lua*、ssl_session_fetch_by_lua*、ssl_session_store_by_lua*

Ngx_Lua HTTP 状态码清单见表 7-5。

表 7-5　Ngx_Lua HTTP 状态码清单

赋值方式	HTTP 状态码
value = ngx.HTTP_CONTINUE	100
value = ngx.HTTP_SWITCHING_PROTOCOLS	101
value = ngx.HTTP_OK	200
value = ngx.HTTP_CREATED	201
value = ngx.HTTP_ACCEPTED	202
value = ngx.HTTP_NO_CONTENT	204
value = ngx.HTTP_PARTIAL_CONTENT	206
value = ngx.HTTP_SPECIAL_RESPONSE	300
value = ngx.HTTP_MOVED_PERMANENTLY	301
value = ngx.HTTP_MOVED_TEMPORARILY	302
value = ngx.HTTP_SEE_OTHER	303
value = ngx.HTTP_NOT_MODIFIED	304
value = ngx.HTTP_TEMPORARY_REDIRECT	307
value = ngx.HTTP_PERMANENT_REDIRECT	308
value = ngx.HTTP_BAD_REQUEST	400
value = ngx.HTTP_UNAUTHORIZED	401
value = ngx.HTTP_PAYMENT_REQUIRED	402
value = ngx.HTTP_FORBIDDEN	403
value = ngx.HTTP_NOT_FOUND	404
value = ngx.HTTP_NOT_ALLOWED	405
value = ngx.HTTP_NOT_ACCEPTABLE	406
value = ngx.HTTP_REQUEST_TIMEOUT	408
value = ngx.HTTP_CONFLICT	409
value = ngx.HTTP_GONE	410
value = ngx.HTTP_UPGRADE_REQUIRED	426
value = ngx.HTTP_TOO_MANY_REQUESTS	429
value = ngx.HTTP_CLOSE	444
value = ngx.HTTP_ILLEGAL	451
value = ngx.HTTP_INTERNAL_SERVER_ERROR	500
value = ngx.HTTP_METHOD_NOT_IMPLEMENTED	501
value = ngx.HTTP_BAD_GATEWAY	502
value = ngx.HTTP_SERVICE_UNAVAILABLE	503

续表

赋值方式	HTTP 状态码
value = ngx.HTTP_GATEWAY_TIMEOUT	504
value = ngx.HTTP_VERSION_NOT_SUPPORTED	505
value = ngx.HTTP_INSUFFICIENT_STORAGE	507

下面是一个 HTTP 状态码为 0 的示例：

```
server {
    listen       80;
    server_name  testnginx.com;
    default_type 'text/plain';
    location / {
        set $a  '0';
        rewrite_by_lua_block {
             ngx.var.a = '1';
             -- 等价于 ngx.exit(0), 0 即 HTTP 状态码
             ngx.exit(ngx.OK)
        }
        echo $a;  #执行结果等于 1
    }
}
```

ngx.exit(ngx.OK)可以让 Nginx 退出当前的 rewrite_by_lua_block 阶段，继续执行下面的阶段，如上面代码中的 echo。

如果想中断当前的请求，不再继续执行后面的执行阶段，可以设置两种退出状态。

- 设置状态码大于或等于 200 且小于 300，表示成功退出当前请求。
- 设置状态码大于或等于 500，或者是其他异常的状态码，表示失败退出当前请求。

继续使用上面的例子，这次以非 0 的状态码退出当前请求，如下所示：

```
location / {
    set $a  '0';
    rewrite_by_lua_block {
         ngx.var.a = '1';
         ngx.exit(200)   -- 也可以换成 500，数字代表状态码的值
    }
         echo $a;  # 没有执行到这一句
}
```

因为使用了 200 状态码，所以请求在 ngx.exit 处被中断后退出了，所以无法执行 echo 输出

的命令。为了强调退出操作，可以在此命令前加上 return，如下所示：

```
return ngx.exit(ngx.OK)
```

7.17　提升开发和测试效率

在使用 Lua 进行开发的过程中，可能需要频繁修改 Lua 代码，默认情况下都须重启 Nginx 才能使修改生效。使用 lua_code_cache 指令可以对其进行重新配置，并以此来提升开发效率。

语法：lua_code_cache on | off

默认值：lua_code_cache on

环境：http、server、location、location if

含义：打开或关闭*_by_lua_file 指定的 Lua 代码及 Lua 模块的缓存。如果设置为 off，则缓存会被关闭，在*_by_lua_file 中修改的代码不需要重载 Nginx 配置就可以生效。

注意：此指令只适合用于*_by_lua_file 中的代码，不适用于 *_by_lua_block 和 *_by_lua 中的代码，因为这两种指令的代码都是内嵌到 Nginx 配置文件中的，必须通过 reload 配置文件才可以使修改生效。把 lua_code_cache 设置为 on 只适合在开发环境中使用，不适合在线上环境中使用。

7.17.1　断开客户端连接

对于某些 API 请求来说，客户端只管发送请求并不等待返回结果。例如，触发一个请求通知远程服务端执行某个任务或进行日志推送。此时，可以使用如下指令断开连接。

指令：ngx.eof

语法：ok, err = ngx.eof()

环境：rewrite_by_lua*、access_by_lua*、content_by_lua*

含义：显示指定响应的输出结束，会告知客户端主动关闭连接，并在服务器端继续执行剩下的操作。

```
server {
    listen       80;
    server_name  testnginx.com;
    default_type 'text/plain';
    location / {
```

```
            set $a '0';
            content_by_lua_block {
                ngx.var.a = '1';
                -- 告知客户端主动断开连接
                ngx.eof()
                ngx.sleep(3);  -- 让请求休眠 3s
                ngx.log(ngx.ERR,'Test Nginx---',ngx.var.a)
            }
        }
    }
```

执行 curl -i 'http://testnginx.com/后，请求会立刻响应一个状态码 200，表示响应内容已返回，但请求的后续操作仍在服务器端继续执行，3s 后会将日志写入 error.log。

注意：执行完 ngx.eof 后，如果下一步是发送子请求的指令，那么，子请求会被意外中止，导致无法完成子请求的响应，这是受 Nginx 中 proxy_ignore_client_abort 默认值的影响，将 proxy_ignore_client_abort 设置为 on，就可以在执行 ngx.eof 后继续响应子请求了。

7.17.2 请求休眠

指令：ngx.sleep

语法：ngx.sleep(seconds)

环境：rewrite_by_lua*、access_by_lua*、content_by_lua*、ngx.timer.*、ssl_certificate_by_lua*、ssl_session_fetch_by_lua*

含义：通过 ngx.sleep 命令可以在不阻塞 Nginx worker 进程的情况下，让当前请求休眠指定时间（即 seconds 的参数值），seconds 最小值为 0.001s。

示例如下：

```
location / {
    content_by_lua_block {
        -- 5s 后输出 ok
        ngx.sleep(5);
        ngx.say('ok')
    }
}
```

7.17.3 获取系统时间

在 Ngx_Lua 中获取系统时间，都是从 Nginx 的时间缓存中读取的，不涉及系统调用（系

统调用的 Lua 命令类似于通过 os.time 获取系统时间）。相关指令的配置环境都是相同的，都适用于如下执行阶段。

init_worker_by_lua*、set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、balancer_by_lua*、ssl_certificate_by_lua*、ssl_session_fetch_ by_lua*、ssl_session_store_by_lua*

在 Ngx_Lua 中获取系统时间的示例如下：

```
server {
    listen       80;
    server_name  testnginx.com;
    default_type 'text/plain';
    location / {
       content_by_lua_block {
            ngx.say('ngx.today: ',ngx.today())
            ngx.say('ngx.time: ',ngx.time())
            ngx.say('ngx.now: ',ngx.now())
            ngx.say('ngx.localtime: ',ngx.localtime())
            ngx.say('ngx.utctime: ',ngx.utctime())
            ngx.say('ngx.cookie_time: ',ngx.cookie_time(1528721405))
            ngx.say('ngx.parse_http_time:  ',ngx.parse_http_time('Mon,  11-
Jun-18 12:50:05 GMT'))
            ngx.say('ngx.update_time: ',ngx.update_time())
       }
    }
}
```

执行结果如下：

```
ngx.today: 2018-06-11     # 返回系统的本地时间，只包含年、月、日
ngx.time: 1528721734      # 返回当前时间的 UNIX 时间戳
ngx.now: 1528721734.775 # 返回当前时间的 UNIX 时间戳，浮点数类型，小数部分是毫秒级别
ngx.localtime: 2018-06-11 20:55:34    # 返回当前时间
ngx.utctime: 2018-06-11 12:55:34      # 返回 UTC（Coordinated Universal
                                      # Time，即世界标准时间）
ngx.cookie_time: Mon, 11-Jun-18 12:50:05 GMT  # 返回一个可以让 Cookie 过期的时
                                              # 间格式，参数是 UNIX 时间戳格式
ngx.http_time: Mon, 11 Jun 2018 12:50:05 GMT  # 返回一个可以做 HTTP 头部的时间
                                    # 格式，如 Expires 或 Last-Modified
ngx.parse_http_time: 1528721405   # 返回 UNIX 时间戳，和 ngx.http_time 输出的时
                                  # 间格式不一样
```

```
ngx.update_time:      # 返回空，作用是强行更新 Nginx 的时间缓存
                      # 此操作会增加性能开销，不建议使用
```

7.17.4 编码与解码

利用 Ngx_Lua 的 API，可以进行编码和解码的操作。

指令：ngx.escape_uri

语法：newstr = ngx.escape_uri(str)

环境：init_by_lua*、init_worker_by_lua*、set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、balancer_by_lua*、ssl_certificate_by_lua*、ssl_session_fetch_by_lua*、ssl_session_store_by_lua* ngx.quote_sql_str

含义：对参数 str 进行 URI 编码。

指令：ngx.unescape_uri

语法：newstr = ngx.unescape_uri(str)

环境：init_by_lua*、init_worker_by_lua*、set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、balancer_by_lua*、ssl_certificate_by_lua*

含义：对参数 str 进行 URI 解码。

指令：ngx.encode_args

语法：str = ngx.encode_args(table)

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、balancer_by_lua*、ssl_certificate_by_lua*

含义：按照 URI 编码规则，将 Lua 提供的 table 类型数据编码成一个字符串。

指令：ngx.decode_args

语法：table, err = ngx.decode_args(str, max_args?)

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、balancer_by_lua*、ssl_certificate_by_lua*、ssl_session_fetch_by_lua*、ssl_session_store_by_lua*

含义：将 URI 编码的字符串解码为 Lua 的 table 类型的数据。

指令：ngx.md5

语法：digest = ngx.md5(str)

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、balancer_by_lua*、ssl_certificate_by_lua*、ssl_session_fetch_by_lua*、ssl_session_store_by_lua*

含义：对 str 字符串进行 md5 加密，并返回十六进制的数据。

指令：ngx.md5_bin

语法：digest = ngx.md5_bin(str)

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、balancer_by_lua*、ssl_certificate_by_lua*、ssl_session_fetch_by_lua*、ssl_session_store_by_lua*

含义：对 str 字符串进行 md5 加密，并返回二进制的数据。

注意：ngx.escape_uri 和 ngx.unescape_uri 作用相反，ngx.encode_args 和 ngx.decode_args 的作用相反。

关于编码、解码操作的示例如下：

```
   server {
       listen       80;
       server_name  testnginx.com;
       default_type 'text/plain';
       location / {
          content_by_lua_block {
               local ngx = require "ngx";
               -- 对 URI 进行编码
               ngx.say(ngx.var.uri, '---ngx.escape_uri---', ngx.escape_uri
(ngx.var.uri))

               -- 对已经编码过的 URI 进行解码
               ngx.say('%2Ftest%2Fa%2Fb%2Fc', '---ngx.unescape_uri---',
ngx.unescape_uri('%2Ftest%2Fa%2Fb%2Fc'))

               -- 将 Lua 的 table 类型数据编码成字符串
               local  args_table_new =  ngx.encode_args({a = 1, b = 2, c =
```

```
3 })
                ngx.say('{a = 1, b = 2, c = 3 }', '---ngx.encode_args---' ,
args_table_new)

                -- 对 URI 编码的字符串进行解码，解码成 table 类型的数据
                local args = ngx.var.args
                local args_table = ngx.decode_args(args)
                ngx.say(args, '---ngx.decode_args---', 'a=',args_table["a"])
                -- 获取 table 中的 a 的值
                -- 对 URI 进行 md5 编码，返回十六进制数据
                ngx.say(ngx.var.uri, '---ngx.md5---',ngx.md5(ngx.var.uri))
                -- 对 URI 进行 md5 编码，返回二进制数据
                ngx.say(ngx.var.uri, '---ngx.md5_bin---',
ngx.md5_bin(ngx.var.uri))
            }
        }
    }
```

执行结果如下：

```
    # curl  'http://testnginx.com/test/a/b/c?a=1&b=2&c=3'
    /test/a/b/c---ngx.escape_uri---%2Ftest%2Fa%2Fb%2Fc
    %2Ftest%2Fa%2Fb%2Fc---ngx.unescape_uri---/test/a/b/c
    {a = 1, b = 2, c = 3 }---ngx.encode_args---b=2&a=1&c=3
    a=1&b=2&c=3---ngx.decode_args---a=1
    /test/a/b/c---ngx.md5---dfa371a9a8f52c9aadd016bda535fa43
    /test/a/b/c---ngx.md5_bin---▪ q©¨□Z½¥5
```

7.17.5 防止 SQL 注入

Lua API 还提供了 ngx.quote_sql_str 指令，它可以按照 MySQL 格式把字符串转义为 SQL 语句，如果 Nginx 需要和 MySQL 数据库打交道，该指令也可以用来防止 SQL 注入（Nginx 和数据库的交互详见第 9 章）。

指令：ngx.quote_sql_str

语法：quoted_value = ngx.quote_sql_str(raw_value)

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、balancer_by_lua*、ssl_certificate_by_lua*、ssl_session_fetch_by_lua*、ssl_session_store_by_lua*

7.17.6　判断是否为子请求

指令：ngx.is_subrequest

语法：value = ngx.is_subrequest

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*

含义：判断当前请求是否为 Nginx 的子请求，如果是，则结果为 true，反之为 false。

示例如下：

```
server {
    listen       80;
    server_name  testnginx.com;
    default_type 'text/plain';
    location /main1 {
         echo_location /test;
    }

    location /main2 {
       content_by_lua_block {
         return ngx.exec('/test');
       }
    }

    location /main3 {
       content_by_lua_block {
          local res = ngx.location.capture("/test")
          ngx.say(res.body)
       }
    }

    location /test {
      content_by_lua_block {
          ngx.say('test');
          ngx.say('is_subrequest: ',ngx.is_subrequest);
      }
    }
}
```

执行结果如下：

```
[root@testnginx ~]# curl  'http://testnginx.com/main1'
```

```
test
is_subrequest: true
[root@testnginx ~]# curl  'http://testnginx.com/main2'
test
is_subrequest: false
[root@testnginx ~]# curl  'http://testnginx.com/main3'
test
is_subrequest: true
```

从执行结果可以看出，ngx.exec 命令执行的不是子请求，而是内部重定向；echo_location 和 ngx.location.capture 命令执行的是子请求。

7.17.7 设置 MIME 类型

当 Ngx_Lua 作为响应体输出时，默认会使用 Nginx 的 MIME 类型，但这会导致响应体的数据类型（如 JSON、HTML 等）无法区分。

可以使用 Default_Type 来定义 Lua 输出响应体的数据类型，例如定义为 JSON 格式，可以进行如下配置：

```
default_type application/json;
```

语法：default_type mime-type;

默认值：default_type text/plain;

环境：http、server、location

建议：给 Lua 输出的所有响应数据都加上数据类型，也可以在 Ngx_Lua 输出时新增一个请求头 Content-Type 来表明请求类型。

7.18 小结

这一章讲解了 Ngx_Lua 的很多指令，它们都是在 Nginx 开发中常见的指令。Ngx_Lua 模块的功能还在不断迭代中，读者可以经常光顾 Ngx_Lua 的官方 Wiki 或访问 openresty.org，看看这些指令在最新的 Nginx 版本中是否又有了不同的“玩法”。

第 8 章
Ngx_Lua 的执行阶段

第 7 章介绍了 Ngx_Lua 的常用指令，对刚接触 Ngx_Lua 的读者来说，可能会存在下面两种困惑。

- Lua 在 Nginx 的哪些阶段可以执行代码。
- Lua 在 Nginx 的每个阶段可以执行哪些操作。

只有理解了这两个问题，才能在业务中巧妙地利用 Ngx_Lua 来完成各项需求。

在 3.9 节中介绍过 Nginx 的 11 个执行阶段，每个阶段都有自己能够执行的指令，并可以实现不同的功能。Ngx_Lua 的功能大部分是基于 Nginx 这 11 个执行阶段进行开发和配置的，Lua 代码在这些指令块中执行，并依赖于它们的执行顺序。本章将对 Ngx_Lua 的执行阶段进行一一讲解。

8.1 init_by_lua_block

init_by_lua_block 是 init_by_lua 的替代版本，在 OpenResty 1.9.3.1 或 Lua-Nginx-Modulev 0.9.17 之前使用的都是 init_by_lua。init_by_lua_block 比 init_by_lua 更灵活，所以建议优先选用 init_by_lua_block。

本章中的执行阶段都采用*_block 格式的指令，后续不再说明。

8.1.1 阶段说明

语法：init_by_lua_block {lua-script-str}

环境：http

阶段：loading-config

含义：当 Nginx 的 master 进程加载 Nginx 配置文件（加载或重启 Nginx 进程）时，会在全局的 Lua VM（Virtual Machine，虚拟机）层上运行<lua-script-str> 指定的代码，每次当 Nginx 获得 HUP（即 Hangup）重载信号加载进程时，代码都会被重新执行。

8.1.2 初始化配置

在 loading-config 阶段一般会执行如下操作。

1．初始化 Lua 全局变量，特别适合用来处理在启动 master 进程时就要求存在的数据，对 CPU 资源消耗较多的功能也可以在此处处理。

2．预加载模块。

3．初始化 lua_shared_dict 共享内存的数据（关于共享内存的知识详见第 10 章）。

示例如下：

```
    user  webuser webuser;
    worker_processes  1;
    worker_rlimit_nofile 10240;

    events {
        use epoll;
        worker_connections  10240;
    }

    http {
        include       mime.types;
        default_type  application/octet-stream;
        log_format main '$remote_addr-$remote_user[$time_local] "$request" '
                        '$status $body_bytes_sent "$http_referer" '
                        '"$http_user_agent"          "$http_x_forwarded_for"
"$request_time"  "$upstream_addr  $upstream_status  $upstream_response_time"
"upstream_time_sum:$upstream_time_sum"  "jk_uri:$jk_uri"';
        access_log  logs/access.log  main;
        sendfile        on;
        keepalive_timeout  65;


        lua_package_path "/usr/local/nginx_1.12.2/conf/lua_modules/?.lua;;";
```

```
        lua_package_cpath
"/usr/local/nginx_1.12.2/conf/lua_modules/c_package/?.so;;";

        lua_shared_dict dict_a 100k;  -- 声明一个 Lua 共享内存，dict_a 为 100KB
    init_by_lua_block {
    -- cjson.so 文件需要手动存放在 lua_package_cpath 搜索路径下
    -- 如果是 OpenResty，就不必了，因为它默认支持该操作
            cjson = require "cjson";
            local dict_a = ngx.shared.dict_a;
            dict_a:set("abc", 9)
        }

        server {
            listen       80;
            server_name  testnginx.com;
            location / {
               content_by_lua_block {
                   ngx.say(cjson.encode({a = 1, b = 2}))
                   local dict_a = ngx.shared.dict_a;
                   ngx.say("abc=",dict_a:get("abc"))
               }
            }
        }
```

执行结果如下：

```
# curl -I http://testnginx.com/
{"a":1,"b":2}
abc=9
```

8.1.3　控制初始值

在 init_by_lua_block 阶段设置的初始值，即使在其他执行阶段被修改过，当 Nginx 重载配置时，这些值就又会恢复到初始状态。如果在重载 Nginx 配置时不希望再次改动这些初始值，可以在代码中做如下调整。

```
init_by_lua_block {
      local cjson = require "cjson";
      local dict_a = ngx.shared.dict_a;
      local v = dict_a:get("abc");  -- 判断初始值是否已经被 set
      if not v then                 -- 如果没有，就执行初始化操作
        dict_a:set("abc", 9)
      end
}
```

8.1.4 init_by_lua_file

init_by_lua_file 和 init_by_lua_block 的作用一样，主要用于将 init_by_lua_block 的内容转存到指定的文件中，这样 Lua 代码就不必全部写在 Nginx 配置里了，易读性更强。

init_by_lua_file 支持配置绝对路径和相对路径。相对路径是在启动 Nginx 时由-p PATH 决定的。如果启动 Nginx 时没有配置-p PATH，就会使用编译时--prefix 的值，该值一般存放在 Nginx 的$prefix（也可以用${prefix}来表示）变量中。init_by_lua_file 和 Nginx 的 include 指令的相对路径一致。

示例如下：

```
init_by_lua_file conf/lua/init.lua;                    -- 相对路径
init_by_lua_file /usr/local/nginx/conf/lua/init.lua; -- 绝对路径
```

init.lua 文件的内容如下：

```
cjson = require "cjson"
local dict_a = ngx.shared.dict_a
local v = dict_a:get("abc")
if not v then
   dict_a:set("abc", 9)
end
```

8.1.5 可使用的 Lua API 指令

init_by_lua*是 Nginx 配置加载的阶段，很多 Lua API 命令是不支持的。目前已知支持的 Lua API 的命令有 ngx.log、ngx.shared.DICT、print。

注意：init_by_lua*中的*表示通配符，init_by_lua*即所有以 init_by_lua 开头的 API。后续的通配符亦是如此，不再另行说明。

8.2 init_worker_by_lua_block

8.2.1 阶段说明

语法：init_worker_by_lua_block {lua-script-str}

环境：http

阶段：starting-worker

含义：当 master 进程被启动后，每个 worker 进程都会执行 Lua 代码。如果 Nginx 禁用了

master 进程，init_by_lua*将会直接运行。

8.2.2　启动 Nginx 的定时任务

在 init_worker_by_lua_block 执行阶段最常见的功能是执行定时任务，示例如下：

```
user  webuser webuser;
worker_processes  3;
worker_rlimit_nofile 10240;

events {
    use epoll;
    worker_connections  10240;
}

http {
    include       mime.types;
    default_type  application/octet-stream;
    sendfile        on;
    keepalive_timeout  65;

    upstream  test_12 {
        server 127.0.0.1:81 weight=20 max_fails=300000 fail_timeout=5s;
        server 127.0.0.1:82 weight=20 max_fails=300000 fail_timeout=5s;
    }

    lua_package_path "${prefix}conf/lua_modules/?.lua;;";
    lua_package_cpath "${prefix}conf/lua_modules/c_package/?.so;;";

    init_worker_by_lua_block  {
        local delay = 3  --3s
        local cron_a
    -- 定时任务的函数
        cron_a = function (premature)
    -- 如果执行函数时没有传参，则该任务会一直被触发执行
            if not  premature then
                ngx.log(ngx.ERR, "Just do it !")
            end

        end
    -- 每隔 delay 参数值的时间，就执行一次 cron_a 函数
        local ok, err = ngx.timer.every(delay, cron_a)
        if not ok then
            ngx.log(ngx.ERR, "failed to create the timer: ", err)
```

```
            return
        end
    }
```

8.2.3 动态进行后端健康检查

在 init_worker_by_lua_block 阶段，也可以实现后端健康检查的功能，用于检查后端 HTTP 服务是否正常，类似于 Nginx 商业版中的 health_check 功能。

如果使用 OpenResty 1.9.3.2 及以上的版本，默认已支持 lua-upstream-nginx-module 模块；如果使用 Nginx，则首先需要安装此模块，安装方式如下：

```
# git clone https://github.com/openresty/lua-upstream-nginx-module.git
# cd nginx-1.12.2
# ./configure --prefix=/opt/nginx \
    --with-ld-opt="-Wl,-rpath,$LUAJIT_LIB" \
    --add-module=/path/to/lua-nginx-module \
    --add-module=/path/to/lua-upstream-nginx-module
# make && make install
```

注意：安装完成后，在重新编译 Nginx 时，请首先确认已安装模块的数量，避免遗忘某个模块。

然后，将 lua-resty-upstream-healthcheck 模块中的 lib 文件复制到 lua_package_path 指定的位置，示例如下：

```
# git clone https://github.com/openresty/lua-resty-upstream-healthcheck.git
# cp lua-resty-upstream-healthcheck/lib/resty/upstream/healthcheck.lua /usr/local/nginx_1.12.2/conf/lua_modules/resty/
```

实现动态进行后端健康检查的功能，配置如下：

```
user  webuser webuser;
worker_processes  3;
worker_rlimit_nofile 10240;

events {
    use epoll;
    worker_connections  10240;
}
http {
    include       mime.types;
    default_type  application/octet-stream;
    sendfile        on;
    keepalive_timeout  65;
```

```
    upstream  test_12 {
        server 127.0.0.1:81  weight=20  max_fails=10 fail_timeout=5s;
        server 127.0.0.1:82  weight=20  max_fails=10 fail_timeout=5s;
        server 127.0.0.1:8231  weight=20  max_fails=10 fail_timeout=5s;
    }
    lua_shared_dict healthcheck 1m;  # 存放 upstream servers 的共享内存
                                     # 后端服务器组越多，配置就越大
    lua_socket_log_errors off;   # 当 TCP 发送失败时，会发送 error 日志到
                                 # error.log 中，该过程会增加性能开销，建议关
                                 # 闭以避免在健康检查过程中出现多台服务器宕机的
                                 # 情况，异常情况请使用 ngx.log 来记录
    lua_package_path "${prefix}conf/lua_modules/?.lua;;";
    lua_package_cpath "${prefix}conf/lua_modules/c_package/?.so;;";

    init_worker_by_lua_block  {
        local hc = require "resty.upstream.healthcheck"
        local ok, err = hc.spawn_checker{
            shm = "healthcheck",  -- 使用共享内存
            upstream = "test_12", -- 进行健康检查的 upstream 的名字
            type = "http",        -- 检查类型是 http
            http_req      =      "GET      /status      HTTP/1.0\r\nHost:
testnginx.com\r\n\r\n",
            -- 用来发送 HTTP 请求的格式和数据，核实服务是否正常
            interval = 3000,  -- 设置检查的频率为 3s/次
            timeout = 1000,   -- 设置请求的超时时间为 1s
            fall = 3,  -- 设置连续失败 3 次后就把后端服务改为 down
            rise = 2,  -- 设置连续成功 2 次后就把后端服务改为 up
        valid_statuses = {200, 302},  -- 设置请求成功的响应状态码是 200 和 302
            concurrency = 10,  -- 设置发送请求的并发等级
        }
        if not ok then
            ngx.log(ngx.ERR, "failed to spawn health checker: ", err)
            return
        end
    }
    server {
        listen       80;
        server_name  testnginx.com;
        location / {
           proxy_pass http://test_12;
        }
        # /status 定义了后端健康检查结果的输出页面
        location = /status {
```

```
            default_type text/plain;
            content_by_lua_block {
                local hc = require "resty.upstream.healthcheck"
                -- 输出当前检查结果是哪个 worker 进程的检查结果
                ngx.say("Nginx Worker PID: ", ngx.worker.pid())
                -- status_page()输出后端服务器的详细情况
                ngx.print(hc.status_page())
            }
        }
    }
```

访问 http://testnginx.com/status 查看检查的结果，图 8-1 所示为健康检查数据结果。

图 8-1　健康检查数据结果

如果要检查多个 upstream，则配置如下（只有黑色加粗位置的配置有变化）：

```
local ok, err = hc.spawn_checker{
    shm = "healthcheck",
    upstream = "test_12",
    type = "http",

    http_req = "GET /status HTTP/1.0\r\nHost: testnginx.com\r\n\r\n",

    interval = 3000,
    timeout = 1000,
    fall = 3,
    rise = 2,
    valid_statuses = {200, 302},
    concurrency = 10,
}
local ok, err = hc.spawn_checker{
    shm = "healthcheck",
    upstream = "test_34",
    type = "http",
```

```
    http_req = "GET /test HTTP/1.0\r\nHost: testnginx.com\r\n\r\n",
    interval = 3000,
    timeout = 1000,
    fall = 3,
    rise = 2,
    valid_statuses = {200, 302},
    concurrency = 10,
```

如果把 lua_socket_log_errors 设置为 on，那么当有异常出现时，例如出现了超时，就会往 error.log 里写日志，日志记录如图 8-2 所示。

```
2018/05/28 16:08:36 [error] 32374#32374: *64 connect() failed (111: Connection refused), context: ngx.timer
2018/05/28 16:08:38 [error] 32375#32375: *84 connect() failed (111: Connection refused), context: ngx.timer
2018/05/28 16:08:39 [error] 32375#32375: *84 lua tcp socket read timed out, context: ngx.timer
2018/05/28 16:08:39 [error] 32375#32375: *84 lua tcp socket read timed out, context: ngx.timer
2018/05/28 16:08:39 [error] 32375#32375: *84 lua tcp socket read timed out, context: ngx.timer
2018/05/28 16:08:40 [error] 32376#32376: *104 connect() failed (111: Connection refused), context: ngx.timer
2018/05/28 16:08:43 [error] 32374#32374: *124 connect() failed (111: Connection refused), context: ngx.timer
2018/05/28 16:08:45 [error] 32376#32376: *144 connect() failed (111: Connection refused), context: ngx.timer
2018/05/28 16:08:47 [error] 32375#32375: *164 connect() failed (111: Connection refused), context: ngx.timer
2018/05/28 16:08:49 [error] 32374#32374: *184 connect() failed (111: Connection refused), context: ngx.timer
2018/05/28 16:08:50 [error] 32374#32374: *184 lua tcp socket read timed out, context: ngx.timer
2018/05/28 16:08:52 [error] 32375#32375: *204 connect() failed (111: Connection refused), context: ngx.timer
2018/05/28 16:08:54 [error] 32376#32376: *224 connect() failed (111: Connection refused), context: ngx.timer
2018/05/28 16:08:55 [error] 32376#32376: *224 lua tcp socket read timed out, context: ngx.timer
2018/05/28 16:08:56 [error] 32374#32374: *244 connect() failed (111: Connection refused), context: ngx.timer
2018/05/28 16:08:57 [error] 32374#32374: *244 lua tcp socket read timed out, context: ngx.timer
2018/05/28 16:08:59 [error] 32375#32375: *264 connect() failed (111: Connection refused), context: ngx.timer
```

图 8-2　日志记录

经过 lua-resty-upstream-healthcheck 的健康检查发现异常的服务器后，Nginx 会动态地将异常服务器在 upstream 中禁用，以实现更精准的故障转移。

8.3　set_by_lua_block

8.3.1　阶段说明

语法：set_by_lua_block $res {lua-script-str}

环境：server、server if、location、location if

阶段：rewrite

含义：执行<lua-script-str>代码，并将返回的字符串赋值给$res。

8.3.2　变量赋值

本指令一次只能返回一个值，并赋值给变量$res（即只有一个$res 被赋值），示例如下：

```
server {
    listen       80;
    server_name  testnginx.com;
    location / {
    set $a '';
    set_by_lua_block $a {
         local t = 'tes'
         return t
    }
    return 200 $a;
}
```

执行结果如下：

```
#  curl http://testnginx.com/
test
```

那如果希望返回多个变量，该怎么办呢？可以使用 ngx.var.VARIABLE，示例如下：

```
server {
    listen       80;
    server_name  testnginx.com;
    location / {
    # 使用 ngx.var.VARIABLE 前须先定义变量
    set $b '';
    set_by_lua_block $a {
        local t = 'test'
        ngx.var.b = 'test_b'
        return t
    }
    return 200 $a,$b;
    }
}
```

执行结果如下：

```
#  curl http://testnginx.com/test
test,test_b
```

8.3.3 rewrite 阶段的混用模式

因为 set_by_lua_block 处在 rewrite 阶段，所以它可以和 ngx_http_rewrite_module、set-misc-nginx-module，以及 array-var-nginx-module 一起使用，在代码执行时，按照配置文件的顺序从上到下执行，示例如下：

```
server {
    listen       80;
    server_name  testnginx.com;
    location / {

    set $a '123';
    set_by_lua_block $b {
        local t = 'bbb'
        return t
    }
    set_by_lua_block $c {
        local t = 'ccc'  .. ngx.var.b
        return t
    }
    set $d "456$c";
    return 200 $a,$b,$c,$d;
    }

}
```

从执行结果可以看出数据是从上到下执行的，如下所示：

```
# curl  http://testnginx.com/test
123,bbb,cccbbb,456cccbbb
```

8.3.4　阻塞事件

set_by_lua_block 指令块在 Nginx 中执行的指令是阻塞型操作，因此应尽量在这个阶段执行轻、快、短、少的代码，以避免耗时过多。set_by_lua_block 不支持非阻塞 I/O，所以不支持 yield 当前 Lua 的轻线程。

8.3.5　被禁用的 Lua API 指令

在 set_by_lua_block 阶段的上下文中，下面的 Lua API 是被禁止的（只罗列其中一部分）。

- 输出类型的 API 函数（如 ngx.say 和 ngx.send_headers）。
- 控制类型的 API 函数（如 ngx.exit）。
- 子请求的 API 函数（如 ngx.location.capture 和 ngx.location.capture_multi）。
- Cosocket API 函数（如 ngx.socket.tcp 和 ngx.req.socket）。
- 休眠 API 函数 ngx.sleep。

8.4 rewrite_by_lua_block

8.4.1 阶段说明

语法：rewrite_by_lua_block {lua-script-str}

环境：http、server、location、location if

阶段：rewrite tail

含义：作为一个重写阶段的处理程序，对每个请求执行<lua-script-str>指定的 Lua 代码。

这些 Lua 代码可以调用所有的 Lua API，并且运行在独立的全局环境（类似于沙盒）中，以新的协程来执行。因为可以调用所有的 Lua API，所以此阶段可以实现很多功能，例如对请求的 URL 进行重定向、读取 MySQL 或 Redis 数据、发送子请求、控制请求头等。

8.4.2 利用 rewrite_by_lua_no_postpone 改变执行顺序

rewrite_by_lua_block 命令默认在 ngx_http_rewrite_module 之后执行，示例如下：

```
server {
    listen       80;
    server_name  testnginx.com;
    location / {
     set $b  '1';
     rewrite_by_lua_block { ngx.var.b = '2'}
     set $b  '3';
     echo $b;
    }
}
```

从代码来看，预计应输出的结果是 3，但输出的结果却是 2，如下所示：

```
#  curl  http://testnginx.com/test
2
```

上述配置的操作结果说明 rewrite_by_lua_block 始终都在 rewrite 阶段的后面执行，如果要改变这个顺序，需要使用 rewrite_by_lua_no_postpone 指令。

语法：rewrite_by_lua_no_postpone on|off

默认值：rewrite_by_lua_no_postpone off

环境：http

含义：在 rewrite 请求处理阶段，控制 rewrite_by_lua*的所有指令是否被延迟执行，默认为

off，即延迟到最后执行；如果设置为 on，则会根据配置文件的顺序从上到下执行。

示例如下：

```
rewrite_by_lua_no_postpone on;    # 只能在 http 阶段配置
server {
    listen       80;
    server_name  testnginx.com;
    location / {
        set $b  '1';
        rewrite_by_lua_block { ngx.var.b = '2'}
        set $b  '3';
        echo $b;
    }
}
```

执行结果如下：

```
#  curl -i http://testnginx.com/test
HTTP/1.1 200 OK
Server: nginx/1.12.2
Date: Mon, 28 May 2018 12:47:37 GMT
Content-Type: application/octet-stream
Transfer-Encoding: chunked
Connection: keep-alive

3
```

注意：if 语句是在 rewrite_by_lua_block 阶段之前执行的，所以在运用 if 语句时要特别留意执行顺序，避免出现意想不到的结果。

8.4.3　阶段控制

在 rewrite_by_lua_block 阶段，当调用 ngx.exit(ngx.OK)之后，请求会退出当前的执行阶段，继续下一阶段的内容处理（如果想了解更多关于 ngx.exit 的使用方式，请参考 7.16.3 小节）。

8.5　access_by_lua_block

8.5.1　阶段说明

语法：access_by_lua_block {lua-script-str}

环境：http、server、location、location if

阶段：access tail

含义：在 Nginx 的 access 阶段，对每个请求执行<lua-script-str>的代码，和 rewrite_by_lua_block 一样，这些 Lua 代码可以调用所有的 Lua API，并且运行在独立的全局环境（类似于沙盒）中，以新的协程来执行。此阶段一般用来进行权限检查和黑白名单配置。

8.5.2 利用 access_by_lua_no_postpone 改变执行顺序

access_by_lua_block 默认在 ngx_http_access_module 之后执行。但把 access_by_lua_no_postpone 设置为 on 可以改变执行顺序，变成根据配置文件的顺序从上到下执行。access_by_lua_no_ postpone 的用法和 rewrite_by_lua_no_postpone 类似。

8.5.3 阶段控制

access_by_lua_block 和 rewrite_by_lua_block 一样，都可以通过 ngx.exit 来结束当前的执行阶段。

8.5.4 动态配置黑白名单

Nginx 提供了 allow、deny 等指令来控制 IP 地址访问服务的权限，但如果每次新增 IP 地址都要重载配置文件就太不灵活了，而且反复重载 Nginx 也会导致服务不稳定。此时，可以通过 access_by_lua_block 来动态添加黑白名单。

下面是两种常见的动态配置黑白名单的方案。

- 将黑白名单存放在 Redis 中，然后使用 Nginx+Lua 读取 Redis 的数据，通过修改 Redis 中的数据来动态配置黑白名单。这种方案的缺点是增加了对网络 I/O 的操作，相比使用共享内存的方式，性能稍微差了些。
- 将黑白名单存放在 Ngx_Lua 提供的共享内存中，每次请求时都去读取共享内存中的黑白名单，通过修改共享内存的数据达到动态配置黑白名单的目的。本方案的缺点是在 Nginx 重启的过程中数据会丢失。

8.6 content_by_lua_block

8.6.1 阶段说明

语法：content_by_lua_block {lua-script-str}

环境：location、location if

阶段：content

含义：content_by_lua_block 作为内容处理阶段，对每个请求执行<lua-script-str>的代码。和 rewrite_by_lua_block 一样，这些 Lua 代码可以调用所有的 Lua API，并且运行在独立的全局环境（类似于沙盒）中，以新的协程来执行。

content_by_lua_block 指令不可以和其他内容处理阶段的模块如 echo、return、proxy_pass 等放在一起，示例如下：

```
server {
    listen       80;
    server_name  testnginx.com;

    location / {
        content_by_lua_block {
             ngx.say("content_by_lua_block")
        }
        echo 'ok';
    }
}
```

输出结果只有 ok，并未执行 content_by_lua_block 指令。

8.6.2　动态调整执行文件的路径

指令 content_by_lua_file 可以用来动态地调整执行文件的路径，它后面跟的 Lua 执行文件路径可以是变量，该变量可以是不同的参数或 URL 等，示例如下：

```
location / {
    # content_by_lua_file 可以直接获取 URL 中参数 file_name 的值
    content_by_lua_file conf/lua/$arg_file_name;
}
```

8.7　balancer_by_lua_block

8.7.1　阶段说明

语法：balancer_by_lua_block { lua-script }

环境：upstream

阶段：content

含义：在 upstream 的配置中执行，它会忽视原 upstream 中默认的配置并控制后端服务器的地址。

示例如下：

```
upstream foo {
    server 127.0.0.1;  # 会忽视这个配置
    balancer_by_lua_block {
        # 真实的 IP 地址在这里配置
    }
}
```

注意：如果想了解更多关于此阶段的内容，请参考 11.4 节。

8.7.2 被禁用的 Lua API 指令

在 balancer_by_lua_block 阶段，Lua 代码的执行环境不支持 yield，因此须禁用可能会产生 yield 的 Lua API 指令（如 cosockets 和 light threads）。但可以利用 ngx.ctx 创建一个拥有上下文的变量，在本阶段前面的某个执行阶段（如 rewrite_by_lua*阶段）将数据生成后传入 upstream 中。

8.8 header_filter_by_lua_block

8.8.1 阶段说明

语法：header_filter_by_lua_block { lua-script }

环境：http、server、location、location if

阶段：output-header-filter

含义：在 header_filter_by_lua_block 阶段，对每个请求执行<lua-script-str>的代码，以此对响应头进行过滤。常用于对响应头进行添加、删除等操作。

例如，添加一个响应头 test，值为 nginx-lua，代码如下：

```
location / {
    header_filter_by_lua_block {
       ngx.header.test = "nginx-lua";
    }
    echo 'ok';
}
```

8.8.2　被禁用的 Lua API 指令

在 header_filter_by_lua_block 阶段中，下列 Lua API 是被禁止使用的。

- 输出类型的 API 函数（如 ngx.say 和 ngx.send_headers）。
- 控制类型的 API 函数（如 ngx.redirect 和 ngx.exec）。
- 子请求的 API 函数（如 ngx.location.capture 和 ngx.location.capture_multi）。
- cosocket API 函数（如 ngx.socket.tcp 和 ngx.req.socket）。

8.9　body_filter_by_lua_block

8.9.1　阶段说明

语法：body_filter_by_lua_block { lua-script-str }

环境：http、server、location、location if

阶段：output-body-filter

含义：在 body_filter_by_lua_block 阶段执行<lua-script-str>的代码，用于设置输出响应体的过滤器。在此阶段可以修改响应体的内容，如修改字母的大小写、替换字符串等。

8.9.2　控制响应体数据

通过 ngx.arg[1]（Lua 的字符串类型的数据）输入数据流，结束标识 eof 是响应体数据的最后一位 ngx.arg[2]（Lua 的布尔值类型的数据）。

由于 Nginx chain 缓冲区的存在，数据流不一定都是一次性完成的，可能需要发送多次。在这种情况下，结束标识 eof 只是 Nginx chain 缓冲区的 last_buf（主请求）或 last_in_chain（子请求），因此 Nginx 输出过滤器在一个单独的请求中会被调用多次，此阶段的 Lua 指令也会被执行多次，示例如下：

```
    server {
        listen       80;
        server_name  testnginx.com;

        location / {
            # 将响应体全部转换为大写
            body_filter_by_lua_block   {   ngx.arg[1]   =   string.upper(ngx.
arg[1]) }
            echo 'oooKkk';
```

```
        echo 'oooKkk';
        echo 'oooKkk';
    }
}
```

执行结果如下：

```
#  curl -i http://testnginx.com/?file_name=1.lua
HTTP/1.1 200 OK
Server: nginx/1.12.2
Date: Tue, 29 May 2018 11:36:54 GMT
Content-Type: application/octet-stream
Transfer-Encoding: chunked
Connection: keep-alive

OOOKKK
OOOKKK
OOOKKK
```

使用 return ngx.ERROR 可以截断响应体，但会导致数据不完整，请求无效：

```
location / {
    body_filter_by_lua_block {
        ngx.arg[1] = string.upper(ngx.arg[1]);
        return ngx.ERROR
    }
    echo 'oooKkk';
    echo 'oooKkk';
    echo 'oooKkk';
}
```

执行结果如下：

```
#  curl -i http://testnginx.com/
curl: (52) Empty reply from server
```

ngx.arg[2]是一个布尔值，如果把它设为 true，可以用来截断响应体的数据，和 return ngx.ERROR 不一样，此时被截断的数据仍然可以输出内容，是有效的请求，示例如下：

```
server {
    listen       80;
    server_name  testnginx.com;
    location / {
        body_filter_by_lua_block {
           local body_chunk = ngx.arg[1]
           -- 如果响应体匹配到了 2ooo，就让 ngx.arg[2]=true
```

```
            if string.match(body_chunk, "2ooo") then
                ngx.arg[2] = true
                return
            end
            -- 设置 ngx.arg[1] = nil，表示此响应体不会出现在输出内容中
            ngx.arg[1] = nil

        }
        echo '1oooKkk';
        echo '2oooKkk';
        echo '3oooKkk';
    }
}
```

执行结果如下：

```
#  curl -i http://testnginx.com/?file_name=1.lua
HTTP/1.1 200 OK
Server: nginx/1.12.2
Date: Tue, 29 May 2018 11:48:52 GMT
Content-Type: application/octet-stream
Transfer-Encoding: chunked
Connection: keep-alive

2oooKkk
```

从输出结果可以得出如下结论。

- 没有输出 1oooKkk，是因为它在 if 语句中匹配不成功，设置 ngx.arg[1] = nil 将没有匹配成功的数据屏蔽了。
- 没有输出 3oooKkk，是因为 2oooKkk 被匹配成功了，使用 ngx.arg[2] = true 终止了后续的操作，剩下的内容就不会再被输出了。

8.9.3　被禁用的 Lua API 指令

在 body_filter_by_lua_block 阶段中，下列 Lua API 是被禁止的。

- 输出类型的 API 函数（如 ngx.say 和 ngx.send_headers）。
- 控制类型的 API 函数（如 ngx.redirect 和 ngx.exec）。
- 子请求的 API 函数（如 ngx.location.capture 和 ngx.location.capture_multi）。
- cosocket API 函数（如 ngx.socket.tcp 和 ngx.req.socket）。

8.10 log_by_lua_block

8.10.1 阶段说明

语法：log_by_lua_block { lua-script }

环境：http、server、location、location if

阶段：log

含义：在日志请求处理阶段执行的{ lua-script }代码。它不会替换当前 access 请求的日志，而会运行在 access 的前面。

log_by_lua_block 阶段非常适合用来对日志进行定制化处理，且可以实现日志的集群化维护（详见第 12 章）。另外，此阶段属于 log 阶段，这时，请求已经返回到客户端，对 Ngx_Lua 代码的异常影响要小很多。

示例如下：

```
server {
    listen       80;
    server_name  testnginx.com;

    location / {
        log_by_lua_block {
            local ngx = require "ngx";
            xxxsada  -- 一个明显的语法错误
            ngx.log(ngx.ERR, 'x');
        }
        echo 'ok';
    }
}
```

上述代码的 error_log 虽有报错，但执行结果仍会输出“ok”。

8.10.2 被禁用的 Lua API 指令

在 log_by_lua_block 阶段中，下列 Lua API 是被禁止的。

- 输出类型的 API 函数（如 ngx.say 和 ngx.send_headers）。
- 控制类型的 API 函数（如 ngx.redirect 和 ngx.exec）。
- 子请求的 API 函数（如 ngx.location.capture 和 ngx.location.capture_multi）。
- cosocket API 函数（如 ngx.socket.tcp 和 ngx.req.socket）。
- 休眠 API 函数 ngx.sleep。

8.11　Lua 和 ngx.ssl

Lua API 还可以对 HTTPS 协议进行处理，可以使用 ssl_certificate_by_lua_block、ssl_session_fetch_by_lua_block、ssl_session_store_by_lua_block 这 3 个指令块进行配置，这 3 个指令块都涉及 lua-resty-core 模块的 ngx.ssl 指令，该指令超出了本章所讲的内容范围，有兴趣的读者可以自行查看 lua-resty-core 的相关资料。

8.12　Ngx_Lua 执行阶段

通过前面的学习，读者已经了解了 Ngx_Lua 模块各个执行阶段的作用，现在将这些执行阶段汇总到一起来观察一下整体的执行流程。

图 8-3 所示为 Ngx_Lua 的执行顺序，它引用自 Lua-Nginx-Module 的官方 Wiki，很清晰地展示了 Ngx_Lua 在 Nginx 中的执行顺序。下面举一个例子来验证一下这个执行顺序。

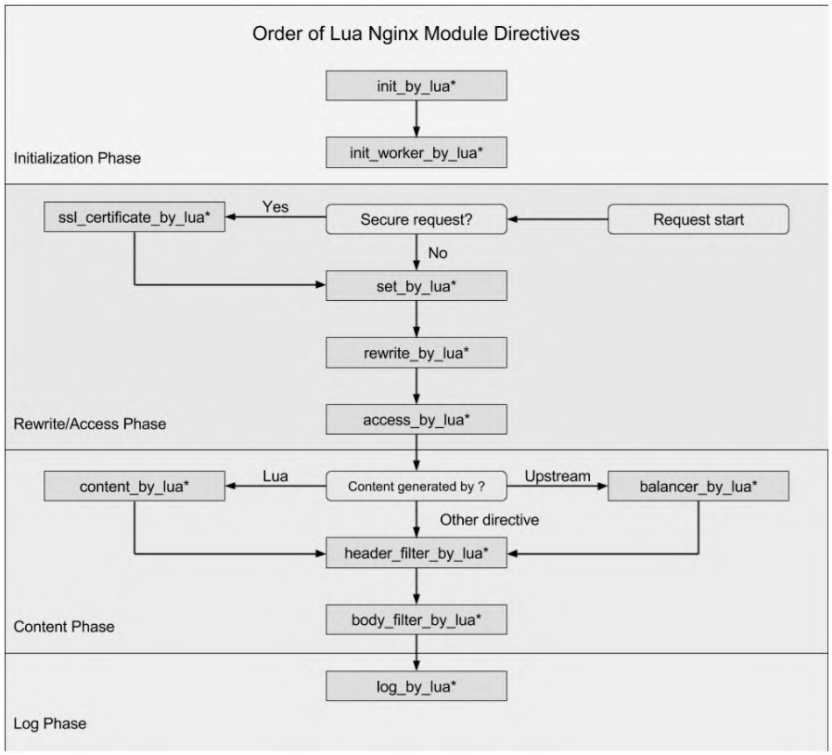


图 8-3　Ngx_Lua 的执行顺序

1．测试代码使用 *_by_lua_file 指令来配置，首先，建立如下的 test.lua 文件：

```
# vim /usr/local/nginx_1.12.2/conf/lua/test.lua
 local ngx  = require "ngx"
-- ngx.get_phase()可以获取 Lua 代码在运行时属于哪个执行阶段
 local phase = ngx.get_phase()
 ngx.log(ngx.ERR, phase, ':   Hello,world!')
```

2．在 Nginx 配置中载入如下文件：

```
# init_by_lua_file 和 init_worker_by_lua_file 只能在 http 指令块中执行
init_by_lua_file  /usr/local/nginx_1.12.2/conf/lua/test.lua;
init_worker_by_lua_file  /usr/local/nginx_1.12.2/conf/lua/test.lua;

server {
    listen       80;
    server_name  testnginx.com;

    location / {
        #文件的顺序是随意摆放的，观察日志，留意文件顺序是否对执行顺序有干扰
        body_filter_by_lua_file
/usr/local/nginx_1.12.2/conf/lua/test.lua;
        header_filter_by_lua_file
/usr/local/nginx_1.12.2/conf/lua/test.lua;
        rewrite_by_lua_file  /usr/local/nginx_1.12.2/conf/lua/test.lua;
        access_by_lua_file  /usr/local/nginx_1.12.2/conf/lua/test.lua;
        set_by_lua_file $test /usr/local/nginx_1.12.2/conf/lua/test.lua;
        log_by_lua_file  /usr/local/nginx_1.12.2/conf/lua/test.lua;
        # content_by_lua_file 和 balancer_by_lua_block 不能存在于同一个
        # location，所以只用 content_by_lua_file 示例
        content_by_lua_file  /usr/local/nginx_1.12.2/conf/lua/test.lua;
    }
}
```

3．重载 Nginx 配置让代码生效，此时不要访问 Nginx，可以观察到 error.log 在没有请求访问的情况下也会有内容输出。

如果输出“init”，表示此阶段属于 init_by_lua_block 阶段；如果输出“init_worker”，表示此阶段属于 init_worker_by_lua_block 阶段。

有些读者可能会发现 init_worker 的输出有 3 行，那是因为此阶段是在 Nginx 的 worker 进程上执行的，每个 worker 进程都会独立运行一次 Lua 代码，因此可以看出此时 Nginx 启动了 3 个 worker 进程，如下所示：

```
   2018/06/04  19:00:23  [error]  12034#12034:  [lua]  test.lua:3:  init:
Hello,world!
   2018/06/04  19:00:23  [error]  21019#21019:  *914  [lua]  test.lua:3:
init_worker:  Hello,world!, context: init_worker_by_lua*
   2018/06/04  19:00:23  [error]  21020#21020:  *915  [lua]  test.lua:3:
init_worker:  Hello,world!, context: init_worker_by_lua*
   2018/06/04  19:00:23  [error]  21018#21018:  *916  [lua]  test.lua:3:
init_worker:  Hello,world!, context: init_worker_by_lua*
```

4．发送请求：

```
   curl -i http://testnginx.com/
   HTTP/1.1 200 OK
   Server: nginx/1.12.2
   Date: Mon, 04 Jun 2018 11:00:29 GMT
   Content-Type: application/octet-stream
   Transfer-Encoding: chunked
   Connection: keep-alive
```

5．观察 access.log 日志，能够看到每个执行阶段的优先级顺序：

```
   2018/06/04 19:16:20 [error] 21019#21019: *920 [lua] test.lua:3: set:
Hello,world!, client: 10.19.64.210, server: testnginx.com, request: "GET
/test?ss HTTP/1.1", host: "testnginx.com"
   2018/06/04 19:16:20 [error] 21019#21019: *920 [lua] test.lua:3: rewrite:
Hello,world!, client: 10.19.64.210, server: testnginx.com, request: "GET
/test?ss HTTP/1.1", host: "testnginx.com"
   2018/06/04 19:16:20 [error] 21019#21019: *920 [lua] test.lua:3: access:
Hello,world!, client: 10.19.64.210, server: testnginx.com, request: "GET
/test?ss HTTP/1.1", host: "testnginx.com"
   2018/06/04 19:16:20 [error] 21019#21019: *920 [lua] test.lua:3: content:
Hello,world!, client: 10.19.64.210, server: testnginx.com, request: "GET
/test?ss HTTP/1.1", host: "testnginx.com"
   2018/06/04  19:16:20  [error]  21019#21019:  *920  [lua]  test.lua:3:
header_filter:  Hello,world!, client: 10.19.64.210, server: testnginx.com,
request: "GET /test?ss HTTP/1.1", host: "testnginx.com"
   2018/06/04  19:16:20  [error]  21019#21019:  *920  [lua]  test.lua:3:
body_filter:  Hello,world!, client: 10.19.64.210, server: testnginx.com,
request: "GET /test?ss HTTP/1.1", host: "testnginx.com"
   2018/06/04 19:16:20 [error] 21019#21019: *920 [lua] test.lua:3: log:
Hello,world!  while  logging  request,  client:  10.19.64.210,  server:
testnginx.com, request: "GET /test?ss HTTP/1.1", host: "testnginx.com"
```

通过 access.log 日志截图（如图 8-4 所示）可以看得更清晰。

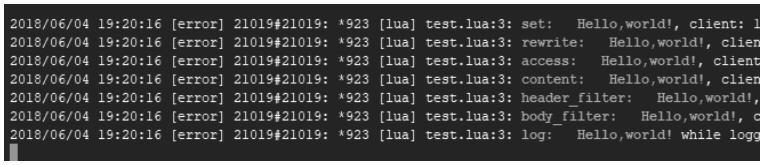


图 8-4　access.log 日志截图

观察 Ngx_Lua 整体的执行流程，可得出如下结论。

- Ngx_Lua 执行顺序如下：

```
init_by_lua_block
init_worker_by_lua_block
set_by_lua_block
rewrite_by_lua_block
access_by_lua_block
content_by_lua_block
header_filter_by_lua_block
body_filter_by_lua_block
log_by_lua_block
```

- 指令在配置文件中的顺序不会影响执行的顺序，这一点和 Nginx 一样。
- http 阶段的 init_by_lua_block 和 init_worker_by_lua_block 只会在配置重载时执行，在处理 HTTP 请求的过程中是不会被执行的。

注意：init_by_lua_file 在 lua_code_cache 为 off 的情况下，每次执行 HTTP 请求都会执行重载配置阶段的 Lua 代码。

8.13　小结

本章对 Ngx_Lua 的执行阶段进行了讲解，掌握每个执行阶段的用途会让大家在实际开发中更加得心应手。

第 9 章
Nginx 与数据库的交互

通常情况下，我们更习惯使用 Python、PHP、Java 等流行的编程语言和数据库打交道，很少会有人想到使用 Nginx 和数据库进行交互，但这个少数派方案，却拥有非常出色的表现，这也使 Nginx 在各大互联网公司“登台”的机会越来越多。本章将以 MySQL 和 Redis 为例，对 Nginx 和数据库的交互进行讲解。

注意：如果读者使用 Nginx 源码进行编译，则需要安装相关模块；如果使用 OpenResty，就不需要进行安装了。这也是 OpenResty 能作为 Web 开发一大利器的原因之一，只要是精良的模块它都会尽收囊中。

9.1 安装 cjson

在和数据打交道的过程中，会经常用到 JSON 格式，Ngx_Lua 中最常用的 JSON 解析库是使用 C 语言开发的 cjson，其安装方式如下：

```
    #     wget     https://www.kyne.com.au/~mark/software/download/lua-cjson-
2.1.0.tar.gz
    # tar -zxvf lua-cjson-2.1.0.tar.gz
    # cd lua-cjson-2.1.0
    # make
    # cp cjson.so /usr/local/nginx_1.12.2/conf/c_package/
```

需要注意的是，因为 Ngx_Lua 安装时使用的是 LuaJIT，不是原生的 Lua，所以执行 make 时可能会出现如下报错：

```
    # make
    cc  -c  -O3  -Wall  -pedantic  -DNDEBUG    -I/usr/local/include  -fpic  -o
lua_cjson.o lua_cjson.c
```

```
lua_cjson.c:43:17: error: lua.h: No such file or directory
lua_cjson.c:44:21: error: lauxlib.h: No such file or directory
lua_cjson.c:192: error: expected ')' before '*' token
lua_cjson.c:206: error: expected ')' before '*' token
lua_cjson.c:218: error: expected ')' before '*' token
lua_cjson.c:237: error: expected ')' before '*' token
lua_cjson.c:266: error: expected ')' before '*' token
lua_cjson.c:279: error: expected ')' before '*' token
lua_cjson.c:288: error: expected ')' before '*' token
lua_cjson.c:296: error: expected ')' before '*' token
lua_cjson.c:304: error: expected ')' before '*' token
```

修正方式是在刚刚安装的 lua-cjson-2.1.0 的目录下编辑 Makefile 文件，找到包含“LUA_INCLUDE_DIR=”的那行代码，修改配置为如下所示（指向 LuaJIT 的 include 地址）：

```
##### Build defaults #####
LUA_VERSION =       5.1
TARGET =            cjson.so
PREFIX =            /usr/local
#CFLAGS =            -g -Wall -pedantic -fno-inline
CFLAGS =            -O3 -Wall -pedantic -DNDEBUG
CJSON_CFLAGS =      -fpic
CJSON_LDFLAGS =     -shared
LUA_INCLUDE_DIR =   $(PREFIX)/include/luajit-2.1
LUA_CMODULE_DIR =   $(PREFIX)/lib/lua/$(LUA_VERSION)
LUA_MODULE_DIR =    $(PREFIX)/share/lua/$(LUA_VERSION)
LUA_BIN_DIR =       $(PREFIX)/bin
```

然后，重新执行 make，会在当前目录生成一个 csjon.so 文件，将 csjon.so 文件复制到 lua_package_cpath 所配置的地址，如 lua_package_cpath "${prefix}conf/c_package/?.so;;"。

现在可以对数据进行处理了，示例如下：

```
location = /test1 {
    content_by_lua_block {
        local  cjson = require "cjson"
        ngx.say(cjson.encode{a = 1, b = 2, c =3 })
    }
}
```

执行结果如下：

```
# curl  'http://testnginx.com/test1'
{"b":2,"a":1,"c":3}
```

9.2　与 MySQL 交互

在 Nginx 和 MySQL 交互时，需要使用 lua-resty-mysql 模块，该模块采用的是非阻塞 worker 进程的方式。

9.2.1　安装 lua-resty-mysql 模块

下载最新的 lua-resty-mysql releases 版本，解压后，在 lua_package_path 搜索路径下新建一个 resty 目录，将 lib 目录下的 mysql.lua 文件复制到 resty 目录下。

如果要加载 lua-resty-mysql 模块，则需要配置 resty 目录。例如，要加载 mysql.lua 模块，则配置为 local mysql = require "resty.mysql"。

```
   #  wget  -S  https://codeload.github.com/openresty/lua-resty-mysql/tar.gz/
v0.21 -O lua-resty-mysql-0.21.tar.gz
   # tar -zxvf lua-resty-mysql-0.21.tar.gz
   #cp              lua-resty-mysql-0.21/lib/resty/mysql.lua             \
/usr/local/nginx_1.12.2/conf/lua_modules/resty/
```

9.2.2　读取 MySQL 数据

下面就可以使用 lua-resty-mysql 来读取 MySQL 数据了，常用的配置方式如下所示：

```
   lua_package_path "${prefix}conf/lua_modules/?.lua;;";
   lua_package_cpath "${prefix}conf/c_package/?.so;;";

   server {
       listen       80;
       server_name  testnginx.com;
       default_type 'text/plain';

       location = /test {
           content_by_lua_block {
               local mysql = require "resty.mysql";
               local db, err = mysql:new();
               if not db then
                 ngx.say("failed to instantiate mysql: ", err);
                 return
               end
               -- 设置超时时间为 1s
               db:set_timeout(1000) ;
               -- 连接 MySQL 数据库
               local ok, err, errcode, sqlstate = db:connect{
```

```
                    host = "10.19.40.113",
                    port = 3306,
                    database = "clairvoyant",
                    user = "ngx_test",
                    password = "ngx_test",
                    charset = "utf8",
                    max_packet_size = 2048 * 2048
                }
                -- 如果连接失败，则输出异常信息
                if not ok then
                  ngx.say("failed to connect: ", err, ": ", errcode, " ",
sqlstate);
                  return
                end

                -- 执行 SQL 语句
                local res, err, errcode, sqlstate =
                  db:query("select id,host from  nginx_resource limit 2")

                -- 如果没有获取到数据，则输出异常信息
                if not res then
                  ngx.say("bad result: ", err, ": ", errcode, ": ",
                          sqlstate, ".")
                  return
                end

                -- 获取到的 res 是 table 类型的数据，通过 cjson 把它格式化为 JSON 类型
                local cjson = require "cjson";
                ngx.say("result: ", type(res));
                ngx.say("result: ", cjson.encode(res));

                -- 配置连接池
                -- 设置连接池的大小为 50，最大的空闲时间是 10s
                -- 如果一个连接超过 10s 没有使用就会被放入连接池中
                local ok, err = db:set_keepalive(10000, 50)
                if not ok then
                  ngx.say("failed to set keepalive: ", err);
                  return
                end
            }
        }
    }
```

执行结果如下：

```
    # curl 'http://testnginx.com/test'
    result: table
    result:
[{"host":"shop.zhe800.com","id":703},{"host":"shop.zhe800.com","id":705}]
```

下面是对常见指令用途的说明。

指令：syntax: db, err = mysql:new()

含义：创建 MySQL 的连接对象，如果创建失败，则返回 nil，并将错误信息以字符串形式赋值给 err。

指令：syntax: ok, err, errcode, sqlstate = db:connect(options)

含义：MySQL 的配置信息，具体配置清单如下。

- host：MySQL 的 IP 地址。
- port：MySQL 的启动端口。
- path：MySQL 监听的 UNIX socket 文件地址。
- database：MySQL 的数据库名。
- user：MySQL 的用户名。
- password：MySQL 的密码。
- charset：lua-resty-mysql 连接 MySQL 使用的字符集，它可以和 MySQL 的默认值不同，支持 big5、dec8、cp850、hp8、koi8r、latin1、latin2、swe7、ascii、ujis、sjis、hebrew、tis620、euckr、koi8u、gb2312、greek、cp1250、gbk、latin5、armscii8、utf8、ucs2、cp866、keybcs2、macce、macroman、cp852、latin7、utf8mb4、cp1251、utf16、utf16le、cp1256、cp1257、utf32、 binary、geostd8、cp932、eucjpms、gb18030 等字符格式。
- max_packet_size：设置 lua-resty-mysql 从 MySQL 读取内容的字节上限，默认是 1MB。
- compact_arrays：设置为 true 时，查询结果将返回 array-of-arrays 数组类型。

例如要将 compact_arrays = true 加入上述配置中，重载 Nginx 配置后，result 的数据格式会发生如下变化：

```
    # curl   'http://testnginx.com/test'
    result: table
    result: [[703,"shop.zhe800.com"],[705,"shop.zhe800.com"]]
```

指令：set_timeout

含义：设置读取 MySQL 数据的超时保护，包括对 connect 方法超时的设置，单位是毫

秒。如果连接 MySQL 的时间超过了 set_timeout 的值，就会报错。

指令：syntax: ok, err = db:set_keepalive(max_idle_timeout, pool_size)

含义：配置连接池，第 1 个参数单位是毫秒，第 2 个参数是连接池的数量。每个 worker 进程使用的连接池是独立的，假如设置 pool_size=20，且有 2 个 worker 进程，那么整体上，连接池的数量是 20×2=40。如果使用连接池时出现了异常，错误信息会被写入 err。

指令：syntax: times, err = db:get_reused_times()

含义：调用此指令可以返回当前连接的重用次数，从连接池取出的连接的重用次数一般大于 0；如果等于 0，则代表这是一个新的连接，并非来自连接池。

指令：syntax: bytes, err = db:send_query(query)

含义：用于发送命令到 MySQL 数据库，但不会等待结果返回，如果发送失败会将错误信息赋值给 err。

指令：syntax: res, err, errcode, sqlstate = db:read_result()

含义：读取 MySQL 数据返回的结果，如果是多条 SQL 语句执行的结果集，且 err 的值是 again，则表明还有数据没有被读取出来，此时可以执行 db:read_result 方法来获取下一条结果，直到 again 不再出现为止。如果有错误信息则被存放于 err，而 errcode 存放的是 MySQL 错误代码。

如果发送了多条 SQL 语句，但是只想获取前几条的执行结果，可以在 read_result()的括号里写上想要获取的条数，表示只返回该数量的结果，如 db:read_result(3)，表示只获取 3 条 SQL 语句返回的结果。

指令：syntax: res, err, errcode, sqlstate = db:query(query)

含义：如果请求执行成功，会将返回的内容赋值给 res；如果有错误，错误信息会存放于 err；errcode 存放的是 MySQL 的错误代码。该指令也可以对返回数据的数量进行限制，如只返回 2 条数据的话，示例如下：

```
   local res, err, errcode, sqlstate = db:query("select id,host from nginx_resource ", 2)
```

当然，更建议使用 MySQL 的 limit，因为这样更规范、合理。

指令：syntax: ok, err = db:close()

含义：关闭当前 MySQL 的连接并返回是否关闭成功的状态，一般在短连接中使用，不能和连接池配置同时使用，示例如下：

```
    local ok, err = db:set_keepalive(10000, 50)
```

9.2.3　执行多条 SQL 语句

利用 db:read_result 的功能，可以发送多条 SQL 语句到 MySQL 中去执行，示例如下：

```
    location = /test {
        content_by_lua_block {
            local mysql = require "resty.mysql";
            local db, err = mysql:new();
            if not db then
              ngx.say("failed to instantiate mysql: ", err);
              return
            end
            -- 设置超时时间为 3s
            db:set_timeout(3000) ;
            local ok, err, errcode, sqlstate = db:connect{
               host = "10.19.40.113",
               port = 3306,
               database = "clairvoyant",
               user = "ngx_test",
               password = "ngx_test",
               charset = "utf8",
               max_packet_size = 2048 * 2048
            }
            -- 如果连接失败，则输出异常信息
            if not ok then
              ngx.say("failed to connect: ", err, ": ", errcode, " ",
sqlstate);
              return
            end

            -- 执行 SQL 语句，有 4 条 SQL 语句，请注意不要忘记末尾的分号
            sql1 = 'select id,host from  nginx_resource limit 1;';
            sql2 = 'select id from  nginx_resource limit 1;';
            sql3 = 'select host from  nginx_resource limit 1;';
            sql4 = 'select host from  nginx_resource limit 2;';

            -- SQL 语句用..连接合并
            local res, err, errcode, sqlstate =
            db:query(sql1 .. sql2 .. sql3 .. sql4)

            -- 如果没有获取到数据，则会输出异常信息，读者可以写一条错误的 SQL 语句
            -- 观察 error.log 日志的变化
```

```
            if not res then
               ngx.log(ngx.ERR, "bad result #1: ", err, ": ", errcode, ": ",
sqlstate, ".")
               return ngx.exit(500)
            end
            local cjson = require "cjson";
            -- 输出第一条返回的结果
            ngx.say("result sql: " , cjson.encode(res))
            -- 循环读取请求结果，如果 err 的值是 again，则继续循环执行
            while err == "again" do
                res, err, errcode, sqlstate = db:read_result()
                if not res then
                ngx.log(ngx.ERR, "bad result sql",  ": ", err, ": ",
                errcode, ": ", sqlstate, ".")
                return ngx.exit(500)
                end
             -- 输出剩下的 SQL 语句的执行结果
                ngx.say("result sql", ": ", cjson.encode(res))
            end

            -- 连接池配置
            -- 设置连接池的大小为 50，最大闲置时间 10000ms
            local ok, err = db:set_keepalive(10000, 50)
            if not ok then
              ngx.say("failed to set keepalive: ", err);
              return
            end
        }
    }
```

执行结果如下：

```
# curl 'http://testnginx.com/test'
result sql: [{"host":"shop.zhe800.com","id":703}]
result sql: [{"id":703}]
result sql: [{"host":"shop.zhe800.com"}]
result sql: [{"host":"shop.zhe800.com"},{"host":"shop.zhe800.com"}]
```

从执行结果可知，当执行多条 SQL 语句时，会按照先后顺序返回结果，请求须等执行全部完成后才能一起返回给客户端。但若执行某个 SQL 语句时返回的时间超过了db:set_timeout(3000)设置的值，该语句就会被放弃，请求只会返回已经执行完成的 SQL 语句。

读者可以尝试将上述配置中的 sql2，改为“select sleep(4);”，观察输出发生的变化。

9.2.4　防止 SQL 注入

在 Ngx_Lua 模块中可以使用指令 ngx.quote_sql_str 来防止 SQL 注入。

指令：ngx.quote_sql_str

语法：quoted_value = ngx.quote_sql_str(raw_value)

环境：set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、balancer_by_lua*、ssl_certificate_by_lua*、ssl_session_fetch_by_lua*、ssl_session_store_by_lua*

含义：根据 MySQL 的转义规则返回一个转换后的字符串。

例如，对 URL 中的参数 a 进行转义，代码如下：

```
    local a = ngx.quote_sql_str(ngx.var.arg_a)
    loal sql = 'select id,host from  nginx_resource where test= ' .. a
    local res, err, errcode, sqlstate = db:query(sql)
```

9.3　与 Redis 交互

在 Nginx 和 Redis 交互时，常用的模块有 lua-resty-redis 和 redis2-nginx-module，前者基于 Ngx_Lua 开发和使用；后者是 C 模块，需要在 Nginx 中进行源码编译。两者各有千秋，但都不会阻塞 Nginx 的 worker 进程。本节将以 lua-resty-redis 为核心进行讲解，因为它可以在 Ngx_Lua 的很多阶段中使用，比 redis2-nginx-module 更为灵活。

9.3.1　安装 lua-resty-redis

lua-resty-redis 和 lua-resty-mysql 的安装方式类似，复制 lib 文件夹到 lua_package_path 指定的搜索路径下即可：

```
    # git clone https://github.com/openresty/lua-resty-redis.git
    # cp lua-resty-redis/lib/resty/redis.lua \ /usr/local/nginx_1.12.2/conf/lua_modules/resty/
```

9.3.2　读/写 Redis

使用 lua-resty-redis 读取 Redis 数据，常用配置示例如下：

```
    location = /test {
        content_by_lua_block {
```

```
            local redis = require "resty.redis"

            -- 创建一个 Redis 对象
            local red = redis:new()

        -- 设置超时时间为 1s，包含连接 Redis 和返回数据的时间，和 lua-resty-mysql 一样
            red:set_timeout(1000)

            -- 连接 Redis
            local ok, err = red:connect("127.0.0.1", 6379)
            if not ok then
                ngx.say("failed to connect: ", err)
                return
            end
            -- 发送 set 命令给 Redis
            ok, err = red:set("test", "nginx")
            if not ok then
                ngx.say("failed to set test: ", err)
                return
            end
            local key_1 = 'test'
            -- 发送 get 命令给 Redis，将返回的数据赋值给 res
            local res, err = red:get(key_1)
            if not res then
                ngx.say("failed to get test: ", err)
                return
            end
            -- 如果没有查询到 key，则返回的值会是 ngx.null
            if res == ngx.null then
                ngx.say(key_1," not found.")
                return
            end
            -- 将内容返回给客户端
            ngx.say("test: ", res)

            -- 设置连接池的大小为 100，最大空闲时间为 10s，和 lua-resty-mysql 一样
            local ok, err = red:set_keepalive(10000, 100)
            if not ok then
                ngx.say("failed to set keepalive: ", err)
                return
            end
            }
        }
```

执行结果如下：

```
# curl 'http://testnginx.com/test'
test: nginx
```

读者会发现 lua-resty-redis 和 lua-resty-mysql 有很多相同的配置和参数，因为它们都是基于 Ngx_Lua 的 cosocket API 开发的，拥有很多相同的特性。

下面是对 lua-resty-redis 常见指令用途的说明。

指令：syntax: red, err = redis:new()

含义：创建 Redis 的连接对象，如果创建失败，会返回 nil，并将错误信息以字符串形式赋值给 err。

指令：syntax: ok, err = red:connect(host, port, options_table?)

syntax: ok, err = red:connect("unix:/path/to/unix.sock", options_table?)

含义：Redis 的配置信息，常用配置如下所示。

- host：Redis 的 IP 地址。
- port：Redis 的启动端口。
- options_table：可选参数，一般和连接池的命名有关，日常使用较少。
- unix:/path/to/unix.sock：Redis 启动的 socket 套接字。

指令：syntax: ok, err = red:close()

含义：关闭当前 Redis 的连接并返回是否关闭成功的状态，一般在短连接中使用，使用时需要去掉连接池配置 local ok, err = red:set_keepalive(10000, 100)。

指令：syntax: times, err = red:get_reused_times()

含义：此方法可以返回当前连接的重用次数，从连接池取出的连接的重用次数一般大于 0；如果等于 0，则可以理解为这是一个新的连接，并非来自连接池。

9.3.3　管道命令

利用管理命令，可以在一次交互中给 Redis 发送多条命令，这样可以减少 TCP 建立连接的次数，以提升性能，示例如下：

```
location = /test {
    content_by_lua_block {
        local redis = require "resty.redis"
        -- 创建一个 Redis 对象
```

```
local red = redis:new()

-- 设置超时时间为 1s,连接和返回的时间都包含在内
red:set_timeout(1000)

-- 连接 Redis
local ok, err = red:connect("127.0.0.1", 6379)
if not ok then
    ngx.say("failed to connect: ", err)
    return
end
-- 激活 pipeline 模式
red:init_pipeline()
red:set("a", "1")
red:set("b", "2")
-- 此处写了一个错误命令
red:hset("a s s")
red:get("a")
red:get("b")
red:get("c")

-- 将上面的命令使用 pipeline 执行
local results, err = red:commit_pipeline()
if not results then
    ngx.say("failed to commit the pipelined requests: ", err)
    return
end

-- 返回的结果是 table 类型的，需要循环读取数据
for i, res in ipairs(results) do
    if type(res) == "table" then
        -- 如果命令失败，则 res[1]的值为 false
        if res[1] == false then
            ngx.say("failed to run command ", i, ": ", res[2])
        else
        end
    else
      -- 输出结果
      ngx.say('type:',type(res),'---res:',res)
    end
end

local ok, err = red:set_keepalive(10000, 100)
if not ok then
```

```
            ngx.say("failed to set keepalive: ", err)
            return
        end

        }
    }
}
```

在上述配置中，先使用 red:init_pipeline()指令开启了 pipeline 模式，然后将需要执行的操作依次写入配置，最后使用 red:commit_pipeline()进行提交。

执行结果如下：

```
# curl   'http://testnginx.com/test'
type:string---res: OK
type:string---res: OK
failed to run command 3: ERR wrong number of arguments for 'hset' command
type:string---res: 1
type:string---res: 2
type:userdata---res: null
```

从执行结果可以看出如下几点。

- 发送的 set 命令，返回结果都是 ok。
- 错误的 hset 命令执行后会将错误信息返回，但不会影响后续命令的执行。
- 如果 get 命令查询为空，则会出现 null。

9.3.4　密码登录

有时为了安全需要设置 Redis 密码，可以在上述配置中加入如下配置来完成密码登录：

```
-- 连接 Redis
local ok, err = red:connect("127.0.0.1", 6379)
if not ok then
    ngx.say("failed to connect: ", err)
    return
end
-- 在 red:connect 后面加入密码访问的配置
local res, err = red:auth("testpasswd")
if not res then
    ngx.say("failed to authenticate: ", err)
    return
end
```

9.3.5 其他执行命令

Redis 还支持丰富的指令来完成数据的读取，那么如何在 lua-resty-redis 中使用这些指令呢？lua-resty-redis 和 redis_cli（Redis 原生的客户端执行命令行）指令格式的对比见表 9-1。

表 9-1 lua-resty-redis 与 redis_cli 指令格式的对比

lua-resty-redis 指令格式	redis_cli 指令格式
red:get("a")	get a
red:set("a", "1")	set a 1
red:hmset("mshash", "test1", "a", "test2", "b")	mshash test1 "a" test2 "b"

从表 9-6 中可以看出，在操作同一个指令时，lua-resty-redis 与原生的 Redis 命令相比，只是在格式上有细微的变化。在 lua-resty-redis 中，变量是以逗号分隔的；而在 redis_cli 命令行中，用空格来分隔变量。如果需要在 lua-resty-redis 上使用其他的 Redis 指令，则按照上面的格式进行配置即可。

9.4 与数据库交互的常见问题

9.4.1 连接池

在前面的例子中，默认都使用了 local ok, err = red:set_keepalive(10000, 100)来配置连接池，而没有使用 ok, err = db:close()短连接配置。因为如果请求比较频繁，短连接会造成额外的 TCP 开销，所以除非请求 Redis 的频率非常低，否则不建议使用短连接。

通过 get_reused_times()指令可以判断当前的连接是否来自连接池，如果不是，则 get_reused_times()的值为 0。当数据库需要账号和密码才可以访问时，如果 get_reused_times()的值大于 0 则可以认为已经登录，复用这个连接时就不需要再次输入密码了，性能也会得到提升。可以通过下面的方式测试长连接是否还需要再次输入密码，以 Redis 连接为例：

```
location = /test {
    content_by_lua_block {

        local redis = require "resty.redis"

        -- 创建一个 Redis 对象
        local red = redis:new()

        -- 设置超时时间为 1s，连接和返回的时间都包含在内
        red:set_timeout(1000)
```

```
-- 连接 Redis
local ok, err = red:connect("127.0.0.1", 6379)
if not ok then
    ngx.say("failed to connect: ", err)
    return
end
-- 登录有密码的 Redis
local res, err = red:auth("testpasswd")
if not res then
    ngx.say("failed to authenticate: ", err)
    return
end
-- 获取当前连接的使用次数
local count, err = red:get_reused_times()
ngx.say('get_reused_times: ',count)

red:init_pipeline()
red:get("a")
red:get("b")
local results, err = red:commit_pipeline()
if not results then
    ngx.say("failed to commit the pipelined requests: ", err)
    return
end
for i, res in ipairs(results) do
    if type(res) == "table" then
        -- 如果命令失败，则 res[1]的值为 false
        if res[1] == false then
            ngx.say("failed to run command ", i, ": ", res[2])
        else
        end
    else
      -- 输出结果
      ngx.say('type: ',type(res),'---res: ',res)
    end
end
local ok, err = red:set_keepalive(10000, 10)
if not ok then
    ngx.say("failed to set keepalive: ", err)
    return
end

}
```

```
    }
```

从客户端发出请求，执行结果如下：

```
# curl   'http://testnginx.com/test'
get_reused_times: 0
type:string---res: 1
type:string---res: 2
[root@testnginx ~]# curl   'http://testnginx.com/test'
get_reused_times: 1
type:string---res: 1
type:string---res: 2
[root@testnginx ~]# curl   'http://testnginx.com/test'
get_reused_times: 2
type:string---res: 1
type:string---res: 2
[root@testnginx ~]# curl   'http://testnginx.com/test'
get_reused_times: 3
type:string---res: 1
type:string---res: 2
[root@testnginx ~]# curl   'http://testnginx.com/test'
get_reused_times: 4
type:string---res: 1
type:string---res: 2
```

观察执行结果可以看到 get_reused_times()的值会随着访问的增加而增加，如果停止访问，等待 10s 后再次访问，该值又会重新从 0 开始计数，因为 set_keepalive 设置的闲置时间是 10s，如果一个连接超过 10s 无人访问就会被回收，再次访问时会使用新的连接。

在长连接还保持的情况下（即上一次请求和下一次请求间隔时间不超过 10s 的情况下），修改 Redis 的密码。观察执行结果，会发现密码虽然被修改，但之前 get_reused_times 非 0 的请求仍然在和 Redis 进行交互，只有当 get_reused_times=0 时，新连接才会报错，错误信息如下所示：

```
# curl   'http://testnginx.com/test'
failed to authenticate: ERR invalid password
```

这一点也可以证明，只有新连接被创建时才会用到密码，正在使用中的长连接是不用再次输入密码的。

9.4.2　读/写分离

在实际开发中，经常会用读/写分离的功能来提升服务的性能，特别是当读操作比较频繁时，可以配置多个从库来分担读操作带来的压力。下面是一个读取多个从库的示例，通过初始化两个 Redis 连接（主库和从库），将读/写命令进行分离：

```
local cjson = require "cjson"
local redis = require "resty.redis"
-- 初始化一个主库连接
local red = redis:new()
red:set_timeout(1000)
local ok, err = red:connect("127.0.0.1", 6379)
if not ok then
    ngx.say("failed to connect: ", err)
    return
end
-- 初始化一个从库连接
local red_slave1 = redis:new()
red_slave1:set_timeout(1000)
local ok, err = red_slave1:connect("127.0.0.1", 6380)
if not ok then
    ngx.say("failed to connect: ", err)
    return
end
```

在进行读/写操作时，根据操作的不同选择需使用的 Redis 连接即可。

9.4.3　分离配置文件和代码

读者应该已经发现，在上面的配置中，业务代码和数据库配置信息杂糅在一起，当有很多业务代码都要去读取数据库中的数据时，每个业务代码中都会加入相同的数据库配置信息，这会导致代码的可读性很差。我们可以将数据库配置信息和业务代码进行分离，推荐使用 require 来加载配置文件。

首先，新增一个 db_config.lua，按照 Lua 语言中的模块的格式来声明数据库配置：

```
-- 创建一个 table 类型的 _M
local _M = {}
-- 将 MySQL 的配置存入 _M 中
local _M.mysql_config = {
    host = "10.19.40.113",
    port = 3306,
    database = "clairvoyant",
    user = "ngx_test",
```

```
        password = "ngx_test",
        charset = "utf8",
        max_packet_size = 2048 * 2048
    }
    return _M
```

然后，在业务代码中 require 这个配置文件，通过这种方式将数据库配置信息从代码中分离出来：

```
    -- 导入配置文件
    local db_config = require "db_config"
    -- 直接使用 mysql_config 的配置文件初始化配置
    local ok, err, errcode, sqlstate = db:connect(db_config.mysql_config)
```

9.5 小结

本章对 Nginx 与 MySQL、Redis 的交互分别进行了介绍，基于 Ngx_Lua 的特有功能，Nginx 还支持 Couchbase、MongoDB、Memcached 等主流数据库，而且在与这些数据库交互时性能都非常出色，有兴趣的读者可以试一试。

第 10 章
缓存利器

先来看一张简单的缓存流程图，如图 10-1 所示。

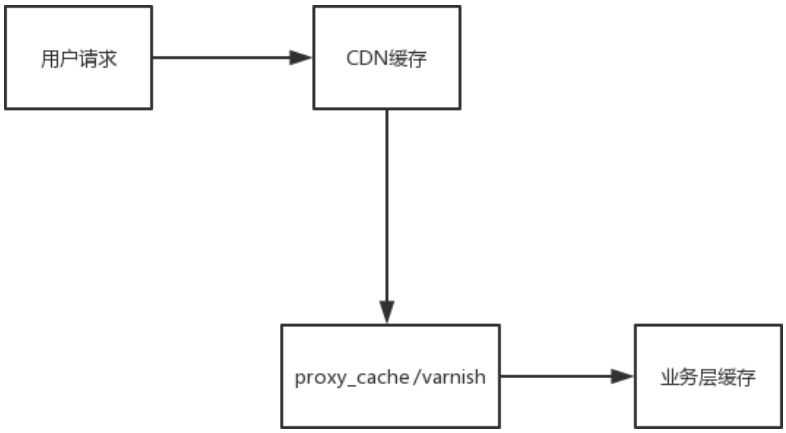


图 10-1　缓存流程图

CDN 缓存和 proxy_cache 在前面的章节中已经有过介绍，它们可以提升访问速度，减少回源流量，从而减少后端业务层的压力。

在业务层中，也有不少服务有自己的缓存，这些缓存既可以减少服务自身的运算量，也可以减轻后端数据库等服务的压力。常见的业务层缓存工具有 Redis、Memcached、Couchbase 等。它们可以用来完成大量的业务数据缓存和计算，除这些工具外，Ngx_Lua 还提供了一套内部缓存系统，它是 key/value 类型的，有个很大的优点就是不会产生额外的 TCP 开销，请求在内部的缓存系统就能完成数据的读取。在数据结构或业务需求不复杂的情况下，使用 Ngx_Lua 内部缓存将是一个非常好的选择。

10.1 worker 进程的共享内存

worker 进程的共享内存，顾名思义就是 Nginx 的全部 worker 进程都共享这份内存数据，如果某个 worker 进程对该数据进行了修改，其他的 worker 进程看到的都将是修改后的数据。我们可以利用共享内存来存放缓存数据，提升吞吐率，减少到达后端的请求次数。

10.1.1 创建共享内存区域

在使用共享内存之前，需要先创建 1 块内存区域，即声明共享内存的大小和名字。这要用到指令 lua_shared_dict。

语法：lua_shared_dict <name> <size>

默认：无

环境：http

含义：声明一个名字为 name，大小为 size 的共享内存区域，用来存放基于 Lua 字典的共享内存。

例如，声明一个名字为 shared_test、内存空间为 1MB 的共享内存区域，如下所示：

```
lua_shared_dict shared_test 1M;
```

lua_shared_dict 的参数 size 支持 KB 和 MB 两种单位，并且允许设置的最小值为 12KB，如果小于 12KB 启动就会报错。当 size 的值小于 12KB 且大于 8KB 时，报错如下：

```
nginx: [crit] ngx_slab_alloc() failed: no memory
```

当 size 的值小于 8KB 时，则报错内容如下：

```
nginx: [emerg] invalid lua shared dict size "2k" in /usr/local/nginx_
1.12.2/conf/nginx.conf
```

通过 lua_shared_dict 声明了多少内存，Nginx 就会在系统中占用多少内存。例如，如果声明一个 500MB 的共享内存区域，那么，当重载 Nginx 配置后，观察系统的内存使用情况，会发现 Nginx 的每个 worker 进程的内存占用量都在 500MB 以上（500MB 共享内存+Nginx 本身要占用的内存，会超过 500MB）。

在配置共享内存区域的数据时须注意如下几点。

1．共享内存区域的数据是所有 worker 进程共享的，因此一旦对其进行修改，所有 worker 进程都会使用修改后的数据，worker 进程之间同时读取或修改同一个共享内存数据会产生锁。

2．共享内存区域的大小是预先分配的，如果内存空间使用完了，会根据 LRU（Least

Recently Used，即最近最少使用）算法淘汰访问量少的数据。

3．在 Nginx 中可以配置多个共享内存区域。

10.1.2　操作共享内存

创建共享内存区域后，可以尝试对其进行读/写操作，示例如下：

```
-- 声明共享内存
lua_shared_dict shared_test 1m;
server {
    listen 80;
    server_name testnginx.com;
    location /set {
        content_by_lua_block {
            -- 获取指定共享内存的 Lua 字典对象
            local shared_test = ngx.shared.shared_test
            -- 在共享内存里存储 URL 参数 a 的值，类似于 Redis 的 set 指令
            shared_test:set("a",ngx.var.arg_a)
            ngx.say("STORED")
        }
    }
    location /get {
        content_by_lua_block {
            -- 获取指定共享内存的 Lua 字典对象
            local shared_test = ngx.shared.shared_test
            -- 读取共享内存的数据，获取 a 的值，类似于 Redis 的 get 指令
            ngx.say(shared_test:get("a"))
        }
    }
}
```

在上述配置中，先将参数 a=123 传递给/set 的 location，由 set 指令完成存储，然后再请求/get，读取共享内存中 a 的值，执行结果如下：

```
# curl   'http://testnginx.com/set?a=123'
STORED
# curl   'http://testnginx.com/get?a'
123
```

Ngx_Lua 提供了很多操作共享内存的指令，下面将会介绍一些常用的指令。

注意：下面的 DICT 是变量，它指的是 lua_shared_dict 声明的共享内存区域的名字。

指令：ngx.shared.DICT

语法：dict = ngx.shared.DICT 或 dict = ngx.shared[name_var]

环境：init_by_lua*、init_worker_by_lua*、set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua* 、 header_filter_by_lua* 、 body_filter_by_lua* 、 log_by_lua* 、 ngx.timer.* ，balancer_by_ lua*、ssl_certificate_by_lua*、ssl_session_fetch_by_lua*、ssl_session_store_by_lua*

含义：获取指定的共享内存（即 Lua 字典）中的数据。

指令：ngx.shared.DICT.set

语法：success, err, forcible = ngx.shared.DICT:set(key, value, exptime?, flags?)

含义：设置 key/value 的值，并存入共享内存中，其中 value 可以是 Lua 布尔值、数字、字符串，或 nil 类型的，不可以是 table 类型的。

该请求会返回如下 3 个值。

- success：布尔值，用来表示是否成功 set。
- err：如果 success 的值是 false，就会将错误信息存放在 err 中。
- forcible：布尔值，用来表示是否有数据被强制删除。在存储数据的过程中，如果内存空间不够就会使用 LRU 算法清除数据，此时，forcible 的值为 true。

除 key/value 外，set 中其他的参数作用如下。

- exptime 为可选参数，作用是设置 key/value 的有效期（单位是秒），默认是不过期的。
- flags 为可选参数，用来做用户标识，可以标识 set 操作的执行者，默认是 0。只能设置 number 类型。例如，可以使用开发人员的工位号作为 flags，这样就可以知道每条 set 指令的操作者了，方便后期排查问题。

指令：ngx.shared.DICT.safe_set

语法：ok, err = ngx.shared.DICT:safe_set(key, value, exptime?, flags?)

含义：与 set 类似，但在内存不足的情况下，它不会覆盖已经存在且没有过期的数据，而会报“no memory”错误并返回 nil。

指令：ngx.shared.DICT.get

语法：value, flags = ngx.shared.DICT:get(key)

含义：获取指定 key 对应的 value，value 的数据类型与存储时的原始数据类型一致，但如果 key 不存在或过期就返回 nil。

flags 用来获取是否对用户做了标识，一般在 set 时都会加入标识，如果没有，则 flags 默认为 0。

指令：ngx.shared.DICT.get_stale

语法：value, flags, stale = ngx.shared.DICT:get_stale(key)

含义：与 get 类似，但返回的是过期的值，过期的值无法保证一直存在，如当内存不够时会被自动清理。

flags，获取标识，一般在 set 时会加入此标识，如果没有，则默认为 0。

stale，代表获取的 value 是否已经过期，如果是，则返回 true。

指令：ngx.shared.DICT.add

语法：success, err, forcible = ngx.shared.DICT:add(key, value, exptime?, flags?)

含义：与 set 类似，但如果 key 在缓存里存在且没有过期就不会执行 add 指令，此时 err 会返回“exist”；如果 key 不存在或已过期，则会执行 add 指令并把 key/value 的值存入共享内存中。

指令：ngx.shared.DICT.safe_add

语法：ok, err = ngx.shared.DICT:safe_add(key, value, exptime?, flags?)

含义：与 set 类似，但在内存不足的情况下，它不会覆盖已经存在且没有过期的数据，而会报“no memory”错误并返回 nil。

指令：ngx.shared.DICT.replace

语法：success, err, forcible = ngx.shared.DICT:replace(key, value, exptime?, flags?)

含义：和 set 类似，但只有当 key 存在时，才会把 key/value 的值存入共享内存中（key 过期属于不存在的情况）。

指令：ngx.shared.DICT.delete

语法：ngx.shared.DICT:delete(key)

含义：删除指定的 key。

指令：ngx.shared.DICT.incr

语法：newval, err, forcible? = ngx.shared.DICT:incr(key, value, init?)

含义：让指定 key 的值递增，步长是 value。适合用来在请求时做计数操作。newval 是数字累加后的值。如果 key 不存在，并且 init 参数为空，则请求会返回“not found”并赋值给 err；如果 key 不存在，并且 init 是 number 类型，那么会创建一个 init+value 的新 key。

注意：这里的 key、value、init，在正常情况下都应该是 number 类型。

指令：ngx.shared.DICT.flush_all

语法：ngx.shared.DICT:flush_all()

含义：清空共享内存中的所有数据，方法是让所有的数据都过期，这意味着虽然可以用 get_stale 的方式去读取数据，但不一定会有值存在。

指令：ngx.shared.DICT.flush_expired

语法：flushed = ngx.shared.DICT:flush_expired(max_count?)

含义：清空共享内存中所有过期的数据，方法是释放掉这些内存，彻底清除数据，并返回所清除数据的数量。

max_count 为可选参数，如果不设置此参数或设为 0，则会清除全部过期数据；如果参数值大于 0，则会清除相应数量的数据。

指令：ngx.shared.DICT.get_keys

语法：keys = ngx.shared.DICT:get_keys(max_count?)

含义：获取共享内存中的全部数据，返回的是 table 类型的数据。

max_count 设置的是需要返回的数据的数量，默认是 1024。如果设置为 0，则表示返回全部数据。在实际运用中要尽量避免设置为 0，因为如果数据量非常大，此操作会阻塞所有访问这个共享内存数据的 worker 进程。

在实践中需要特别注意的地方有如下几点。

1．在存储 key/value 时：

如果 set 的 forcible = true 或 safe_set 的 err = "no memory"，则可以判断出共享内存的空间不够了，需要进行异常报警。

2．在获取 key/value 时：

get 和 get_stale 中的 flags 来自 set 指令的添加，在进行团队开发时建议不要使用默认值，而要定制每个开发人员的 flags。

3．关于 incr 自增长的初始化值：

如果把对数据的初始化放在 init_by_lua*阶段执行，那么每次重载 Nginx 配置时都会导致数据再次被初始化。

4．在获取全部 key 的数据时：

get_keys 默认值是 1024，如果无法确定 key 的数量，则千万不要使用 get_keys(0)来读取 key，因为这就像在 MySQL 中执行没有 limit 的 select * from table 一样。

针对上述指令，示例代码如下：

```
location /set {
    content_by_lua_block {
        local shared_test = ngx.shared.shared_test
        -- 自增长操作
        local newval, err = shared_test:incr("incr_init_n",1,2)
        ngx.say("newval:",newval, " err:",err)  -- 输出 newval:3 err:nil
        shared_test:set("a",ngx.var.arg_a,100,2001)
        shared_test:set("b",ngx.var.arg_a,100,2001)
        local  success, err, forcible = shared_test:set("d",
        ngx.var.arg_a, 100,2001)
        -- 输出 success:true err:nil forcible:false
        ngx.say("success:",success, " err:",err, " forcible:",forcible)
        local  ok, err = shared_test:safe_set("c",ngx.var.arg_a,100,
        2001)ngx.say("ok:",ok," err:",err)       -- 输出 ok:true err:nil
        local value, flags, stale = shared_test:get_stale("a")
        -- 输出 value:123s flags:2001 stale:false
        -- 因为值没有过期，所以 stale 为 false
        ngx.say("value:",value," flags:",flags," stale:",stale)
        -- 读取共享里面的 2 个 key
        local getall_key = shared_test:get_keys(2)
        if type(getall_key) == 'table' then
           ngx.say(#getall_key)                  -- 输出 key 的数量
           for _, k in ipairs(getall_key) do
                                                 -- 输出这 2 个数据
             -- key:b value:123s2001  key:d value:123s2001
             ngx.say("key:",k ," value:",shared_test:get(k))
           end
        end
        shared_test:flush_all()                  -- 清除共享内存中的数据
        local getall_key = shared_test:get_keys()
        -- 判断有几个 key，输出为 type : table，llen_list: 0
        ngx.say("type: ", type(getall_key),  " llen_list: ",#getall_key)
    }
}
```

10.1.3　制造消息队列

ngx.shared.DICT 还支持消息队列功能，与消息队列相关的指令及其使用方式见表 10-1。

表 10-1　与消息队列相关的指令及其使用方式

指　令	说　明
length, err = ngx.shared.DICT:lpush(key, value)	将 string 或 number 类型的 value 从头部插入 key 的列表中，并返回列表的元素数量。如果 key 不存在，则先创建 key；如果 key 存在，但不是列表，则返回 nil，且 err 会显示为"value not a list"

续表

指　令	说　明
length, err = ngx.shared.DICT:rpush(key, value)	和 lpush 类似，只是元素会从尾部插入
val, err = ngx.shared.DICT:lpop(key)	移除 key 列表中的第 1 个值，并将这个值返回给 val。如果 key 不存在，则返回 nil；如果 key 存在，但不是列表，则会返回 nil 且 err 显示为"value not a list"
val, err = ngx.shared.DICT:rpop(key)	和 lpop 类似，只从尾部删除元素
syntax: len, err = ngx.shared.DICT:llen(key)	返回共享字典的 key 的列表的元素数量。如果 key 不存在，则返回 0；如果 key 存在，但不是列表，则会返回 nil 且 err 的值显示为"value not a list"

示例如下：

```
lua_shared_dict shared_test_1 1m;
server {
    listen 80;
    server_name testnginx.com;
    location /lpush {
        content_by_lua_block {
            local shared_test_1 = ngx.shared.shared_test_1
            -- 每次请求都会产生 1 个自增的值，可以把它作为测试数据，当作每次 lpush 的值
            local newval, err = shared_test_1:incr("incr_init_n",1,0)
            -- 将自增的值从列表的头部插入，每次插入都会返回当前列表的长度 length
            local length, err = shared_test_1:lpush("push_abc", newval)
            -- 输出当前请求时，显示队列当前的长度和插入的值
            ngx.say("length:",length," lpush_value:",newval)
        }
    }
    location /lpop {
        content_by_lua_block {
            local shared_test_1 = ngx.shared.shared_test_1
            -- 从队列的末尾取出数据并存放在 val 上
            local val, err = ngx.shared.shared_test_1:lpop("push_abc")
            -- 获取当前 key 的队列长度
            local len, err = ngx.shared.shared_test_1:llen("push_abc")
            -- 输出当前请求的队列长度和值
            ngx.say("length:",len, " val:" ,val)
        }
    }
}
```

发送 HTTP 请求进行队列数据的存取，如下所示：

```
# curl   'http://testnginx.com/lpush'
length:1 lpush_value:1
```

```
[root@testnginx ~]# curl   'http://testnginx.com/lpush'
length:2 lpush_value:2
[root@testnginx ~]# curl   'http://testnginx.com/lpush'
length:3 lpush_value:3
[root@testnginx ~]# curl   'http://testnginx.com/lpop'
length:2 val:3
[root@testnginx ~]# curl   'http://testnginx.com/lpop'
length:1 val:2
[root@testnginx ~]# curl   'http://testnginx.com/lpop'
length:0 val:1
[root@testnginx ~]# curl   'http://testnginx.com/lpop'
length:0 val:nil
```

上述示例是在当前 key 不存在或为空的情况下进行的测试，分析其操作步骤如下。

1．先从列表的头部插入 3 条数据，输出显示列表的长度和值都在增加。

2．再从尾部取出 4 条数据，从数据的输出顺序可以看出数据是从尾部取出来的，因为 val 的值是按照 3、2、1 的顺序输出的。

3．最后 1 条输出为 nil，因为数据被取完了。

上面只是一个示例，如果在真实的开发环境下做 pop 操作，不断地发送 HTTP 请求肯定不是首选，可以使用如下方式解决这个问题。

- 启动一个定时任务，让它不断去 pop 数据，以减少外部 HTTP 建立连接所产生的开销。
- 根据业务的实时性进行判断，评估消息的生成速度，以此来确定定时任务之间的间隔时间，定时任务可以在毫秒级别执行，采用 for 循环读取数据，每次取固定数量的数据进行处理。

10.1.4　lua-resty-core

当对共享内存中的 key 设置有效期后，需要使用与 lua-resty-core 模块有关的指令才可以获取有效期，但使用 lua-resty-core 模块需要安装很多依赖包，并且这些包的版本必须一致，否则会出现不兼容的问题，所以建议使用 OpenResty，因为它已经包含了这些安装包。

下面是与 lua-resty-core 有关的指令。

指令：ngx.shared.DICT.ttl

语法：ttl, err = ngx.shared.DICT:ttl(key)

含义：获取共享内存中的 key 的有效期。

指令：ngx.shared.DICT.expire

语法：success, err = ngx.shared.DICT:expire(key, exptime)

含义：修改共享内存中的 key 的有效期。

指令：ngx.shared.DICT.free_space

语法：free_page_bytes = ngx.shared.DICT:free_space()

含义：确认共享内存的剩余空间大小。

10.1.5 配置环境

共享内存 ngx.shared.DICT.*的指令的执行环境都是一样的，它们的配置环境分别如下。

1． dict = ngx.shared.DICT

支持的环境有 init_by_ lua*、init_worker_by_lua*、set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、balancer_by_lua*、ssl_certificate_by_lua*、ssl_session_fetch _by_lua*、ssl_session_store_by_lua*。

2．ngx.shared.DICT:get_stale 和 ngx.shared.DICT:get

不支持在 init_by_lua*, init_worker_by_lua*中执行。

支持的环境有 set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、balancer_by_lua*、ssl_certificate_by_lua*、ssl_session_fetch _by_lua*、ssl_session_store_by_lua*。

3．其他的 ngx.shared.DICT.*指令

不支持在 init_worker_by_lua*中执行。

支持的环境有 init_by_lua*、set_by_lua*、rewrite_by_lua*、access_by_lua*、content_by_lua*、header_filter_by_lua*、body_filter_by_lua*、log_by_lua*、ngx.timer.*、balancer_by_lua*、ssl_certificate_by_lua*、ssl_session_fetch _by_lua*、ssl_session_store_by_lua*。

共享内存提供了很多类似于 Redis 的操作指令，这些指令可以基本满足日常的开发需求，并且能够减少网络的 I/O 资源的消耗，缓存的数据也可以在每个执行阶段被方便地使用。在本书后面的架构设计上会大量使用共享内存来实现动态化配置。

10.2　Lua 模块下的共享内存

10.1 节讲到了 worker 进程的共享内存，它利用丰富的指令使数据的缓存操作变得非常简单，但也存在一些缺点。

1．worker 进程之间会有锁竞争，在高并发的情况下会增加性能开销。

2．只支持 Lua 布尔值、数字、字符串和 nil 类型的数据，无法支持 table 类型的数据。

3．在读取数据时有反序列化操作，会增加 CPU 资源的开销。

但瑕不掩瑜，共享内存在 Ngx_Lua 中作为缓存工具还是非常出色的。笔者在生产环境中，曾多次使用 lua_shared_dict 的各种特性，并未感受到存在明显的性能问题。但如果读者还是介意这些缺点或需要缓存更复杂的数据的话，可以使用 lua-resty-lrucache。

10.2.1　安装 lua-resty-lrucache

lua-resty-lrucache 是基于 Ngx_Lua 的缓存利器，它拥有如下优点。

1．支持更丰富的数据类型，可以把 table 存放在 value 中，这对数据结构复杂的业务非常有用。

2．可以预先分配 key 的数量，不用设置固定的内存空间，在内存的使用上更为灵活。

3．每个 worker 进程独立缓存，所以当 worker 进程同时读取同一个 key 时不存在锁竞争。

但它与 lua_shared_dict 相比也有一些缺点。

1．因为数据不在 worker 之间共享，所以无法保证在更新数据时，数据在同一时间的不同 worker 进程上完全一致。

2．虽然可以支持复杂的数据结构，但可使用的指令却很少，如不支持消息队列功能。

3．重载 Nginx 配置时，缓存数据会丢失。如果使用 lua_shared_dict，则不会如此。

有利就有弊，读者在使用时可以根据自身需求进行选择。lua-resty-lrucache 的安装方式和其他的 lua-resty 模块一样，如下所示：

```
# git clone https://github.com/openresty/lua-resty-lrucache.git
# cp -r lua-resty-lrucache/lib/resty/lrucache* \
 /usr/local/nginx_1.12.2/conf/lua_modules/resty/
```

10.2.2　使用 lua-resty-lrucache 进行缓存的方法

通过下面的例子来了解一下 lua-resty-lrucache 的使用方式，首先需要对模块进行加载，方

法如下：

```
local lrucache = require "resty.lrucache"
local lrucache = require "resty.lrucache.pureffi"
```

读者在加载 lua-resty-lrucache 时，需要把上面的 2 个文件复制到 lua_package_path 所设置的路径上。它们的作用是一样的，但性能有所区别：resty.lrucache 适合用来缓存命中率高或读操作远远大于写操作的缓存业务，resty.lrucache.pureffi 适合用来缓存命中率低或需要对 key 进行频繁增、删操作的缓存业务。请根据业务需求进行选择。

然后，将下面的代码写入 test_m.lua 中，并将此文件存放到 lua_package_path 所设置的路径下，代码如下：

```
local _M = {}
local lrucache = require "resty.lrucache"
-- 在缓存上声明一个拥有 1000 个 key 的列表
local cache, err = lrucache.new(1000)
if not cache then
    return error("failed to create the cache: " .. (err or "unknown"))
end
-- 此函数用来往缓存中存储 key/value 的值
local function mem_set()
    -- set()中的内容从左到右顺序依次是 key、value、有效期（2s）
    cache:set("a", 19, 2)
    cache:set("b", {"1","2","3"},0.001)  -- 支持插入 table 类型的数据
    return
end
-- 此函数用来获取缓存里的 value。 a 即 value 的值，如果 a 为 nil
-- 则表示 value 不存在或已过期；如果 stale_data 有值，也说明 value 已过期
local function mem_get(key)
    local a,stale_data = cache:get(key)
    return a,stale_data
end
function _M. fromcache ()
    -- 获取 a 的值
local a,stale_data  =  mem_get("a")
    -- 如果 a 存在，就输出 a 的值
    if a then
       ngx.say("a: ", a)
    -- 如果 a 不存在且 stale_data 有值，就输出过期的 value
    -- 并重新执行存储操作，然后再次输出 value
    elseif stale_data then
       ngx.say("a 已经过期: " , stale_data)
       mem_set()
```

```
        local a_again  =  mem_get("a")
        ngx.say("a: ", a_again )
    -- 如果 a 和 stale_data 都不存在，则执行存储操作后再输出 value
    else
        ngx.say("no found a")
        mem_set()
        local a_again  =  mem_get("a")
        ngx.say("a: ", a_again )

    end
end
return _M
```

修改 nginx.conf 文件，代码如下：

```
location / {
    content_by_lua_block {
        -- 加载模块，执行数据的读取操作
        require("test_m").fromcache()
    }
}
```

重载 Nginx 配置，执行结果如下：

```
# curl   'http://testnginx.com/'
no found a
a: 19
[root@testnginx ~]# curl   'http://testnginx.com/'
a: 19
[root@testnginx ~]# curl   'http://testnginx.com/'
a: 19
[root@testnginx ~]# curl   'http://testnginx.com/'
a 已经过期: 19
a: 19
```

从执行结果可以看出：

1．第 1 次请求，因为 a 没有值，所以先输出“no found a”，然后又执行了存储操作。

2．第 2 次请求，因为有缓存值，直接输出 value。

3．第 3 次请求，仍然有缓存值，直接输出 value。

4．第 4 次请求，因为为缓存数据设置的有效期很短，此时已经过期，所以输出了过期的 value，并再次执行存储操作，又输出了 value。

如果尝试重载 Nginx 配置，会发现每次重启（restart）后 a 都没有值，因为在重载配置的

过程中，缓存数据会丢失。

下面将对 lua-resty-lrucache 的常见指令进行说明。

指令：new

语法：cache, err = lrucache.new(max_items [, load_factor])

含义：创建一个缓存实例。如果创建失败则会返回 nil，并将错误信息返回给 err。

max_items 用来声明缓存 key 的数量，从这个设置可以看出它虽然没有规定内存的使用大小，但规定了 key 的数量。

load_factor 参数是加载 resty.lrucache.pureffi 模块时才会用到的，它基于 FFI（Foreign Function Interface，外部功能接口）的 hash 表的负载因子，值的区间在 0.1~1 之间，默认值是 0.5。负载因子与 hash 表数据的读取时间和对内存空间大小的权衡有关，有兴趣的读者可以自行查询相关信息。

指令：set

语法：cache:set(key, value, ttl)

含义：把 key/value 存储到缓存中。ttl 是缓存的有效期，以秒为单位，默认值是 0，表示不会过期；支持设置为 0.001s。

指令：get

语法：data, stale_data = cache:get(key)

含义：获取指定 key 的值，如果 key 不存在或已过期，就返回 nil；如果存在过期数据，过期的值会赋值给 stale_data。

指令：delete

语法：cache:delete(key)

含义：从缓存中移除指定的 key。

指令：flush_all

语法：cache:flush_all(key)

含义：刷新整个缓存区域的数据，等于清空内存中的数据。这种方式比创建新的缓存实例要快得多。

10.3　当前请求在各执行阶段间的数据共享

10.1 节和 10.2 节介绍的都是全局的缓存系统，那么有没有只针对当前请求设置的缓存呢，即某个数据在请求的每个阶段都有缓存，但请求结束时缓存就会消失。

举例来说，请求在 rewrite 阶段生成一个缓存数据，作用是可以让后面的阶段（如 content 阶段）获取到该缓存数据，但该数据在请求完成后就没有用了，可以在请求结束后清除掉。

这种缓存主要针对如下情况使用，在 Ngx_Lua 中 Lua API 的执行是有阶段限制的，那么当某个不能执行 Lua API 的阶段需要使用 Lua API 指令生成数据时，就可以利用刚才介绍的缓存方式，在其他阶段处理好数据并缓存，在需要该数据的阶段调用此缓存数据即可。这里需要用到的缓存指令是 ngx.ctx。

10.3.1　ngx.ctx 的使用

指令：ngx.ctx

环境：init_worker_by_lua* 、set_by_lua* 、rewrite_by_lua* 、access_by_lua* 、content_by_lua* 、header_filter_by_lua* 、body_filter_by_lua* 、log_by_lua* 、ngx.timer.* 、balancer_by_lua*

含义：ngx.ctx 是 Lua 的 table 类型，用来缓存基于 Lua 的环境数据，该缓存在请求结束后会随之清空，类似于 Nginx 中 set 指令的效果。

示例如下：

```
rewrite_by_lua_block {
    -- 设置一个 test 数据
    ngx.ctx.test = 'nginx'
    ngx.ctx.test_1 = {a=1,b=2}

}
access_by_lua_block {
    -- 修改 test
    ngx.ctx.test = ngx.ctx.test .. ' hello'
}
content_by_lua_block {
    -- 输出 test 和 test_1 中的 a 元素的值
    ngx.say(ngx.ctx.test)
    ngx.say("a: ",ngx.ctx.test_1["a"])
}
header_filter_by_lua_block {
```

```
            -- 作为响应头输出
            ngx.header["test"] = ngx.ctx.test .. ' world!'
        }
    }
```

上述配置执行结果如下：

```
# curl -i  'http://testnginx.com/'
HTTP/1.1 200 OK
Server: nginx/1.12.2
Date: Mon, 18 Jun 2018 05:07:15 GMT
Content-Type: application/octet-stream
Transfer-Encoding: chunked
Connection: keep-alive
test: nginx hello world!

nginx hello
a: 1
```

从执行结果可知，ngx.ctx 的数据可以被 Lua 的每个执行阶段读/写，并且支持存储 table 类型的数据。

10.3.2 子请求和内部重定向的缓存区别

ngx.ctx 在子请求和内部重定向中的使用方法有些区别。在子请求中修改 ngx.ctx.* 的数据不会影响主请求中 ngx.ctx.*对应的数据，它们维护的是不同的版本，如执行子请求的 ngx.location.capture、ngx.location.capture_multi、echo_location 等指令时；在内部重定向中修改数据会破坏原始请求中 ngx.ctx.*的数据，新请求将会是一个空的 ngx.ctx table，如当 ngx.exec、rewrite 配合 last/break 使用时。

ngx.ctx 在内部重定向中使用的示例如下：

```
location /subq {
    header_filter_by_lua_block {
        -- 如果 ngx.ctx.test 不存在，则把 not test 赋值给 a
        local a = ngx.ctx.test or 'not test'
        -- 作为响应头输出
        ngx.header["test"] =  a .. ' world!'
    }
    content_by_lua_block {
        ngx.say(ngx.ctx.test)
    }
}
location / {
```

```
    header_filter_by_lua_block {
        ngx.header["test"] =  ' world!'
    }
    content_by_lua_block {
        ngx.ctx.test = "nginx"
        -- 执行内部重定向
        ngx.exec("/subq")
    }
}
```

执行结果如下：

```
# curl   -i 'http://testnginx.com/'
HTTP/1.1 200 OK
Server: nginx/1.12.2
Date: Mon, 18 Jun 2018 05:36:38 GMT
Content-Type: application/octet-stream
Transfer-Encoding: chunked
Connection: keep-alive
test: not test world!

nil
```

从执行结果可知，ngx.ctx.test 在内部重定向后变为空。

注意：查询 ngx.ctx 会产生元方法调用，这会消耗一定的服务器资源，所以如果数据可以通过函数生成，就尽量不要使用 ngx.ctx。如果希望更多地了解 ngx.ctx 对性能的影响，可以通过火焰图对它的性能进行分析（详见第 16 章）。

10.4　利用共享内存配置动态 IP 地址认证

在 8.3 节中曾提到过动态添加黑白名单的方案，下面对该内容做一个补充，代码如下：

```
lua_shared_dict white_list_ip 1m;    #声明存放白名单的共享内存
server {
    listen       80;
    server_name  testnginx.com;
    location / {
        access_by_lua_block {
            local ngx = require "ngx";
            local white_list_ip = ngx.shared.white_list_ip
            -- 获取请求者的 IP 地址，如果 Nginx 的前端还有其他代理或 CDN
            -- 则需要配置 real_ip 来获取真实的用户 IP 地址
            local ip  = ngx.var.remote_addr
```

```
            -- 在共享内存中查找请求的 IP 地址是否存在
            local value, flags = white_list_ip:get(ip)
            -- 如果查到的 IP 地址在白名单中，就退出 access_by_lua_block
            -- 继续执行下一个阶段
            if value then
                ngx.exit(ngx.OK)
            else
                ngx.exit(ngx.HTTP_FORBIDDEN)-如果不在白名单中，就返回 403 错误
            end
        }
        echo 'ok';
    }

  --提供 HTTP 接口，可以修改共享内存中的数据
     location = /white_list_ip_op {
         default_type 'text/plain';
         content_by_lua_block {
              local ngx = require "ngx"
              local white_list_ip = ngx.shared.white_list_ip
              local op = ngx.var.arg_op
              local ip = ngx.var.arg_ip
              -- 增加白名单
              if op == 'add' then
                  white_list_ip:set(ip,'1')
              -- 删除白名单
              elseif op == 'del' then
                  white_list_ip:delete(ip)
              end
              -- 执行完操作后，读取当前白名单的配置
              local ds = white_list_ip:get_keys()
              if type(ds) == 'table' then
                  for _, k in ipairs(ds) do
                      ngx.say("白名单 ip :  ",k)
                  end
              end
            }
      }

}
```

执行操作，添加一条白名单：

```
#  curl 'http://testnginx.com/white_list_ip_op?op=add&ip=10.19.64.210'
白名单 ip :  10.19.64.210
```

使用白名单中的 IP 地址 10.19.64.210 访问 testnginx.com，返回结果如下：

```
#  curl -i http://testnginx.com/
HTTP/1.1 200 OK
Server: nginx/1.12.2
Date: Tue, 29 May 2018 09:43:30 GMT
Content-Type: application/octet-stream
Transfer-Encoding: chunked
Connection: keep-alive

ok
```

删除白名单：

```
#  curl -i 'http://testnginx.com/white_list_ip_op?op=del&ip=10.19.64.210'
HTTP/1.1 200 OK
Server: nginx/1.12.2
Date: Tue, 29 May 2018 09:44:11 GMT
Content-Type: text/plain
Transfer-Encoding: chunked
Connection: keep-alive
```

使用白名单中的 IP 地址 10.19.64.210 再次访问 testnginx.com，会返回异常，并提示禁止访问，如下所示：

```
#  curl -i http://testnginx.com/
HTTP/1.1 403 Forbidden
Server: nginx/1.12.2
Date: Tue, 29 May 2018 09:45:21 GMT
Content-Type: text/html
Content-Length: 169
Connection: keep-alive

<html>
<head><title>403 Forbidden</title></head>
<body bgcolor="white">
<center><h1>403 Forbidden</h1></center>
<hr><center>nginx/1.12.2</center>
</body>
</html>
```

此方案存在一个隐患，即当 Nginx 的 master 进程重启后，共享内存中的数据会丢失，会导致黑白名单的防御失效。为了防止数据丢失，下面提供两种常见的解决方案。

- A 方案，将 IP 地址存放在 MySQL 上，再提供一个读/写 MySQL 的 Lua 函数，每次新增或删除共享内存中的数据时，都将修改后的数据存放到 MySQL 中。当 Nginx 的 master

进程重启后，在 init_worker_by_lua 阶段将数据从 MySQL 中读取出来插入共享内存即可。（本方案的使用方式在后续章节中会有示例。）

- B 方案，每次操作共享内存后，都将共享内存中的数据写入硬盘文件。当 Nginx 的 master 进程重启后，在 init_by_lua_block 阶段读取硬盘中的文件，将文件的 IP 地址重新插入共享内存即可。

10.5 缓存和数据库的交互

在实际开发中，常常会使用 NoSQL 缓存数据来减少 MySQL 的读取压力，同样，也可以利用 Ngx_Lua 的缓存来减少 MySQL 的压力，本节将介绍缓存和数据库的交互方案。

10.5.1 从数据库获取数据

从 MySQL 中获取数据后存放到 Ngx_Lua 缓存中，有多种实现方案。下面是比较常见的 3 种方案。

A 方案，适合在缓存的 key 较多时使用，图 10-2 所示为当 key 较多时的缓存流程图。

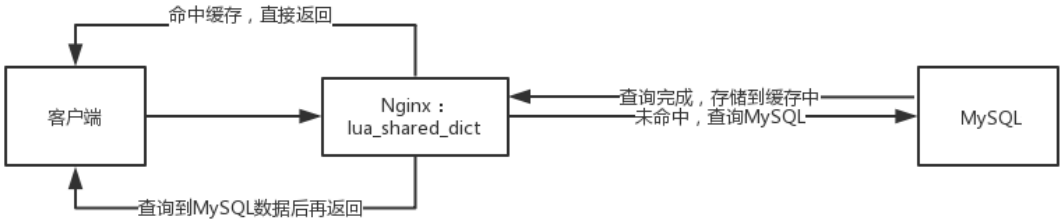


图 10-2　当 key 较多时的缓存流程图

B 方案，适合在缓存的 key 较少时使用，如图 10-3 所示为当 key 较少时的缓存流程图。

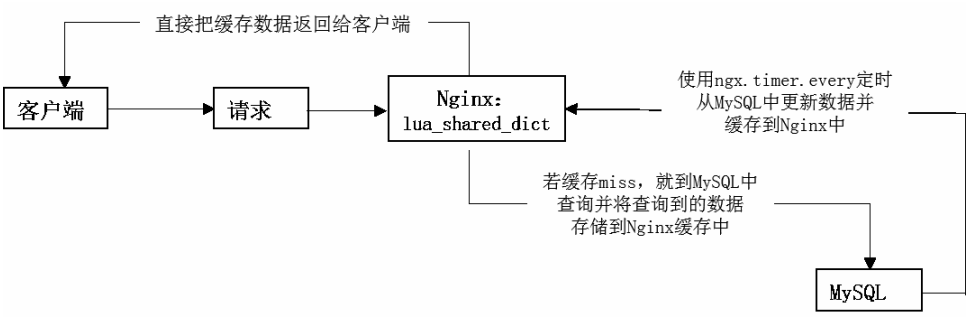


图 10-3　当 key 较少时的缓存流程图

C 方案，适合在缓存的 key 非常少时使用，会定期请求 Nginx 缓存来刷新接口，缓存刷新接口时会同步所有的数据，所以不会存在 miss 缓存的情况。客户端的请求只和 Nginx 缓存打交道，不直接访问 MySQL。图 10-4 所示为当 key 非常少时的缓存流程图。

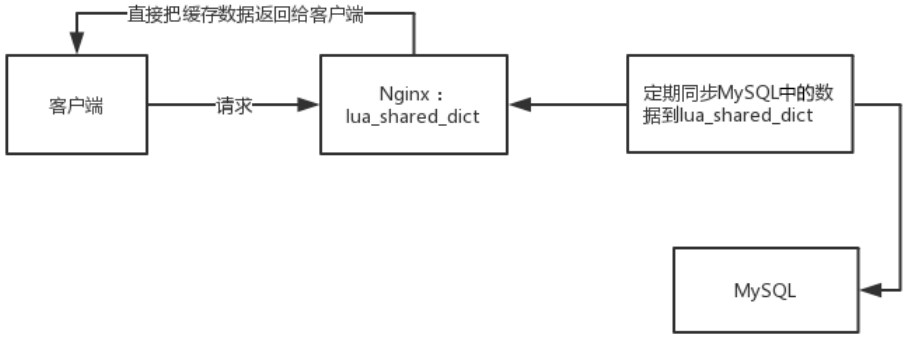


图 10-4　当 key 非常少时的缓存流程图

A 方案和 B 方案的主要区别在于，B 方案有定时任务，可以批量更新缓存中的数据，这样客户端请求一般就不会进入缓存未命中（缓存 miss）的流程了。C 方案和 B 方案的区别在于，在 C 方案中客户端不和 MySQL 数据库直接打交道。

这 3 种方案都用到了指令 lua_shared_dict，其实，使用 lua-resty-lrucache 也可以。下面就以 lua-resty-lrucache 为例来实现缓存与数据库交互的方案。

首先，创建 db_op 模块，用来读取 MySQL 数据。方法是将下面的代码写入 db_op.lua 文件中，并存放到 lua_package_path 路径下：

```
local _DB =  {}

-- 下面函数的主要任务是执行 SQL 语句，将数据提取出来
function _DB.getMySQL(sql)
            local MySQL = require "resty.MySQL";
            local db, err = MySQL:new();
            if not db then
              ngx.say("failed to instantiate MySQL: ", err);
              return
            end
            -- 设置超时时间为 5s
            db:set_timeout(5000) ;
            -- 连接 MySQL
            local ok, err, errcode, sqlstate = db:connect{
               host = "10.19.10.113",
               port = 3306,
```

```
                    database = "clairvoyant",
                    user = "ngx_test",
                    password = "ngx_test",
                    charset = "utf8",
                    max_packet_size = 2048 * 2048

                }
                -- 如果连接失败，则输出异常信息
                if not ok then
                  ngx.say("failed to connect: ", err, ": ", errcode, " ",
sqlstate);
                  return
                end

                -- 执行 SQL 语句
                local sql = sql
                local res, err, errcode, sqlstate =
                  db:query(sql)

                if not res then
                  ngx.say("bad result: ", err, ": ", errcode, ": ", sqlstate,
".")
                  return
                end

                ngx.log(ngx.ERR,db:get_reused_times(),err)
                local ok, err = db:set_keepalive(10000, 10)
                if not ok then
                  ngx.say("failed to set keepalive: ", err);
                  return
                end
                return res
    end

    return _DB
```

然后，创建 host_deny 模块，其主要作用是实现对 Ngx_Lua 中的数据的缓存，将下面的代码写入 host_deny.lua 文件中，并存放到 lua_package_path 的路径下：

```
    local _M = {}

    local lrucache = require "resty.lrucache"
    -- 载入 db_op 模块，用来传递 SQL 语句的参数
    local db_op = require("db_op")
    local cache, err = lrucache.new(1000) -- 声明一个可以缓存 1000 个 key 的列表
```

```
    if not cache then
        return error("failed to create the cache: " .. (err or "unknown"))
    end

    local function mem_set(host)
        -- 利用 Lua 的格式化功能，将参数 host 的值合并到 SQL 语句中
        local  sql  =  string.format([[select  sleep(3),host  from    nginx_
resource where host =  '%s'  limit 1]] , host)
        -- 执行 SQL 语句
        local res = db_op.getMySQL(sql)
        if type(res) == 'table' then
            for i, data  in ipairs(res) do
                -- 将读取到的数据插入共享内存中。'find'只是一个标识
                -- 也可以使用其他任意字符，重点是 key 是 host 要找的值
                cache:set(data["host"],'find',5)

            end
        end
        return
    end

    local function mem_get(host)
        local res_host,stale_data = cache:get(host)
        if res_host then
           return res_host
        elseif stale_data then
        -- 如果数据过期，仍然会读取数据，这在某些场景下是很有用的
        -- 例如当 MySQL 宕机时，它可以先提供过期数据来使用
           mem_set(host)
           res_host  = cache:get(host)
           return res_host

        else
        -- 没有数据，执行 SQL 语句后，再返回数据
           mem_set(host)
           res_host  = cache:get(host)
           return res_host
        end
    end
    function _M.fromcache(host)
        -- 在缓存中查找 URL 的 Host 头信息的值
        local res_host  =  mem_get(host)
        return res_host
    end
```

```
return _M
```

添加 Nginx 配置文件，根据请求访问的 Host 头信息设置白名单，以禁止某些域名的访问：

```
server {
    listen 80;
    location / {
        access_by_lua_block {
            -- 加载 host_deny 模块
            local host_deny = require "host_deny"
            local ngx = require "ngx"
            local host = ngx.var.host
            -- 使用 host_deny 模块的 fromcache 函数查询 host 是否在白名单中
            local white_host = host_deny.fromcache(host)

            -- 如果白名单中没有，就返回 403 错误
            if not white_host then
                ngx.exit(ngx.HTTP_FORBIDDEN)
            else
                ngx.exit(ngx.OK)
            end

        }
        content_by_lua_block {
            ngx.say("hello world!!!")
        }
    }
}
```

先使用一个不在白名单中的域名进行访问，返回 403 错误；再使用一个在白名单中的域名进行访问，返回 200，如下所示：

```
# curl   -i 'http://testnginx.com/' -H 'Host: a.test.com'
HTTP/1.1 403 Forbidden
Server: nginx/1.12.2
Date: Mon, 18 Jun 2018 09:24:04 GMT
Content-Type: text/html
Content-Length: 169
Connection: keep-alive

[root@testnginx ~]# curl -i 'http://testnginx.com/' -H 'Host:
shop.zhe800.com'
HTTP/1.1 200 OK
```

```
Server: nginx/1.12.2
Date: Mon, 18 Jun 2018 09:23:31 GMT
Content-Type: application/octet-stream
Transfer-Encoding: chunked
Connection: keep-alive

hello world!!!
```

此服务存在一个隐患，即当缓存 miss 过多，且有很多重复的请求时，会造成 MySQL 负担过大，从而产生不必要的资源消耗。下一节将会介绍使用锁机制来减少重复请求的方法。

10.5.2　避免出现因缓存失效引起的“风暴”

为了减少重复请求访问数据库的次数，可以使用 lua-resty-lock 模块，它提供加锁的方式去访问数据库，类似于之前讲到的 ngx_http_proxy_module 模块中的 proxy_cache_lock。

下面是在 Nginx 下安装 lua-resty-lock 的方法（OpenResty 不需要安装，默认已经支持）：

```
# wget -S https://codeload.github.com/openresty/lua-resty-lock/tar.gz/
v0.07 -O lua-resty-lock_0.07.tar.gz
# tar -zxvf lua-resty-lock_0.07.tar.gz
# cp lua-resty-lock-0.07/lib/resty/lock.lua \
  /usr/local/nginx_1.12.2/conf/lua_modules/resty
```

注意：如果使用 Nginx 进行开发，但又不打算用 resty.core 模块，则需要使用 lua-resty-lock 0.07 版本。因为大于这个版本的 lua-resty-lock 需要加载 resty.core 模块才可以使用。

模块的 Wiki 已经给出了很直观的例子供读者参考，地址为 https://github.com/ openresty/ lua-resty-lock。

以 10.5.1 节中的代码为例来实现锁机制，为了使锁操作看上去更明显，给 SQL 查询的请求加上了 sleep(3)，这样 MySQL 会等待 3s 后再返回数据，当缓存失效时，就可以看到锁的作用了，具体示例如下。

首先，需要有一个 db_op 模块来读取 MySQL 中的数据，db_op 模块的内容与 10.5.1 节的代码一样，这里不再赘述。然后，创建 host_deny.lua 模块，其内容如下：

```
local _M = {}

-- 载入 db_op 模块，用来传递 SQL 语句的参数
local db_op = require("db_op")

-- db_locks 的作用是存放锁的 key（每个锁都需要一个名字，key 就是锁的名字）
-- 的共享内存，cache 存放的是业务数据，也就是要读取的 key/value 的缓存数据
local db_locks=  ngx.shared.db_locks
```

```
    local cache=  ngx.shared.db_cache

    local function get_MySQL(host)
        -- 获取 MySQL 数据的配置文件没有太大的变化，因为锁操作并不在 MySQL 上
        -- SQL 语句会在 sleep 3s 后才输出，这样当缓存过期时，很多请求就会
        -- 等待 MySQL 的返回数据，从而形成锁的测试环境
        local  sql  =  string.format([[select  sleep(3),host  from    nginx_
resource where host =  '%s'  limit 1]] , host)
        local res = db_op.getMySQL(sql)
        if res[1] then
            local value = res[1]["host"] or nil
            return value
        end
        return nil
    end

    local function lock_db(key)
        -- 导入锁的模块
    local resty_lock = require "resty.lock"
        -- 创建锁的实例，db_locks 就是之前声明存放锁 key 的共享内存
        local lock, err = resty_lock:new("db_locks")
        if not lock then
            ngx.log(ngx.ERR,err)
            return nil,"failed to create lock: " .. err
        end
        -- 对要查询的 key 加锁
        local elapsed, err = lock:lock(key)
        if not elapsed then
            ngx.log(ngx.ERR,err)
            return nil,"failed to acquire the lock: " .. err
        end
        -- 记录当前请求等待锁时花费的时间
    ngx.log(ngx.ERR,elapsed)
        -- 再次查询缓存，因为在锁的过程中，可能前面某个请求已经获得了数据
        -- 并存放到了缓存中。如果没有，则继续执行查询
        local val, err = cache:get(key)
    if val then
        -- 如果获取到值，就释放锁
            local ok, err = lock:unlock()
            if not ok then
            ngx.log(ngx.ERR,err)
                return nil,"failed to unlock: " .. err
            end
            return val
```

```
    end
    -- 从 MySQL 中获取数据
    local val = get_MySQL(key)
if not val then
    -- 即使没有查询到数据，也要释放锁
        local ok, err = lock:unlock()
        if not ok then
        ngx.log(ngx.ERR,err)
            return nil,"failed to unlock: " .. err
        end
        -- 如果某个 key 一直被高并发访问，但在 MySQL 中却没有数据，请求就会
        -- 一直穿透缓存到 MySQL 中进行查询，特别是当服务被攻击时，并发会很高
        -- 这时可以设置一个不存在的值如 null 来缓存一段时间，以减少这种穿透现象的发生
        local ok,err = cache:set(key,'null',1)  -- 1 表示缓存时间是 1s
        return 'null'  -- 将字符串 null 返回，退出此函数
end
    -- 如果查询到 val，就对缓存进行存储
    local ok, err = cache:set(key, val,3)
if not ok then
    -- 即使 set 失败，也要释放锁
        local ok, err = lock:unlock()
        if not ok then
            return nil,"failed to unlock: " .. err
        end
        return nil,"failed to update shm cache:  " .. err
    end
    -- 释放锁
    local ok, err = lock:unlock()
    if not ok then
        return nil,"failed to unlock: " .. err
    end
    return  val
end
local function mem_get(host)
    local res_host = cache:get(host)
    if res_host then
       return res_host
    else
    -- 当缓存中没有数据时，执行锁操作的查询函数
       local res_host = lock_db(host)
       return res_host
    end
end
function _M.fromcache(host)
```

```
        -- 在缓存中查找参数 host
        local res_host  =  mem_get(host)
        return res_host
    end

    return _M
```

上述代码的主要目的是从缓存中获取 Host 头信息，如果没有获取到 Host 头信息的数据，就去 MySQL 中读取，读取前会先给相同的 key 添加一个锁，这样可以确保对同一个 key 的操作在同一时间内只会执行一次，剩下的请求须等锁返回后再执行。

注意：本次代码使用 lua_shared_dict 的共享内存做示例，各位读者也可以看到 lua_shared_dict 在使用上和 lua-resty-lrucache 有细微区别。

配置 nginx.conf 文件，内容如下：

```
-- 创建锁操作的共享内存区域
lua_shared_dict db_locks 1m;
-- 创建缓存数据的共享内存区域
lua_shared_dict db_cache 5m;
server {
    listen 80;
    location / {
        access_by_lua_block {
            local host_deny = require "host_deny"
            local ngx = require "ngx"
            local host = ngx.var.host
            local white_host = host_deny.fromcache(host) or nil
            if not white_host then
                 ngx.exit(ngx.HTTP_FORBIDDEN)
            else
                 ngx.exit(ngx.OK)
            end

        }
        content_by_lua_block {
            ngx.say("hello world!!!")
        }
    }
}
```

执行压测，并发 5 个请求进行访问：

```
# webbench -c 5 -t 10  'http://www.zhe800.com/'
```

error.log 会输出如下的日志：

```
   2018/06/19 19:17:56 [error] 8318#8318: *18671259 [lua] host_deny.lua:38:
lock_db(): 0, client: 10.19.48.161, server: , request: "GET / HTTP/1.0",
host: "www.zhe800.com"
   2018/06/19 19:18:00 [error] 8318#8318: *18671262 [lua] host_deny.lua:38:
lock_db(): 3.511, client: 10.19.48.161, server: , request: "GET / HTTP/1.0",
host: "www.zhe800.com"
   2018/06/19 19:18:00 [error] 8318#8318: *18671261 [lua] host_deny.lua:38:
lock_db(): 3.511, client: 10.19.48.161, server: , request: "GET / HTTP/1.0",
host: "www.zhe800.com"
   2018/06/19 19:18:00 [error] 8318#8318: *18671263 [lua] host_deny.lua:38:
lock_db(): 3.511, client: 10.19.48.161, server: , request: "GET / HTTP/1.0",
host: "www.zhe800.com"
   2018/06/19 19:18:00 [error] 8318#8318: *18671264 [lua] host_deny.lua:38:
lock_db(): 3.511, client: 10.19.48.161, server: , request: "GET / HTTP/1.0",
host: "www.zhe800.com"
   2018/06/19 19:18:02 [error] 8318#8318: *18694720 [lua] host_deny.lua:38:
lock_db(): 0, client: 10.19.48.161, server: , request: "GET / HTTP/1.0",
host: "www.zhe800.com"
   2018/06/19 19:18:05 [error] 8318#8318: *18694721 [lua] host_deny.lua:38:
lock_db(): 3.011, client: 10.19.48.161, server: , request: "GET / HTTP/1.0",
host: "www.zhe800.com"
   2018/06/19 19:18:05 [error] 8318#8318: *18694722 [lua] host_deny.lua:38:
lock_db(): 3.011, client: 10.19.48.161, server: , request: "GET / HTTP/1.0",
host: "www.zhe800.com"
   2018/06/19 19:18:05 [error] 8318#8318: *18694723 [lua] host_deny.lua:38:
lock_db(): 3.011, client: 10.19.48.161, server: , request: "GET / HTTP/1.0",
host: "www.zhe800.com"
   2018/06/19 19:18:05 [error] 8318#8318: *18694724 [lua] host_deny.lua:38:
lock_db(): 3.011, client: 10.19.48.161, server: , request: "GET / HTTP/1.0",
host: "www.zhe800.com"
```

从日志中可以观察到如下情况。

- lock_db() 打印出的超过 3s 的请求占比很高，这是因为加了 sleep(3)。
- 最初缓存里没有数据，当第 1 条请求获取数据时加了锁。
- 如果在 3s 内多次请求相同的 key，会产生锁，ngx.log(ngx.ERR,elapsed)输出的值就是锁等待的时间。

注意：当查询的数据在 MySQL 中不存在时，会发现打印日志要快很多，这是因为当 MySQL 查询为空时，sleep 是不起作用的，但锁仍然在正常工作。

锁操作也可以做一些微调，避免出现因死锁或忘记释放锁而引发的性能问题，这些微调主要设置在 new 的指令中。

```
new
```

语法：obj, err = lock:new(dict_name, opts?)

含义：创建锁的新实例，dict_name 是在 Nginx 配置中声明的共享内存。

opts 是可选参数，它是 table 类型的，包含如下参数。

- exptime：持有锁的有效时间（单位为秒），默认是 30s，支持最小设置为 0.001s。它可以用来避免产生死锁。
- timeout：等待锁的最长时间，可以用来避免出现一直等待锁的情况。timeout 的值不能超过 exptime 的值，并且支持设置为 0 立即返回。
- step：等待锁的休眠时间（单位为秒），默认是 0.001s，如果发现已经有锁，在等待锁时会休眠 0.001s 后再去尝试获取锁，如果锁仍然很忙（如被其他请求占用），就继续等待，但每次等待的时间会受到 ratio 控制。
- ratio：控制等待锁的每次步长的比率，默认是 2，这意味着下一次等待的时间会翻倍，但总的等待时间不能超过 max_step 的值。
- max_step：设置最大的等待锁的睡眠时间（单位为秒），默认是 0.5s。

10.6 小结

本章讲解了 Ngx_Lua 中常见的缓存功能，它们各有利弊，在使用中通过合理的设计可以将其“利”发挥到最大，将其“弊”控制到最小。

第 11 章
动态管理 upstream

动态管理 upstream 是指 Nginx 在请求代理的过程中不用重载 Nginx 配置就可以完成对后端服务器的修改，这能够极大地提升 Nginx 的灵活度。这种方式在反向代理和网关系统中很受欢迎，它可以解决如下问题。

1．随着业务的高峰和低峰弹性伸缩资源配比，使 Nginx 可以动态分配权重和增减节点。

2．当后端服务节点出现异常时，可以快速停止分流并移除节点。

3．在后端服务进行更新的过程中，可以动态关闭分流，让新的请求不再分流到正在进行更新的服务器上，更新完成后再开启分流，这样可以实现平滑更新。

首先来讲述一下笔者在业务中维护 upstream 的历史概况。

1．当 Nginx 上代理的后端服务较少时，因为配置不多，所以选择直接修改 upstream，然后重载 Nginx 配置的方式。

2．随着服务类型越来越多，upstream 的配置变化较大，为了节省时间，这时选择使用自动化脚本来修改 upstream，然后自动重载 Nginx 配置。

3．当微服务流行起来后，upstream 的配置变得非常庞大，再加上云服务器的混用，导致对 upstream 的修改变得非常频繁，同时，Nginx 也会被多次重载配置，在高并发的 Nginx 下重载配置，需要重新建立连接，这对 Nginx 的吞吐是有很大影响的。因此不得不停止使用重载配置的方式，去寻找动态调整 upstream 的方法。

本章的核心内容就是动态管理 upstream 的实践方案。

11.1 实战需求分析

为了免去“造轮子”的痛苦，首先需要筛选出合适的工具来解决这个问题，筛选时需要考虑如下几个方面。

1．该工具提供的动态 upstream 是否会对 Nginx 的性能造成影响。

2．该工具的设计和架构是否清晰合理，开源项目维护的热度如何。

3．在使用该工具过程中，需要花费多少精力去维护。

4．在该工具使用过程中是否存在对 Nginx 指令的兼容问题，如是否支持长连接、权重配置、健康检测等。

经过筛选，可以找到多个工具来实现动态管理 upstream 的方案，下面将会对这些工具进行一一介绍。

11.2 ngx_http_dyups_module

ngx_http_dyups_module 是一个第三方开源软件，它提供 API 动态修改 upstream 的配置，并且支持 Nginx 的 ip_hash、keepalive 等与 upstream 有关的配置。

11.2.1 安装 ngx_http_dyups_module

ngx_http_dyups_module 是用 C 语言开发的，需要在 Nginx 中编译安装，安装方式如下：

```
# git clone git://github.com/yzprofile/ngx_http_dyups_module.git
# ./configure --add-module=/path/ngx_http_dyups_module
```

11.2.2 动态管理 upstream

下面是使用 ngx_http_dyups_module 来动态管理 upstream 的示例。首先，将 nginx.conf 文件的内容改成如下代码，并重载 Nginx 配置：

```
upstream test_12 {
    server 127.0.0.1:8001;
}
upstream test_34 {
    server 127.0.0.1:8001;
}

server {
```

```
        listen 8000;
    location / {
            # 进行 IP 地址限制或权限管理，避免数据接口被恶意修改
            allow 127.0.0.1;
            deny  all;

            # 启动管理 upstream 配置的接口
            dyups_interface;
        }
    }
    server {
        listen 80;
        location / {
            proxy_set_header     Host $host;
            proxy_set_header     X-Real-IP $remote_addr;
            proxy_set_header     X-Forwarded-For $proxy_add_x_forwarded_for;

            # 将 test_ser 赋值给$ups，test_ser 是 upstream 的 name
            # 所以使用$ups 时 proxy_pass 后面的 upstream 必须是已赋值的变量
            set $ups test_ser;
            proxy_pass http://$ups;
        }
    }
```

确认目前的 upstream 清单中是否有 test_ser：

```
    # curl 127.0.0.1:8000/list
    test_12
    test_34
```

上面的输出显示没有 test_ser，则添加 upstream，并命名为 test_ser（它是以覆盖当前 upstream 的内容完成配置的）：

```
    # curl -d "server 127.0.0.1:81 weight=10 max_fails=5 fail_timeout=10;
server 127.0.0.1:83 weight=20 max_fails=7 fail_timeout=10;" 127.0.0.1:8000/
upstream/test_ser

    success
```

获取 upstream 清单的详情：

```
    # curl 127.0.0.1:8000/detail
    test_12
    server  127.0.0.1:81  weight=20  max_fails=10  fail_timeout=5  backup=0
down=0

    test_34
```

```
    server  127.0.0.1:111  weight=20  max_fails=10  fail_timeout=5  backup=0
down=0
    server  127.0.0.1:821  weight=20  max_fails=10  fail_timeout=5  backup=0
down=0

    test_servers
    server  10.19.48.162:81  weight=10  max_fails=5  fail_timeout=10  backup=0
down=0

    test_ser
    server  127.0.0.1:81  weight=10  max_fails=5  fail_timeout=10  backup=0
down=0
    server  127.0.0.1:83  weight=20  max_fails=7  fail_timeout=10  backup=0
down=0
```

访问此 upstream 下的服务，发送测试请求确认是否正常（结果显示请求正常）。然后，验证节点删除功能：

```
    # curl -i -X DELETE 127.0.0.1:8081/upstream/test_ser
```

进行压力测试，与原生的 Nginx upstream 配置相比，压测结果显示正常，没有明显的性能波动。之后，持续压测一周观察服务的稳定性，压测过程中，upstream 的动态添加和删除操作会随机执行。（数据显示：稳定性正常。）

dyups 支持的接口说明见表 11-1。

表 11-1　dyups 支持的接口说明

请求方法	HTTP 接口	用　　途
GET	/detail	获取所有 upstream 的清单明细
GET	/list	获取所有 upstream 的 name
GET	/upstream/name	获取指定的 upstream 内的后端服务器的 IP 地址和端口号
POST	/upstream/name	覆盖指定的 upstream 的内容，包括 IP 地址、端口号，权重等有关信息
DELETE	/upstream/name	删除指定的 upstream

功能验证完后，准备灰度环境。

此模块还提供了 Ngx_Lua 接口，方便 Ngx_Lua 处理动态的 upstream，有兴趣的读者可参考官方 Wiki。

11.2.3　确保 upstream 数据的完整性

使用 ngx_http_dyups_module 时有一个隐患，即动态修改的配置是存放在 Nginx 内存中的，如果重载 Nginx 配置，这些数据就会丢失。

对此，可行的解决方案是开发一个脚本，用来调用 ngx_http_dyups_module 的接口/detail，获取 upstream 的全部信息，并存储为 Nginx 可以读取的 upstream 配置文件；在 upstream 发生变更时就会触发该脚本生成一个 upstream 文件，将这个文件 include 到 Nginx 配置中；当 Nginx 重载配置时，它会先从这个 upstream 文件加载最新的数据。生成一个 Nginx 的 upstream 脚本很简单，这里就不再讲解了。

ngx_http_dyups_module 提供了 Ngx_Lua 接口，所以数据的加载可以通过 Ngx_Lua 来完成，如果想用 Ngx_Lua 来控制数据的更新和存储，可参考官网说明，网址为 https://github.com/yzprofile/ngx_http_dyups_module/blob/master/README.md。

如果是需要进行大规模部署的 Nginx 集群，就要做一些合理的规划，这里给出几个很实用的建议。

- 监控每台服务器的 upstream 的详细信息，确保每次执行的数据都是一致的。
- 监控每台服务器的脚本生成的 upstream 文件，确保文件内容一致。
- 确保动态修改 upstream 的功能是正常的，设置一个测试节点，定时进行添加和删除操作，确保这个测试节点的变化符合要求，并请求该测试节点，验证访问是否正常。

11.3　nginx-upsync-module

nginx-upsync-module 是微博的技术团队开源的第三方模块，它和 ngx_http_dyups_module 一样在修改 upstream 时不需要重载 Nginx 配置，但 nginx-upsync-module 的 upstream 数据存放在 Consul 中，需要从 Consul 中进行同步。

11.3.1　安装 nginx-upsync-module 和 Consul

此模块是用 C 语言开发的，在 Nginx 中编译即可：

```
# git clone https://github.com/weibocom/nginx-upsync-module.git
# ./configure --add-module=/path/nginx-upsync-module
```

因为原始数据存放在 Consul 中，所以重点说一下 Consul：它是一个基于 Go 语言开发的高可用、分布式系统。它主要用来发现和配置服务，类似于 Zookeeper；也可以用来存储 key/value 数据，并提供 API 去操作 key/value 数据。 Consul 用途非常广泛，但在本书中只需学会用它来存储 upstream 数据就可以了。

安装 Consul 时，首先请安装 1.9 以上版本的 Go 语言，一般情况下 yum 的就是这个版本，如下所示：

```
# yum install golang -y
```

下载 Consul 安装包，将文件复制到 Linux 系统的/bin 目录下，如下所示：

```
# wget -S https://releases.hashicorp.com/consul/1.1.0/consul_1.1.0_linux_amd64.zip
# unzip consul_1.1.0_linux_amd64.zip
# cp consul /usr/local/bin/
```

安装完成后，启动 Consul（建议先使用开发模式，以便更好地观察情况）：

```
# consul agent -dev -client 10.19.48.161
```

启动后可以访问 Consul 的后台界面，地址为 http:// 服务器 IP:8500/ui/，本步骤是可选操作。

11.3.2 Consul 的键值操作

Consul 提供了存储键值数据的 API，其对键值数据进行增、删、改的操作方式如下。

1．添加操作，语法格式如下：

```
curl -X PUT -d '{"weight":1, "max_fails":2, "fail_timeout":10}' http://$consul_ip:$port/v1/kv/$dir1/$upstream_name/$backend_ip:$backend_port
```

示例如下：

```
#curl -X PUT http://127.0.0.1:8500/v1/kv/upstreams/test_ser/127.0.0.1:82 -d '{"weight":1, "max_fails":20, "fail_timeout":20}'
```

2．删除操作，语法格式如下：

```
curl -X PUT -d '{"weight":1, "max_fails":2, "fail_timeout":10}' http://$consul_ip:$port/v1/kv/$dir1/$upstream_name/$backend_ip:$backend_port
```

示例如下：

```
# curl -X DELETE http://127.0.0.1:8500/v1/kv/upstreams/test_ser/127.0.0.1:82
```

3．只修改权重等参数，语法格式如下：

```
curl -X PUT -d '{"weight":2, "max_fails":2, "fail_timeout":10}' http://$consul_ip:$port/v1/kv/$dir1/$upstream_name/$backend_ip:$backend_port
```

示例如下：

```
#curl -X PUT http://127.0.0.1:8500/v1/kv/upstreams/test_ser/127.0.0.1:82 -d '{"weight":3 "max_fails":10, "fail_timeout":11}'
```

11.3.3　动态管理 upstream

之前将 upstream 的数据存放在了 Consul 中，现在要将它读取到 Nginx 中，完整的示例代码如下：

```
    http {
    upstream test_ser {
            # 从 Consul 中读取 test_ser 的信息，存放到 Nginx 的内存中
            upsync  127.0.0.1:8500/v1/kv/upstreams/test_ser/  upsync_timeout=
3m upsync_interval=5s upsync_type=consul strong_dependency=off;

            # 内存中的 upstream 的每次变更都会同步到这个配置文件中
            upsync_dump_path/usr/local/nginx/conf/servers/servers_test.conf;

            # 存放初始化的 upstream 数据文件，和 upsync_dump_path 保持一致
            include /usr/local/nginx/conf/servers/servers_test.conf;

            # 当启动指令 least_conn 或 hash 时，必须声明 upsync_lb 以使负载均衡生效
            least_conn;
            upsync_lb least_conn;
        }
        server {
            listen 80;
            location / {
                proxy_pass http://test_ser;
            }

            location = /upstream_show {
               # 需要使用 allow、deny 做访问限制
               # 展示所有后端服务器的配置信息
                upstream_show;
            }
        }
    }
```

下面是对上述配置中的一些指令的解读：

指令：upsync

环境：upstream

含义：配置 Consul 的服务地址和端口号，并从 Consul 中获取指定 key 的值，即 upstream 在对应的后端服务器上的路径。

由此可知，上述配置中 127.0.0.1:8500/v1/kv/upstreams/test_ser/的意思是 Consul 的 IP 地址+

端口号为 127.0.0.1:8500，Consul 中 key 的值为/v1/kv/nginx_upstream/test_ser/。

关于 upsync 其他参数的说明见表 11-2。

表 11-2　关于 upsync 其他参数的说明

参数名	作　　用
upsync_interval	设置查询 Consul 的间隔时间，默认为 5s
upsync_timeout	设置查询 Consul 的超时时间，默认为 6ms
upsync_type	服务器的类型，支持 Consul 和 Etcd
strong_dependency	若设置为 on，则当 Nginx 重载配置或重启时，Nginx 会主动从 Consul 或 Etcd 中读取数据；若设置为 off，则 Nginx 重载配置或重启时会从 upsync_dump_path 对应的文件中读取数据

注意：建议把 strong_dependency 设置为 off，这样重载配置或重启 Nginx 时，不会在第一时间去使用 Consul 中的数据，可以避免在第 1 次读取时出现超时或延迟的情况，当设置为 on 时，会使用 upsync_dump_path 对应的文件中的数据。

指令：upsync_dump_path

语法：upsync_dump_path $path

默认：/tmp/servers_$host.conf

环境：upstream

含义：将后端服务器的列表信息存放到指定的文件中，建议和初始化时读取的文件路径一致（如上述配置中的 include /usr/local/nginx/conf/servers/servers_test.conf;）。这样，当 strong_dependency 为 off 时，每次重启 Nginx 都可以获取最近的版本。

指令：upsync_lb

语法：upsync_lb $load_balance

默认值：round_robin/ip_hash/hash modula

环境：upstream

含义：如果在 upstream 中需要使用 least_conn 和 hash，必须配合 upsync_lb 来声明，可以参考上述配置中的内容。

指令：upstream_show

语法：upstream_show

默认：无

环境：location

含义：输出后端服务器的所有配置信息

11.3.4　验证动态配置功能

原始数据被存放在 Consul 中，所以只要操作 Consul，upstream 的数据就会有变动。仍然使用上述配置中的格式，再给 test_ser 配置一个初始化地址（如果是线上服务，请先配置正在使用的真实的服务器），示例如下：

```
# vim  /usr/local/nginx/conf/servers/servers_test.conf

server 1.0.0.1:1222 weight=10 max_fails=10 fail_timeout=10s;
```

查看当前 Nginx 内存中 test_ser 的 upstream 配置：

```
# curl http://127.0.0.1/upstream_show

Upstream name: test_ser; Backend server count: 1
        server 127.0.0.1:81 weight=10 max_fails=10 fail_timeout=10s;
```

修改权重等参数：

```
# curl -X PUT http://127.0.0.1:8500/v1/kv/upstreams/test_ser/127.0.0.1:
82 -d '{"weight":1, "max_fails":20, "fail_timeout":20}'
```

再次查看 upstream 配置会发现 test_ser 已经被修改，请注意，upstream 的生效不一定是实时的，因为配置生效的频率是由 upsync_interval 设置的时间来控制的。

11.3.5　高可用、高并发设计

Consul 中的数据是由 Nginx 主动同步的，每个 upstream 配置都独立同步，并且每个 worker 进程都需要执行同步，因此，在大规模部署的 Nginx 集群下，会导致 Consul 的请求非常多，为了确保 Consul 的性能，建议采取 Consul 集群来做负载均衡，配置方案如下。

- 给一个 server 配置多个 client，建议把 client 配置在每台 Nginx 服务器上。
- 全部使用 127.0.0.1:8500 来访问本机的 Consul。
- 监控 Consul 集群状态，确保数据的稳定性（Consul 已经提供了相关的功能）。
- 当 Consul“挂掉”后，服务仍然会提供访问，只是无法再对 upstream 的配置进行操作，此时，可以离线修复 Consul，这样对服务的影响较小。

11.4 基于 balancer_by_lua_block 的灵活控制

前面讲到的 ngx_http_dyups_module 和 nginx-upsync-module 两个模块都是用 C 语言开发的，现在来介绍一个使用 Ngx_Lua 来完成动态管理 upstream 的方案。

balancer_by_lua* 让 Ngx_Lua 可以控制 upstream 阶段，并且它也继承了 Nginx 长连接的特性。如果读者打算使用 Ngx_Lua 来控制 upstream，建议使用 OpenResty，因为它支持使用很多指令，并且维护方便。

官方文档（https://github.com/openresty/lua-resty-core/blob/master/lib/ ngx/balancer.md）给出了完整的配置示例，如下所示：

```
http {
    upstream backend {
        server 0.0.0.1;   # just an invalid address as a place holder
        balancer_by_lua_block {
            local balancer = require "ngx.balancer"

            -- well, usually we calculate the peer's host and port
            -- according to some balancing policies instead of using
            -- hard-coded values like below
            local host = "127.0.0.2"
            local port = 8080

            local ok, err = balancer.set_current_peer(host, port)
            if not ok then
                ngx.log(ngx.ERR,"failed to set the current peer: ", err)
                return ngx.exit(500)
            end
        }

        keepalive 10;  # connection pool
    }

    server {
        # this is the real entry point
        listen 80;

        location / {
            # make use of the upstream named "backend" defined above:
            proxy_pass http://backend/fake;
        }
    }
```

```
    server {
        # this server is just for mocking up a backend peer here···
        listen 127.0.0.2:8080;

        location = /fake {
            echo "this is the fake backend peer···";
        }
    }
}
```

把上述配置中的最后两行换成 Ngx_Lua 共享内存中的数据，然后再提供一个修改共享内存数据的接口，就可以完成动态 upstream 管理了：

```
local host = "127.0.0.2"
local port = 8080
```

另外 OpenResty 还支持设置权重等更多功能，有兴趣的读者可以去了解一下，地址为 https://github.com/openresty/lua-resty-balancer。

使用 Ngx_Lua 实现动态 upstream 管理能够提升灵活度，但如果对 upstream 的需求并不复杂，建议使用 nginx-upsync-module 或 ngx_http_dyups_module 即可。

11.5　小结

本章讲解了 3 种常见的动态管理 upstream 的解决方案。一个完善的服务监控系统，可以更精准地控制后端服务器的权重、分流等配置，这给有自动伸缩资源配比需求的系统提供了强有力的支撑。

第 12 章
Nginx 日志分析系统

Nginx 的访问日志记录了每条请求的来龙去脉，通过日志可以分析出很多有用的监控信息，例如下面的这些信息。

1．请求的响应时间。

2．请求到达的后端服务器的地址和端口。

3．请求是否存在缓存配置。

4．请求体、请求头、响应体和响应头的大小等。

5．客户端的 IP 地址、User_Agent 等信息。

6．自定义变量的内容。

通过这些信息，可以对请求的各项指标进行监控，这对应用级别的服务来说是非常重要的。本章将会对 Nginx 日志分析中常见的需求进行说明，并提供一个有效的日志分析方案。

12.1 实战需求分析

首先需要确认 Nginx 日志分析应该使用什么类型的工具。和筛选动态 upstream 管理工具的方式一样，它应该至少满足如下条件。

1．可以计算出 URI 响应的平均值，以及 p90、p99 等任意比例的值。

URI 在指定的某段时间内按照请求的响应时间进行升序排序，p99 的意思是指在 99%这个位置的响应时间，即确认出 99%的请求所花费的时间，用于体现服务的响应能力。

2．可以对 URI 进行分组和汇总统计，特别是当 URI 为正则表达式类型时。

在很多业务中，URI 会使用正则表达式类型的路由，如折 800 商城的详情页 https://shop.zhe800.com/products/ze171126205509136896 和 https://shop.zhe800.com/products/ze170814104348738286，这两个网址属于同一个 URI 服务类型，只是后面的数字不一样，它们在服务中都可以表示为 https://shop.zhe800.com/products/[a-z0-9]+的正则表达式。如果不将这些带有正则表达式的 URI 进行归类，在汇总时 URI 分散性会非常高，从而无法做到准确的定位和报警。因此对 URI 进行归类分组是非常重要的。

3．可以监控 URI 的走向，确认是否存在缓存，确认后端服务器属于哪个 Web 应用。

确认是否存在缓存的主要目的是为了避免出现上线时需要加缓存的服务没加的情况。可以通过日志分析定期梳理这些服务，找到没有添加缓存的服务，然后和业务部门确认是否需要配置缓存。（URI 是否有缓存可以通过在响应头中加入特定的头信息进行标识，例如 CDN 缓存一般有 Cache-Control 头。）

4．可以支持集群模式，大多数互联网公司会使用多台 Nginx 服务器，数据的分析需要在日志集中搜集后再进行。

5．可以提供实时监控，及时进行分析并反馈异常信息。

6．可以提供定制化服务，满足不同业务的需求。

上述功能的实现不必一蹴而就，可以在使用中通过不断迭代完成，在高并发、多业务模式下的服务均可参考这些需求。但如果 Nginx 的服务单一且并发程度较低，只需用一些简单的分析工具甚至脚本即可。

下面将会介绍一些工具，利用它们来完成上述各种需求。

12.2　ngxtop 实时分析

ngxtop 是用 Python 语言开发的在线分析的工具，它可以对 Nginx 请求进行实时分析，使用方法也非常简单。

首先，安装 ngxtop。ngxtop 是 Python 的包，所以使用 Python 的 pip 命令安装即可（ngxtop 支持 python2 和 python3）：

```
# yum install python-pip
# pip install ngxtop
```

在使用 ngxtop 时，需要确保 Nginx 日志格式是默认格式，因为 ngxtop 是通过对日志格式

的匹配得到的数据，如果格式改变将会导致数据分析异常。如果有新的变量要加入日志格式中，请将其添加到默认配置的后面（新的变量在 ngxtop 中无法被分析，只能记录到硬盘上，以方便业务的其他需求），代码如下：

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

执行分析命令如下：

```
#  ngxtop --config  /usr/local/nginx/conf/nginx.conf -n  10
```

ngxtop 会找到配置文件中 access_log 的位置，-n 的作用是显示所输出 URI 的行数，默认是 10 行。ngxtop 日志分析结果如图 12-1 所示。

```
running for 20 seconds, 1230 records processed: 61.19 req/sec

Summary:
|   count |   avg_bytes_sent |   2xx |   3xx |   4xx |   5xx |
|---------+------------------+-------+-------+-------+-------|
|    1230 |         2945.304 |  1175 |    46 |     9 |     0 |

Detailed:
| request_path                    |   count |   avg_bytes_sent |   2xx |   3xx |   4xx |   5xx |
|---------------------------------+---------+------------------+-------+-------+-------+-------|
| /app/cart/item/count            |      90 |           44.389 |    90 |     0 |     0 |     0 |
| /cn/f/status                |      70 |          558.414 |    70 |     0 |     0 |     0 |
| /check.json                     |      69 |           13.000 |    69 |     0 |     0 |     0 |
| /native/jump                    |      48 |           38.000 |    48 |     0 |     0 |     0 |
| /dx/pricebanner |      45 |          115.156 |    43 |     2 |     0 |     0 |
| /dx/promotion   |      43 |          176.791 |    40 |     3 |     0 |     0 |
| /dx/status      |  38 |          351.684 |    38 |     0 |     0 |     0 |
| /dx/comment     |      35 |          723.943 |    34 |     1 |     0 |     0 |
| /dx/shop        |      33 |         1044.606 |    33 |     0 |     0 |     0 |
| /dx/graph       |      32 |          903.500 |    31 |     1 |     0 |     0 |
```

图 12-1　ngxtop 日志分析结果

从图 12-1 中可以看到请求的总量、URI 的访问次数、平均发送字节数及 HTTP 状态码。

ngxtop 还支持如下功能。

- 支持直接分析指定的日志文件，命令为“ngxtop -l“Nginx 日志文件路径”;”。
- 支持对 IP 地址进行不同条件的排名，命令为“ngxtop --config/usr/local/nginx/ conf/nginx.conf top remote_addr”。
- 支持日志过滤，如只要 HTTP 状态码为 502 的数据，则命令为“ngxtop -l /data1/access.log --filter 'status == 502'”。

在网站搭建的初期，使用 ngxtop 对实时分析有很大的帮助，但随着网站规模的扩大，业务的增多，继续使用此工具会遇到瓶颈。总结 ngxtop 的优缺点如下。

优点：

- 安装和使用非常简单。
- 有多种实时工具和数据分析模型。
- 对 Nginx 版本没有依赖性，不必担心 Nginx 版本升级造成的兼容问题，只要日志格式符合要求即可。

缺点：

- 无法自定义监控数据。
- 数据汇总能力欠佳，无法对有正则表达式的服务进行监控。
- 不支持集群化，适合对单点的 Nginx 服务器进行分析。
- 日志格式不灵活，只能在后面追加格式。

总而言之，它比较适合用于小型单机网站或临时问题的分析。还有一个工具 GoAccess 与 ngxtop 比较类似。

12.3　Flume 方案的日志分析

由于 ngxtop 工具的局限性，特别是它在集群化方面的缺陷，我们需要重新寻找工具，经过筛选决定采用 Flume 来收集数据完成日志汇总，并利用比较流行的 Elasticsearch 来进行数据存储。Flume 方案的日志分析流程图如图 12-2 所示。

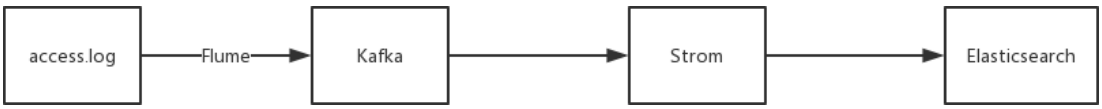


图 12-2　Flume 方案的日志分析流程图

- Flume 负责收集日志，并将数据格式规范化后写入 Kafka。
- Kafka 中的数据被 Storm 实时分析，并在计算后写入 Elasticsearch。
- Elasticsearch 中的数据被用于实时和离线的各类数据统计。

先说说这个方案的优缺点。

优点：

- 使用 Flume 来收集日志，性能有保证，并且可以自定义日志格式。
- Kafka 可以横向扩展，性能好且数据不易丢失，适用于资源消耗压力较大的情况。
- Storm 是非常有名的实时分析工具，可以很方便地实现自定义的需求。

- Elasticsearch 也支持扩展，并支持多种 SQL 查询，使数据的汇总分析变得更加简单。

缺点：

- 使用了过多的组件，如果只是用来监控服务，有点小题大做。
- 当自定义需求时，如把正则表达式类型的 URI 服务归类，此时，在 Storm 中计算需要和开发语言（如 Java）进行互动，在高并发状态下，资源消耗会过多。
- Elasticsearch 的 SQL 功能虽多，但仍然无法满足各种数据分析的需要，会导致很多计算仍需依靠代码分析或混用多条 SQL 语句来完成。
- 在高并发情况下 Flume 的收集和格式化操作容易对 Nginx 服务器的资源产生过多消耗。

本节相关组件的安装和使用方法在网上有很多相关资料，这里不再赘述。

Flume 方案对资源消耗过多，维护成本也高，随着业务的发展，最终会被放弃，取而代之的是一个全新的数据收集和分析工具，这也是本章要讲的重点内容——智能化 nginx_log_analysis。

12.4 智能化 nginx_log_analysis

12.4.1 架构重构

现在的需求是除进行日志分析外，日志分析工具还需同时满足高性能、低消耗、迭代方便等需求。看上去确实有点令人头疼，毕竟在开源社区中，Nginx 的日志分析工具能产品化的非常少，那么，可以自己开发一套工具来实现这些功能。

如果可以在 Nginx 上将数据格式化后直接通过网络发送给数据库，再由数据库完成分析，这样中间的其他环节（如读取硬盘、格式化日志等）都可以省略掉了，那么，将会极大地降低维护成本。但如何才能让这种设想得以实现呢？这就要用到前面学习过的 Ngx_Lua 了。具体流程分析如下。

- Nginx 的日志内容是由 Nginx 的变量组成的，Ngx_Lua 可以直接获取这些变量。
- Ngx_Lua 可以对 Nginx 变量进行数据处理，如格式化、对 URI 进行分类等。
- Ngx_Lua 可以利用 log_by_lua*在 log 执行阶段将整合好的数据传到远程服务器上。
- 远程服务器是一个时序数据库，它可以执行多种函数，例如 p90 计算、平均数计算、热点数据计算、分组、正则匹配，甚至设置定时任务等。
- 数据库要求是高性能的，能处理实时的数据分析。

最后将计算后的数据和监控系统打通，提供报表和报警功能。因为代码过多，笔者将这个

系统的代码在 GitHub 上进行了开源，地址是 https://github.com/leehomewl/nginx_ log_analysis。

注意：本章不会对所有代码都进行讲解，但会选取一部分代码进行说明，主要目的是让读者了解如何在开发中使用 Ngx_Lua 来完成架构设计和流程规划，从而提升开发水平。

12.4.2　日志远程传输

首先，需要解决日志远程传输的问题，而且要支持集群化，即数据的传输要统一存储、统一计算。

这里需要用到模块 lua-resty-logger-socket，它的主要功能是以非阻塞 I/O 的方式推送数据到远程服务器上。它属于 Nginx 的 log 执行阶段，是在请求反馈给客户端后执行的操作，所以在此处传输日志，即使数据推送失败，也不会影响客户端的响应。关于 lua-resty-logger-socket 的用法请参考 12.5 节。

12.4.3　时序数据库

Nginx 对日志的分析基于时间的维度，如波动的报表、请求 PV（Page View，即页面浏览量）的涨幅、平均响应时间的对比等都是在时间的基础上进行的。再如常用的 Nagios 和 Zabbix，在报警和监控中也是以时间为基数的，所以需要找一个时序数据库来支撑数据分析。

在时序数据库的选择上，可以使用 InfluxDB，它是一款基于 Go 语言开发的开源分布式时序、事件和指标数据库，非常适合用来处理监控数据，它提供了很多函数，这些函数包含了绝大部分的数据计算方式，可以简化数据分析的代码。关于 InfluxDB 的用法请参考 12.6 节。

12.4.4　日志规则设计

Nginx 记录后端响应时间的变量是$upstream_response_time，但有时它会有多个值，各值之间以逗号分隔，这是因为受 proxy_next_upstream 的控制，当后端服务响应异常时会将请求代理到另外一台后端服务器上进行响应，所以就出现了多个值。这样会导致存放在数据库中的响应时间字段的部分数据不是数字， InfluxDB 函数无法执行计算。为了避免出现这种情况，在 Nginx 写入时就要将以逗号分隔的数字累加后再插入数据库中。

在 Nginx 中 URI 的变量是$uri，它不包含 URL 中的参数，$uri 即服务。之前提到过，如果 URI 无法区分正则表达式就会导致计算分散化，从而失去分析和监控的意义。因此需要在 Nginx 中将这些正则表达式的数据区分出来。

那么 URI 是否是正则表达式是如何让 Ngx_Lua 知道的呢？这就涉及 URI 认领的问题了，大致方式如下：

- 清理公司业务线上使用的 URI。
- 对 URI 进行筛选，确认哪些是精确 URI、哪些是可以合并到正则 URI 上的。
- 将筛选后的 URI 数据存放到 MySQL 中。
- Ngx_Lua 从 MySQL 读取 URI 数据并将其存放到内存中。
- 当客户端请求发送到 Nginx 时，Ngx_Lua 在 log 阶段判断当前请求属于哪个 URI 服务（正则 URI 或精确 URI）。

URI 认领并非只是为了方便数据汇总，它还可以实现更多功能，例如，想要判断某个项目是否使用了新的 URI，可以用 Ngx_Lua 在测试环境下做一层验证，判断请求的 URI 是否为认领中的服务，如果不是，则为新 URI。这样每次上线前都可以提前配置和监控 URI。在补充监控时也可以对新增的 URI 添加其他属性，如配置缓存、监控 POST 的数据长度范围、进行降级容灾处理（详见第 13 章）等。总之，通过这种方式，可以对 URI 做非常细致的监控。

关于正则表达式 URI 的存放和匹配格式，下面举例说明。例如，MySQL 中存放了一个路由是/a/b/[0-9]+的 URI 服务，并由 Nginx 将该数据读取到内存中进行缓存，当客户端请求 Nginx 且 URI 是/a/b/123 或/a/b/345 时，就会被 Ngx_Lua 匹配成/a/b/[0-9]+，最后将/a/b/[0-9]+写入 InfluxDB，并会用它来完成数据分析。

通过这套流程，就可以得到想要的分析报告了。

注意：以上对 MySQL 的应用，只是为了区分 URI 是正则表达式还是精确类型，有些读者的服务可能只有精确的 URI，在这种情况下，MySQL 是可以去掉的，但如果读者希望使用更多的监控功能，MySQL 还是非常重要的，这在第 13 章中会有详细的讲解。

12.5 lua-resty-logger-socket 传输方案

可以使用 lua-resty-logger-socket 进行日志传输，它在 log_by_lua*阶段执行，此阶段是在请求响应完成后执行的，所以对服务的影响很小，但如果 Ngx_Lua 的代码质量很差也会增加对服务器 CPU 和网络 I/O 资源的消耗，从而影响 Nginx 的整体性能。关于代码的性能问题请参考第 16 章。

12.5.1 安装 lua-resty-logger-socket

lua-resty-logger-socket 通常会和 syslog-ng 打交道，但很显然 syslog-ng 不适用于本方案。在安装 lua-resty-logger-socket 之前，请先安装 lua-nginx-module，然后，下载 lua-resty-logger-socket 安装包，复制 lua_package_path 所设置的值，如下所示：

```
# git clone https://github.com/cloudflare/lua-resty-logger-socket.git
# cp -pR lua-resty-logger-socket/lib/resty/* \
/usr/local/nginx_1.12.2/conf/lua_modules/resty/
```

12.5.2　远程传输配置

下面是远程传输日志的一个简单示例：

```
lua_package_path "/usr/local/nginx_1.12.2/conf/lua_modules/?.lua;;";

server {
    location / {
        log_by_lua_block {
            local ngx = require "ngx"
            local uri = ngx.var.uri
            local host = ngx.var.host
            -- 声明一个 UDP 的远程服务器，格式是 table 类型
            local db = {
                host = '127.0.0.1',
                port = 8911,
                sock_type = udp,
                flush_limit = 8096,
                drop_limit = 2097152
            }
            local logger = require "resty.logger.socket"
            if not logger.initted() then
                -- 初始化日志收集的服务
                local ok, err = logger.init(db)
                if not ok then
                    ngx.log(ngx.ERR,"failed to initialize the logger: ",
                            err)
                    return
                end
            end
            local msg = host .. uri
            -- 将 host 和 uri 的日志传输到远程服务器上
            local bytes, err = logger.log(msg)
            if err then
                ngx.log(ngx.ERR, "failed to log message: ", err)
                return
            end
        }
```

```
            proxy_pass http://servers;
        }
    }
```

上述配置中将日志传输到远程的服务器使用的是 UDP（User Datagram Protocol）。

12.5.3 参数解读

lua-resty-logger-socket 提供了多个指令和参数来调整服务。

1．初始化配置

语法：ok, err = logger.init(user_config)

说明：logger.init 是用来初始化远程服务器的，这是日志传输前的必备操作，user_config 是 table 类型的，它可以支持表 12-1 所示的初始化参数。

表 12-1 user_config 支持的初始化参数

参　数	作　用
flush_limit	数据不是一条一条传输的，而是当内存缓冲区大小 + 当前请求日志的大小 ≥ flush_limit 设置值时才会进行传输。默认值是 4096 字节，即 4KB。如果不希望请求频繁发送，可以调整此参数
drop_limit	在数据传输过程中，当内存缓冲区大小 + 当前请求日志的大小 ≥ drop_limit 设置值时将会丢弃缓存区的数据，所以 drop_limit 的值需要根据业务来设置，默认是 1048576 字节，即 1MB
timeout	传输中的超时设置，主要是为了避免请求无法返回响应导致服务一直处于等待状态的情况发生
host	日志服务器的 IP 地址
port	日志服务器的端口
sock_type	传输类型，支持 TCP、UDP，如 TCP 类型，即 sock_type = ‘tcp’。默认是 TCP 类型
path	支持 UNIX 套接字的连接，path 是套接字的路径
max_retry_times	日志传输失败后，允许重试的最大次数
retry_interval	重试传输的间隔时间，默认情况下 retry_interval=100，即 0.1s
pool_size	传输连接池的大小，默认情况下 pool_size =10
max_buffer_reuse	内部日志缓冲区在创建新缓存之前的最大重用次数，主要是为了防止内存泄漏
periodic_flush	定期刷新日志到日志服务器上，若设置为 nil 即关闭此功能，单位是秒
ssl	布尔值，开启或关闭 SSL（Secure Sockets Layer ，安全套接层）连接，默认情况下 ssl=false
ssl_verify	布尔值，启用或关闭验证主机和证书是否匹配，默认为 true
sni_host	设置主机名在 SNI（Server Name Indication）中发送，并在验证证书匹配时使用

2．检查初始化配置

语法：initted = logger.initted()

说明：返回一个布尔值，用来确认服务是否已经初始化。如果返回 false，代表没有初始化，此时应该先进行初始化。

3．日志传输

语法：bytes, err = logger.log(msg)

说明：当此模块的缓冲区中存放的数据达到 flush_limit 限制或达到 periodic_flush 设置的刷新时间后，就会将变量 msg 的值传输到远程日志服务器上。返回值中的 bytes 是缓冲区的字节数，在刷新缓存区时，如果出现异常，错误信息会存放到 err 中。

4．立即传输日志

语法：bytes, err = logger.flush()

说明：表示立即刷新缓存的数据到日志服务器上，但一般不需要执行此操作，因为当数据超过 flush_limit 设置的值时，会自动刷新缓冲区数据到日志服务器上。

12.6　时序数据库 InfluxDB

日志完成远程传输后，需要使用 InfluxDB 来存放日志，本书涉及 InfluxDB 的内容只和日志分析有关，所以不对 InfluxDB 进行详细讲解。对 InfluxDB 感兴趣的读者可以访问如下地址：https://docs.influxdata.com/influxdb/v1.6/。

12.6.1　安装 InfluxDB

下面以在 CentOS 环境下进行安装为例， 对 InfluxDB 的安装进行示范：

```
    # wget https://dl.influxdata.com/influxdb/releases/influxdb-1.5.4.x86_64.
rpm
    # sudo yum localinstall influxdb-1.5.4.x86_64.rpm
```

安装成功后，启动 InfluxDB：

```
    # /etc/init.d/influxdb start
```

更多操作系统上的安装方法请参考 https://portal. influxdata.com/downloads。

12.6.2　基本概念和操作

在 InfluxDB 中，关键词的名字和传统数据库是有一些区别的，关于 InfluxDB 关键字的说明见表 12-2。

表 12-2　关于 InfluxDB 关键字的说明

关键字	说　明	示　例
database	数据库	创建数据库： create database nginx;
measurement	数据库中的表，相当于传统数据库中的 table	展示数据库中的表： show measurements
point	表中的 1 行数据	插入 1 行数据： insert nginx_log,host=testnginx.com uri= "/abc"

InfluxDB 中的表不需要显式创建，执行 insert 操作时会自动创建，表中的 1 行数据（point）是由 time、tags、fields 组成的。

time：每条数据插入的时间，是自动生成的索引，单位是纳秒（nanosecond）。

tags：可选的索引，用来记录各种值，利用该索引可以加快查询速度，还可以在索引上执行 group by、order by 等操作，但不支持使用函数。

fields：记录各种值，是普通的表字段，但支持使用函数查询。

注意：time 和 tags 都是索引，它们会一起组成一个唯一值，这就意味着在 time 和 tags 的值完全相同的情况下只会存储一条数据。

12.6.3　数据分析之查询函数

InfluxDB 提供了大量的函数来完成查询任务，这极大地简化了数据分析的工作，表 12-3 是监控分析中常见的查询函数及其作用。

表 12-3　监控分析中常见的查询函数及其作用

函　数	作　用	在 Nginx 日志分析中的作用
count	返回一个 field 字段的数量	计算某个 URI 在一段时间内的请求量，这对评估服务的请求变化有很大帮助，即 PV 查询
top	返回一个 field 字段中最大的值，支持返回 top N，即可以返回前 N 个较大的值	计算出返回请求量最大的值，也可以返回某个 URI 响应时间的最大值
percentile	返回一个 field 字段进行升序排序后的百分比的值	计算某个 URI 响应时间的 p90、p99 的值
mean	返回一个 field 字段的算术平均值	计算某个 URI 的平均响应时间
derivative	返回一个 field 字段的变化率	计算服务的波动率

将 Nginx 日志存放到 InfluxDB 之后，计算 URI 在后端的响应时间最大值，代码如下：

```
    >     select   top(upstream_time,3)   from   nginx   where   mysql_host_uri=
'"www.zhe800. com/"' and time > (now()-1m)  tz('Asia/Chongqing');
```

```
name: nginx
time                                top
----                                ---
2018-07-20T16:00:51.425490336+08:00 0.111
2018-07-20T16:01:02.480582679+08:00 0.102
2018-07-20T16:01:04.428871068+08:00 0.116
```

InfluxDB 还支持大量其他的函数，读者可以在业务中根据需求选择使用。自定义函数加入日志分析平台的方法可查看 https:// github.com/leehomewl/nginx_log_analysis。

12.6.4　数据存放之保留策略

Nginx 的实时日志进入 InfluxDB 后，会被及时分析生成报表和报警，当数据被使用完后，继续存放在 InfluxDB 中就会显得多余，因此需要对数据的存放设置时间限制，超过一定时间的数据可以被自动丢弃。这样可以节省 InfluxDB 的空间，以提升查询的性能。

InfluxDB 提供了保留策略来完成此操作。例如，在 InfluxDB 中创建一个数据库 Nginx，再对这个数据库创建一个 1h 的保留策略，存放时间超过 1h 的数据会被自动丢弃，这适合用来存放 Nginx 的实时日志，并作为默认值存在，代码如下：

```
CREATE DATABASE nginx
CREATE RETENTION POLICY "1_hours" ON "nginx" DURATION 1h REPLICATION 1
DEFAULT
```

如果在数据库 Nginx 创建一个 1 个月的保留策略，那么，1 个月前存放的数据就会被自动丢弃，这适合用来存放已经分析完成的报表数据，代码如下：

```
CREATE RETENTION POLICY "1_month" ON "nginx" DURATION 30d REPLICATION  1
```

创建完成后，可以通过如下命令来观察保留策略：

```
> SHOW RETENTION POLICIES ON nginx
name    duration shardGroupDuration replicaN default
----    -------- ------------------ -------- -------
autogen 0s       168h0m0s           1        false
1_hours 1h0m0s   1h0m0s             1        true
1_month 720h0m0s 24h0m0s            1        false
```

12.6.5　定时任务之连续查询

绝大部分的日志分析都以一段时间内的数据进行计算，如每分钟、每小时等。InfluxDB 还提供了连续查询的功能，可以在数据库上执行定时查询，并将执行后的结果存放在新的表中，这比依赖代码执行定时查询的方式更为简便。

例如要建立一个连续查询，它可以计算每分钟每条 URI 的 p90 的值，并将结果存放在保留策略是 1 个月的 tp90_nginx 的表中，代码如下：

```
   >CREATE CONTINUOUS QUERY tp90 ON nginx BEGIN SELECT percentile
(upstream_time, 90) AS tptime, mysql_host_uri INTO nginx."1_month".
tp90_nginx FROM nginx."1_hours".nginx GROUP BY time(1m), mysql_host_uri END
```

查看数据库中存在的连续查询如下：

```
   >show continuous queries
   name: _internal
   name query
   ---- -----

   name: nginxxx
   name query
   ---- -----

   name: mydb
   name query
   ---- -----

   name: nginx
   name  query
   ----  -----
   tp90   CREATE CONTINUOUS QUERY tp90 ON nginx BEGIN SELECT percentile
(upstream_time, 90) AS tptime, mysql_host_uri INTO nginx."2_month".tp90_
nginx FROM nginx."1_hours".nginx GROUP BY time(1m), mysql_host_uri END
```

12.6.6 客户端操作之 API

InfluxDB 提供 API 来操作数据库，所以可以让 Ngx_Lua 调用 API 去初始化 InfluxDB 的连续查询、创建数据库等，这样做能够减少加入的组件数量，代码示例如下：

```
   # curl -GET 'http://localhost:8086/query?pretty=true' --data-urlencode
"db=nginx"  --data-urlencode "q=select uri from nginx limit 1"
   {
       "results": [
           {
               "statement_id": 0,
               "series": [
                   {
                       "name": "nginx",
                       "columns": [
                           "time",
```

```
                        "uri"
                    ],
                    "values": [
                        [
                            "2018-07-20T07:00:00.10149097Z",
                            "/gateway/app/detail/graph"
                        ]
                    ]
                }
            ]
        }
    ]
}
```

12.6.7　使用 UDP 模式传输数据

InfluxDB 支持 TCP、UDP 两种协议，这里采用 UDP 模式来传输数据，在 InfluxDB 中开启 UDP 模式，需要修改配置文件/etc/influxdb/influxdb.conf。

下面是开启 UDP 的示例，默认访问的是 Nginx 日志的数据库：

```
    [[udp]]
      enabled = true
      bind-address = ":8911"
      database = "nginx"
      retention-policy = ""

     # These next lines control how batching works. You should have this
enabled
     # otherwise you could get dropped metrics or poor performance.
Batching
     # will buffer points in memory if you have many coming in.

     # Flush if this many points get buffered
      batch-size = 5000

     # Number of batches that may be pending in memory
      batch-pending = 10

     # Will flush at least this often even if we haven't hit buffer limit
      batch-timeout = "1s"

     # UDP Read buffer size, 0 means OS default. UDP listener will fail if
set above OS max.
      read-buffer = 0
```

12.7 利用 lua-resty-http 实现 API 交互

在 12.6.6 小节中提到 InfluxDB 支持使用 API 来执行 SQL 语句，那么 Ngx_Lua 如何调用这些 API 呢？这里推荐使用 lua-resty-http 模块。

注意：千万不要使用 Ngx_Lua 的子请求，在对外建立 HTTP 请求时，使用子请求既不“优雅”又会存在无法执行的可能性，如在 ngx.timer.at 中就无法使用子请求的 API。

12.7.1 安装 lua-resty-http

下载软件包，复制 lib 库到指定的 lua_package_path 路径：

```
# git clone https://github.com/pintsized/lua-resty-http.git
# cp lua-resty-http/lib/resty/* /usr/local/nginx/conf/lua_modules/resty/
```

12.7.2 使用方式

以 SQL 语句 select uri from nginx limit 1 为例，使用 lua-resty-http 来发送 HTTP 请求的代码如下：

```
location =  /q {
     content_by_lua_block {
          -- 加载模块
          local http = require "resty.http"
          local ngx = require "ngx"
          -- 定义 SQL 语句
          local post_data = "db=nginx&q=select uri from nginx limit 1"
          -- 初始化连接
          local hc = http:new()
          -- 发起请求
          local res, err=hc:request_uri('http://127.0.0.1:8086/query', {
              method = "POST", -- POST or GET
              headers  =  {["Content-Type"]  =  "application/x-www-form-
urlencoded" },
              body = post_data
          })
          if res.status ~= 200 then
             ngx.log(ngx.ERR,"SQL failed , " .. err)
             return
          end
          -- 输出结果
          ngx.say(res.body)
     }
}
```

执行结果如下：

```
   # curl  http://www.zhe800.com/q
   {"results":[{"statement_id":0,"series":[{"name":"nginx","columns":["time
","uri"],"values":[["2018-07-
20T08:00:00.101242877Z","/operation/abtest/pageconfig/v1"]]}]}]}
```

使用此模块配合 Ngx_Lua 提供的定时任务的指令，就可以核查数据库初始化连续查询及其他的配置了。

12.8　提升 InfluxDB 性能

虽然 InfluxDB 可以轻松完成数据分析，但 InfluxDB 开源版是单节点的，如果写入的数据量非常大，并且 SQL 语句执行频繁，就会存在性能问题，下面是在实际应用中需要注意和优化的细节。

- 需要使用固态硬盘，并且配置好缓存写入等条件，在压测中单节点可满足每秒 4 万条纪录的写入速度。
- 充分利用索引，把能够依赖索引进行查询的数据全部放在索引上，但索引的使用也要合理，如果存放几百万行的索引也会有性能问题。
- 数据量大的分组查询建议使用 InfluxDB 的持续查询功能，如可以创建一个每分钟执行汇总的查询。
- 如果使用 UDP 接收数据，可以调整 read-buffer 的参数来优化写入的性能。

12.9　小结

本章对日志集中存储所需要的工具进行了介绍，这些工具使 Nginx 的日志分析变得简洁、明了，详细的使用方案和代码都在 https://github.com/leehomewl/nginx_log_analysis 上。

第 13 章
静态容灾系统

我们一直在追求服务运行的稳定性，但总会发生一些意外导致服务稳定性下降甚至服务崩溃，影响服务稳定性的因素如下。

- 缺少性能压测。
- 缺少容错机制。
- 缺少服务的高可用设计。
- 大量的慢查询。
- 程序有 Bug，修复 Bug 又导致新的 Bug。
- 宕机、机房断电、断网。
- 一周上百次的更新。

……

针对这些问题大多数互联网公司都会制定一些防范措施，这其中就包括降级和容错设计。毕竟以现在微服务的发展状况，很难确保所有的服务都有合适的容错能力，更何况机房断网、光纤被弄断等第三方问题也屡屡出现。

那如何才能创建更加稳固的环境，让服务在出现异常时也能提供访问呢？

这就是本章要重点讲解的内容，利用 Nginx 搭建静态容灾系统，该系统具备高可用、高稳定性、智能化的服务体系，当后端服务出现异常时能提供降级功能，以减少重大事故的发生。

注意：静态容灾系统，是指在服务发生异常时使用静态化的数据供访问使用。静态化数据是指最近一段时间内用户访问的热点数据，例如列表页、详情页、专场活动页、首页、分类页等页面的访问数据。静态系统无法支持订单、购物车等动态服务。以电商类型的服务为例，供

用户访问的 80%的数据在异常时都可以通过降级提供过期数据，因此静态系统的任务就是让这 80%的数据在任何情况下都可以使用。

13.1　荆棘之路

一个完善的系统往往是在不断迭代中形成的，而迭代的原因常常是服务出现异常或有了新的功能需求。

根据缓存的使用情况，一般可以把服务分为 3 类。

1．使用简单缓存（可以直接配置 CDN 缓存）的服务，这些缓存的业务逻辑一般都很简单，缓存的 key 一般是 Host+URL，偶尔也会有一些特定的信息。常用的缓存系统有 proxy_cache、varnish、squid 等，它们的稳定性非常高，如果出现异常一般和硬件、网络有关。这些缓存系统还可以在后端服务器出现 5xx 等错误时提供过期的缓存数据。但这也可能出现一种极端的异常状况，那就是后端服务器响应的数据是错误的，却给出了 200 或 304 等正确的状态码，这会导致缓存系统缓存一份错误的数据。从整体上来说，这种缓存系统发生异常的概率很低。

2．使用复杂缓存（例如同一个请求需要根据登录用户的不同展示不同的结果，那么只用简单的 CDN 缓存是无法实现的）的服务，这些缓存一般由业务层控制。以推荐系统为例，它会给每个用户定制缓存，缓存的数据可以存放在 Redis、Memcached、Couchbase 等上面。这些缓存会受到应用层代码的影响，如果代码出现内存泄漏或性能问题，将会导致缓存无法被调用，从而导致页面无法正常展示。此类缓存发生异常的概率很高。

3．完全没有使用缓存的服务。这种服务大多数和订单页、购买页、个人中心页、抽奖页、优惠券页等有关，能够使用静态系统的比较少。异常发生的概率很高。因为它的业务体系不支持使用静态容灾，所以不在本章的讲解范围内。

当务之急是解决大概率事件，因此需要先来看第 2 类问题，对此有如下几种解决方案。

方案 1：提供两套数据，使用主备模式。

技术方案说明（主要涉及开发人员）：在代码层封装容错机制，设置严格的超时机制，并对数据的准确性进行判断。如果异常就进行容错降级，降级操作使用统一的数据模板，所有用户会看到相同的数据。

服务降级流程图如图 13-1 所示。

图 13-1　服务降级流程图

方案 1 解决了大部分问题，但在使用中仍然存在很多隐患，例如下面这些。

- 由于调用方自身的不稳定性，程序可能会“挂掉”。
- 由于微服务的分散性，会导致在每个服务下去设置这样的容错降级机制难度较大，难免会存在漏网之鱼。
- 破坏性测试的准确度会随着服务的容灾降级触发条件的不同而不同，测试难度较大。

方案 1 虽不能完美解决问题，但在实际应用中仍有广泛推广的必要，因为在代码设计上它是一个良好的示范。

方案 2：将一部分客户端请求随机保存到缓存系统中，让缓存区拥有热数据，当出现异常时将请求切换到缓存系统。缓存系统没有业务逻辑，只和硬件资源有关，因此稳定性极高。

技术方案说明（主要涉及运维人员）：通过 Cookie 定制，将很小一部分客户端请求固定发送到缓存系统中，通过 Nginx 的 error_page 功能，将 5xx、4xx 错误请求代理到缓存系统中，当后端服务响应缓慢或崩溃时，可手动配置 Nginx 转发到缓存系统。此方案简单而粗糙。

利用用户请求刷新缓存系统流程图，如图 13-2 所示。

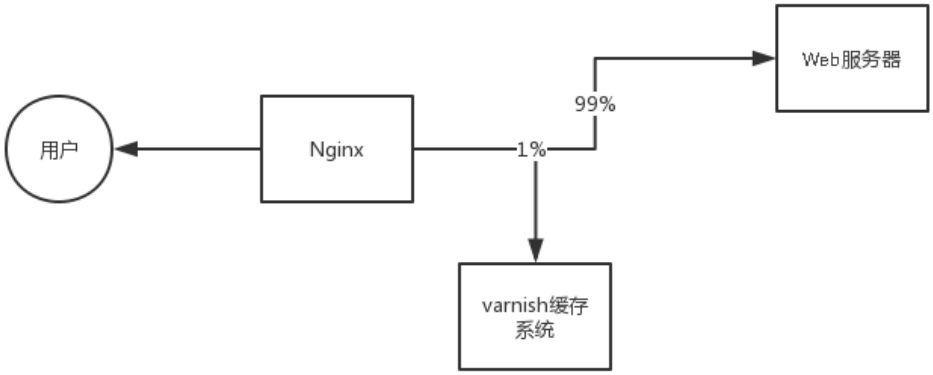


图 13-2　利用用户请求刷新缓存系统流程图

方案 2 的缺点：

- 会有一小部分用户信息始终存在于缓存中，无法实现全站的大数据推荐功能。
- 很难保证缓存系统的数据能够覆盖全部访问数据。

方案 3：自建爬虫模拟用户访问，将数据爬取后存放到缓存系统中，当出现异常时，将请求转发到缓存系统。

技术方案说明（涉及运维人员和开发人员）：通过 Nginx 的 error_page 功能，将 5xx、4xx 错误请求代理到缓存系统中，当后端服务响应缓慢或崩溃时，可手动配置 Nginx 转发到缓存系统。

爬虫方案流程图，如图 13-3 所示。

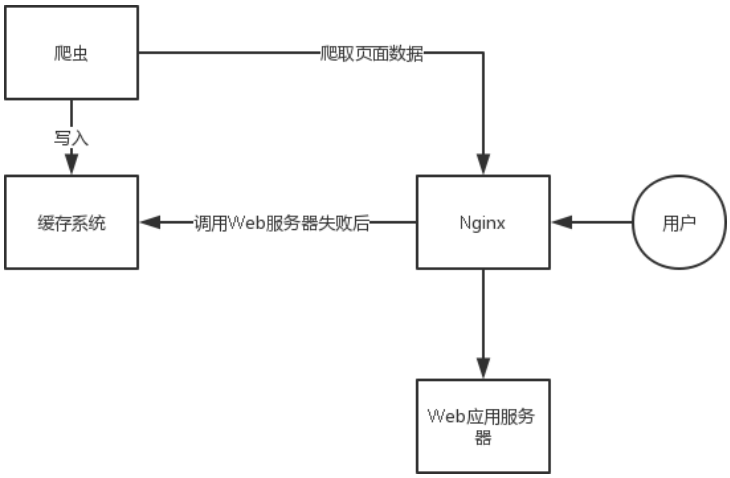


图 13-3　爬虫方案流程图

方案 3 的缺点：

- 页面数量庞大，爬虫模拟用户的方式进行爬取，会非常缓慢。
- JSON 格式的数据过多，爬虫需要获取页面上的这些接口，会增加开发难度。
- App 版本过多，且爬取 App 比较复杂，爬取到的数据可能会缺失很多。
- 无法满足自动切换的需求。

13.2　设计之路

对比 3.1 节中的 3 个方案，重新梳理容灾系统的流程，总结出容灾系统应具备如下特性。

1．动态配置，将 URL 服务分类——哪些支持容灾、哪些不支持，并全部存放在 MySQL

上，使用 Ngx_Lua 将 MySQL 数据读取到共享内存中。当服务出现异常时，Ngx_Lua 会改变请求的后端服务器走向，从而可以动态控制数据而不需要重启服务。

2．智能感应，在第 12 章中讲过 nginx_log_analysis 可以对 URL 的稳定性进行实时监控，所以可以根据请求的响应状态判断 URL 的稳定性，如果有异常情况发生，接口就会去修改 Ngx_Lua 的共享内存，将 URL 的请求代理到静态容灾的缓存系统中。如果是其他的监控系统，也同样适用，前提是该系统可以监控到 URL 服务的响应情况，例如 p90、平均值、可用性等和服务稳定性有关的数据。

3．热点数据，用户的请求日志就是热点数据，每隔一段时间要对日志进行去重操作，然后，将数据通过回放刷新到静态容灾的缓存系统中，这样可以解决因 App 版本不同而导致的 URL 爬取不全的问题。

4．多机部署，将静态容灾的缓存数据制作两套，可以使用主备模式，也可以直接串联在一起，但需要跨机房部署。（因为串联需要考虑网络的稳定性，所以如果是双机房，一般会使用光纤打通两个机房的内网，这样就没有跨机房的顾虑了。）

5．逻辑简单，整个系统的切换操作全部由 Nginx 来控制，研发团队只需在后台认领自己开发的 URL，并确认是否需要支持容灾即可。

6．持续维护，在 MySQL 中存放了所有已经被认领的线上的 URL 服务，如果开发人员需要新增一个 URL，在测试环境下提测时可以通过 Ngx_Lua 判断 URL 是否已被认领，如果没有，则在测试环境下无法进行访问，并会提醒开发人员到后台认领 URL，在认领过程中需要配置容灾等一系列属性。此方法不仅可以解决静态容灾问题，也可以对线上所有的 HTTP 服务进行控制和管理，是非常实用的功能。

7．智能核查，容灾系统并不是常用服务，很多时候它是处于离线状态的，毕竟线上服务不会每天都“挂掉”，所以需要设计一个灰度环境，以回归验证。

图 13-4 显示了 URL 服务认领平台的后台界面，非常简洁，维护成本也非常低。

图 13-4　URL 服务认领平台的后台界面

13.3　架构流程图

分析清楚静态容灾系统的需求和流程后，再来画一下流程图，通过流程图可以看出整个服务的走向和设计方案。

13.3.1　反向代理系统

反向代理系统主要负责分流，用来判断请求是去后端服务器还是去容灾系统。判断的方式由 Ngx_Lua 控制。Ngx_Lua 提供 API，可供监控系统调用。图 13-5 为反向代理切换容灾的流程图。

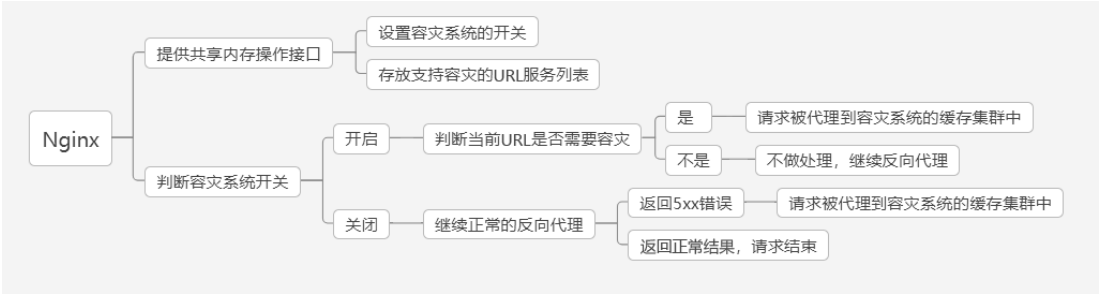


图 13-5　反向代理切换容灾的流程图

13.3.2　日志分析系统

日志分析系统的作用是确保 URI 服务的稳定性，URI 状态监控流程图如图 13-6 所示。

图 13-6　URI 状态监控流程图

13.3.3　后台系统

初始化数据源后，URL 是否具备容灾能力需要开发人员在后台配置。首次上线静态容灾系统前需要确保 URL 都被认领过，因为没被认领的 URL 无法进行容灾。后台管理流程图如图 13-7 所示。

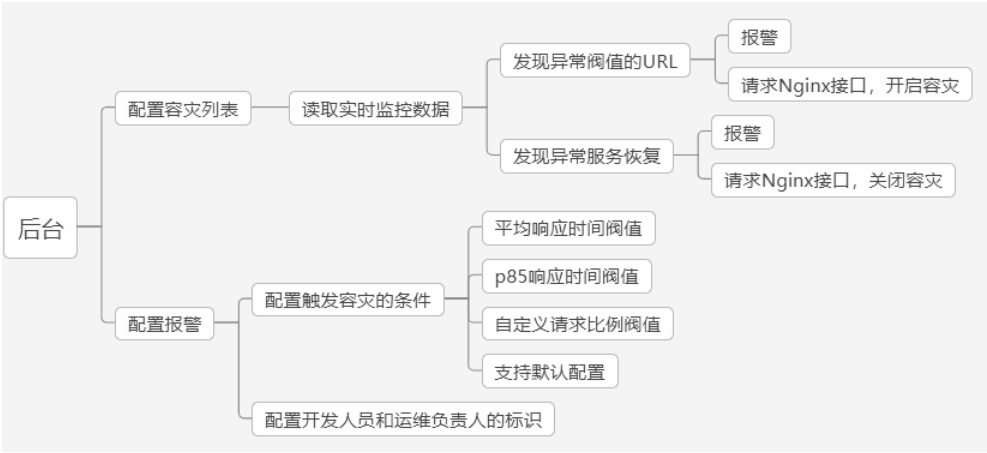


图 13-7　后台管理流程图

13.3.4　爬虫系统

爬虫系统主要负责从日志系统中获取支持容灾的 URL 清单（包含参数），对这些 URL 去重后定时爬取即可。

使用日志系统的数据的原因是：该数据是用户实时访问、真实存在的服务数据，属于热点数据，并可以确保缓存系统数据的完整性。爬虫系统流程图如图 13-8 所示。

图 13-8　爬虫系统流程图

13.3.5　容灾的缓存系统

主要作用是存放缓存数据，并且缓存的 key 是 Host+URL+时间版本。

爬虫系统是在收集一定时间内的数据后进行爬取的，这样缓存系统可以根据爬取频率生成时间版本，例如每 20min 爬取一次去重数据，那么 20min 就是 1 个版本，1h 就会有 3 个版本，1 天则有 72 个时间版本。容灾系统的缓存流程图如图 13-9 所示。

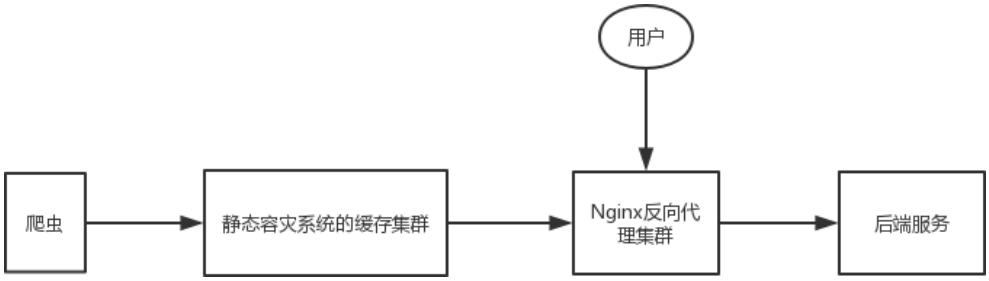


图 13-9　容灾系统的缓存流程图

从图 13-9 可以看出，当爬虫的请求到达缓存系统时，和用户的请求访问了相同的服务，并最终缓存到系统集群中。

13.3.6　时间版本的用途

引入时间版本主要是为了解决如下问题。

当在极端情况下，后端服务出现不可预期的错误时，如请求返回 200 的状态码，但请求内容是空或错误的信息，这样会导致容灾系统的数据也是错误的。如果有时间版本的存在，就可以将 URL 切换到之前的某个时间点，虽然这样找到的是过期数据，但至少能用。至于具体使用哪个时间版本，需要相关人员去后台进行切换，如果是自动切换，则只能切换到最近的时间版本。

时间版本还有一个好处，当某个请求在最新的时间版本内没有静态容灾缓存时，出现容灾后，Ngx_Lua 就会去搜索离它最近的时间版本。

13.3.7　异地容灾

缓存系统的数据如果可以跨机房存储，那么当主机房“挂掉”后，它还能利用备用机房提供网站访问的功能，从而加快全站的恢复速度。

技术方案说明：使用串联方式，双机房同时搭建缓存系统，并配置同样的缓存集群。如果两个机房由光纤打通变成内网，此时使用串联并不会消耗公网带宽；如果内网无法直接使用，就需要备份缓存系统到备用机房中，但这会消耗一定的外网带宽。

如果打算使用主备模式，则要避免缓存系统存在单点故障。异地容灾流程图如图 13-10 所示。

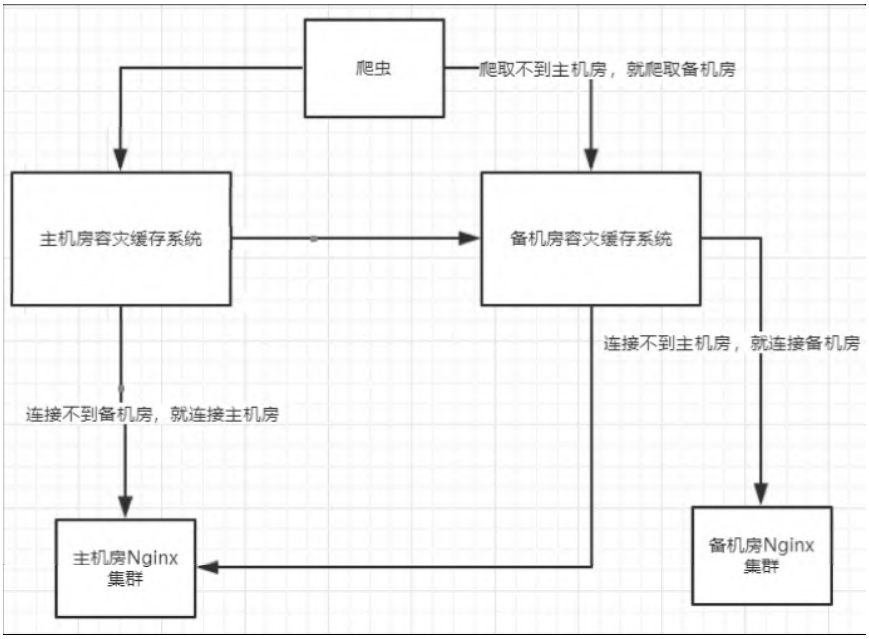


图 13-10　异地容灾流程图

13.4　核心代码解说

梳理完整体架构和流程，剩下的就是编写代码了，本节主要针对系统的核心代码进行说明，目的是让读者了解设计思路的实现方式和 Ngx_Lua 在整个流程中的作用。如果读者对全部代码感兴趣，请关注 nginx_log_analysis 的 Wiki。

13.4.1　Ngx_Lua 应用

反向代理 Nginx 中包含 Ngx_Lua 开发的代码，充分利用了 Lua API 的特性。首先在共享内存中存放一个 crash_status=0 的容灾标识，0 表示没有 URL 需要容灾，此时请求走正常的反向代理，只做简单的判断，对 Nginx 的开销是很小的；如果标识被设置成 1，表示监控系统已经修改了标识，并且存放了需要容灾的 URL 到共享内存中，此时须对当前 URL 进行判断并捕获到需要容灾的 URL，将其切换到容灾系统中。

关于时间，假如设计每 20min 为一个版本，需要容灾时会触发使用的版本，代码如下：

```
local ngx = require "ngx"
local ab_ver = tonumber(os.date('%H', ngx.time())) * 3
ab_ver = ab_ver + math.floor(tonumber(os.date('%M', ngx.time())) / 20)
```

关于容灾匹配，URL 一般分为精确 URL、正则 URL、目录 URL 3 种类型。当出现容灾时修改 Ngx_Lua 共享内存中的数据需要区分这 3 种类型，所以需要在后台中规范这 3 种 URL 的配置。这是性能开销最大的功能块，因为 Ngx_Lua 需要判断当前请求的 URL 属于哪个 URL 服务，才能决定是否切换到容灾系统。如果是精确 URL，只需查找 key/value 即可；如果是正则 URL 或目录 URL，就需要做正则匹配。正则匹配是 CPU 资源消耗最多的环节，如何减少 CPU 资源消耗是最核心的问题，请看下面的解决方案。

一般在互联网公司中，精确 URL 的使用比例是最高的，和业务相关的信息都存放在参数中，只有少部分 URL 会使用正则表达式。所以匹配流程应先匹配精确的 key/value，再匹配正则表达式。目前的压测显示，如果对所有的请求都进行正则匹配，QPS 会下降 8%左右，但在实际应用中不会有全都是正则 URL 的服务。

为了减少正则匹配的次数，建议研发团队尽量将正则 URL 修改成精确 URL，并将业务数据存放在参数中。对有正则匹配需求的 URL 可以缓存到 Lrucache 中，Lrucache 的数据结构具体如下。

- Key（键）：域名。
- Value（值）：table 类型，存放的是正则表达式的 URL，如 {URL1,URL2,URL3}。

在请求执行正则匹配前，会先查询精确的域名，域名对应成功后再通过 for 循环将 URL 读取出来进行匹配，下面是匹配 URL 的代码：

```
-- host 是当前请求的 Host，即 ngx.var.host
-- uri 是当前请求的 URI，即 ngx.var.uri
-- url_list 是存在正则表达式的 table，匹配到 host 后，就会获取到 url_list
local find_uri = function(host,uri,url_list)
    local res_uri
    for key, value_uri in pairs(url_list) do
    -- 循环读取进行正则匹配，添加 jo 参数，性能会更好
        local m, err  = ngx.re.find(uri ,  value_uri[1]  ,"jo")
        if m then
    -- res_uri 就是匹配到的 URI
            res_uri =  host .. value_uri[1]
        end
    end
    return res_uri
end
```

如果读者无法接受进行正则匹配时性能的损失，可以使用如下设计。

首先将包含正则表达式的 URL 使用 location 标识出来，此处可以利用 Nginx 原生 URL 的查询功能完成。

然后，将正则表达式的 URL 写成字符串格式的变量$ngx_uri，每个进入 location 的请求都是此字符串格式，当后台配置中存放的 URL 也是此格式时，Ngx_Lua 查询就只是精确匹配字符串的问题了，代码如下：

```
    location ~ ^/a/[0-9]+$ {
# 把变量$ngx_ur 设置为和 URL 完全一样的格式，Ngx_Lua 需要获取这个变量
# 且$find_uri 的值要和在 MySQL 中存放的 URL 一样，这样就可以直接进行精确匹配了
        set $ngx_uri '/a/[0-9]+$';
        proxy_pass http://ups;
    }
```

此方法可以将正则匹配和目录匹配全部转换成精确匹配，性能会提升不少，但 Nginx 的配置中也会因此出现很多变量，读者需要根据自身业务需求和现有服务器资源进行选择。

13.4.2 爬虫和日志系统的关系

在第 12 章中讲过日志分析系统 nginx_log_analysis 中的数据库 InfluxDB 会临时存放 Nginx 日志，该日志包含所有支持容灾的原始请求。因此只需读取 Nginx 日志中的数据，并进行去重操作，然后批量刷新容灾缓存系统即可。

读者可能会担心，爬取服务会增加服务器的压力从而出现性能问题。这个问题笔者也考虑过，那么，不妨先做如下的分析（以每 20min 为一个时间版本为例）。

- 以京东商城的 App 为例，用户的常见操作大概有几十种，虽然每个用户的数据不一样（受推荐系统等影响），但 URL 的格式基本一致。例如首页、活动页、列表页等，每天的访问量轻松过亿，但去重后每 20min 也就几百条数据，事实上，大部分服务都是如此。
- 控制爬取速度，可以将目前需要爬取的 URL 总数分摊到 1200s 内进行均匀爬取，以减少并发带来的压力。
- 如果仍然无法满足需求，可以将时间版本从 20min 改为 30min 甚至 1h 来试试，绝大多数情况下 20min 是没有问题的。

13.4.3 全部容灾和部分容灾功能

全部容灾更适合用于后端服务已经没有任何能力提供响应的情况，但有时只是因为资源紧张导致部分请求响应异常，在这样的情况下使用全部容灾是不明智的选择，所以就有了部分容灾功能，即当服务压力大时触发的降级功能。

方案说明如下。

- 通过日志分析系统可以确认在 URI 服务中有多大比例的请求出现了异常。

- 获取到该比例后（如 35%的请求缓慢或报 5xx 错误），监控系统将发送请求修改 Nginx 的共享内存，并将该比例作为参数传入，例如 p=35 代表 35%的请求需要进入容灾系统。
- 在 Lua 代码中执行 1~100 的随机数计算。将小于或等于 35 的请求传入容灾系统，大于 35 的请求继续传到后端服务器上，利用 lua-resty-random 模块可以完成随机数生成。
- 如果进行如此操作后，性能依然没有明显好转，可以再去触发全部容灾功能。

13.5　静态容灾的智能关闭方案

切换静态容灾后，用户的请求会部分或全部进入容灾系统，后端服务器就无法接收到正常数量的请求，而研发团队在问题解决后也很难判断服务是否已经恢复正常，因此何时关闭容灾功能也是个问题。

如何才能确定服务已经恢复了呢？在实际应用中笔者先后使用过 3 种方案。

每种方案的主要流程都是在进入容灾系统时，将用户的请求复制一份重新发回线上服务器。通过这种方式可以让后端服务器维持真实流量，问题是否被修复很快就可以知晓了。下面将会对这 3 种方案一一进行介绍。

13.5.1　从日志分析系统中复制请求

日志分析系统中存放有实时的日志请求（在 InfluxDB 中），将被切换到容灾系统的 URL 请求从 InfluxDB 中读取出来，然后发送到反向代理的 Nginx 集群上，模拟用户请求（这些请求理论上都应该使用 GET 方法）。

此方案有个问题，即发送模拟请求时，无法完全模拟真实的并发情况，因为请求的数据是从已经完成的日志中读取出来的，这中间存在时间上的延迟，并且 QPS 也不好控制，会导致回放不够真实，功能不稳定。

13.5.2　利用 goreplay 复制流量

后来笔者引入了 goreplay 来解决复制请求不真实的问题。goreplay 是用 Go 语言开发的流量复制工具，使用起来非常简单。在静态容灾缓存系统的服务器上开启 goreplay，一旦有请求到达缓存系统就证明有了服务容灾，那么请求就会被 goreplay 同步复制到线上的服务器，这样请求的并发效果就和真实的请求一致了。goreplay 的安装方法如下：

```
# wget \
https://github.com/buger/goreplay/releases/download/v0.16.1/gor_0.16.1_x
```

```
64.tar.gz
    # tar -zxvf gor_0.16.1_x64.tar.gz
    # cp  goreplay /usr/bin
```

举个例子，将所有 GET 方法的请求流量复制到 IP 地址为 192.168.1.15 的服务器上，并且保留原始的 Host 头信息，代码如下：

```
    # goreplay --input-raw :80 --output-http="http://192.168.1.15" -http-
allow-method GET --http-original-host
```

13.5.3 Nginx 的镜像功能

goreplay 很好用，但如果不想引入更多的组件，也可以使用 Nginx 原生的工具。Nginx 1.13.4 版本新增了 ngx_http_mirror_module 模块，使用它可以便捷地制造镜像请求，OpenResty 1.13.6.2 版本也包含此模块，所以果断将容灾系统缓存集群代理升级为 OpenResty 1.13.6.2 吧。

一般进入容灾系统的请求只有客户端请求和爬虫请求（爬虫的作用是为了缓存数据），这里需要将客户端请求通过镜像功能复制一份到真实环境下的反向代理上。因此需要为爬虫的请求配置特殊的请求头，用来和客户端请求进行区分。然后，对非爬虫的请求配置镜像功能即可，代码如下：

```
    location / {
        # 启动镜像
        mirror /crash_mirror;
        # 禁止发送请求体
        mirror_request_body off;
        # 容灾系统的缓存配置
        proxy_cache crash_cache_store;
    }

    location /crash_mirror {
        internal;

        # 如果是爬虫服务，不需要进行镜像配置
        if ($http_crw_rd = 'spider_static_crash')  {
           return 411 ;
        }
        proxy_pass_request_body off;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

        # 保留请求的完整的 URL，代理到反向代理的 Nginx 集群上
        proxy_pass http://nginx_servers$request_uri;
```

```
        # 表明是镜像请求， Nginx 发现此头就会将请求代理到真实的后端服务器上
        proxy_set_header C-Mirror 1;
    }
```

此方案可以使请求的并发速度和用户的真实请求几乎能够保持一致，而且配置非常简单。

13.5.4　灰度验证容灾系统缓存

容灾系统只有在出现异常时才会提供服务，所以该如何确保容灾系统一直处于稳定状态呢？除各项存活监控外，还可加入灰度验证方案，具体如下。

- 搭建一个灰度的反向代理 Nginx，在 upstream 里只保留容灾缓存系统的地址，剩下的 upstream 服务器配置都写成错误的地址，这样就模拟了线上服务器全部异常的情况。
- 在灰度的反向代理 Nginx 的共享内存中开启全部 URI 服务的容灾，表明所有支持容灾的服务全部进入了容灾系统。
- 自建一个 DNS 服务并进行配置，将所有与网站有关的域名都指向这个 Nginx。
- 将 PC 端或手机端 App 的 DNS 配置为自建的 DNS 系统，这样访问的域名就都会被解析到该 Nginx 上，并进行服务验证。
- 如果在验证后页面仍然可以使用，就证明容灾系统是正常的；如果有些页面打不开，则要考虑容灾系统是否有问题。一般可能出现的问题是新增了 URI 服务，却没有配置容灾功能或在容灾系统的缓存中没有数据。

13.6　小结

本章主要以思路讲解为主，Ngx_Lua 在架构上有很大的灵活性，它的代码其实没有太高的复杂度，只需在每个阶段合理地安排业务逻辑就可以完成整套智能化的静态容灾系统了。另外本章涉及的代码会进行开源，开源地址为 https://github.com/leehomewl/nginx_log_analysis。

第 14 章
深入挖掘反向代理

利用 Nginx 的反向代理功能，可以做出很多通用的组件，这些组件将会降低后端业务的复杂度。本章将会重点介绍如何充分利用 Nginx 的反向代理功能来提升系统架构的灵活度。

14.1 验证码防御中心

相信大部分互联网平台都接到过各式各样的非正常请求，如恶意爬虫、流量攻击、撞库等，它们严重干扰了网站的安全和稳定，为了抵御这些非正常请求，验证码拦截服务应运而生。

注意：本节的验证码防御功能主要针对恶意请求而设计。

一般在开发验证码对服务进行拦截的初期，只有个别的应用服务会使用验证码功能，应用服务器负责和验证码服务交互对请求进行拦截和验证。但慢慢地会有更多服务需要使用验证码的拦截功能，这就增加了应用服务器的代码耦合度，并且一旦验证码服务有新功能变更，应用服务器都可能需要更新代码来适应它。

图 14-1 所示为应用层调用验证码服务的流程图，图中风控中心的作用是对服务 a、b 的请求进行过滤，一旦发现可疑请求，就会触发验证码服务。为了降低业务代码的复杂度，这里将验证码服务的逻辑切换到了 Nginx 上，利用 Nginx 反向代理的功能实现验证码拦截透明化。使用 Nginx 和验证码服务交互需要解决如下问题。

- 如果拦截的请求是 HTML 的页面请求，可以将请求跳转到验证码页面，验证通过后再跳转回当前页面，也可以直接在当前页面弹出验证码窗口。
- 如果拦截的请求是接口，则不能做页面跳转操作，而应在当前页面弹出验证码窗口。
- 是否在当前访问地址上弹出验证码窗口需要前端 JS 代码进行判断，可以通过判断请求

的内容或状态码来完成页面加载验证码的功能。

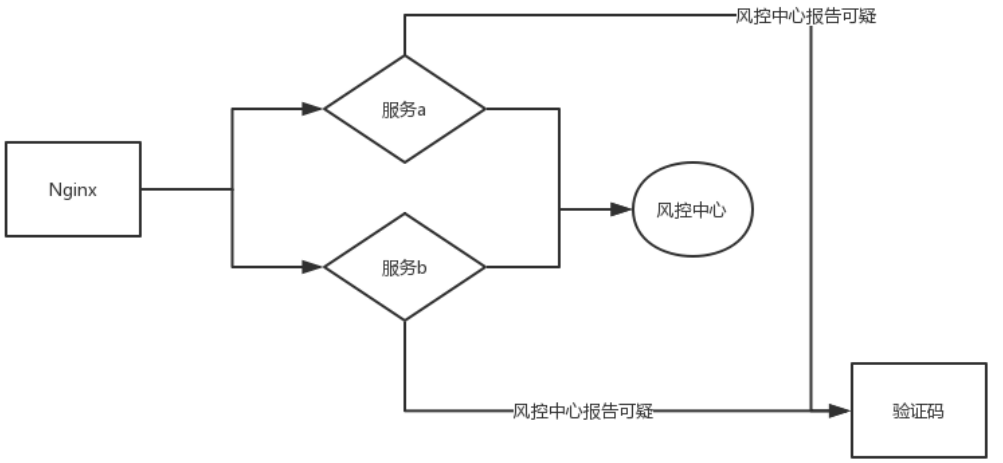


图 14-1　应用层调用验证码服务的流程图

那么问题来了，如果将 HTML 页面跳转到验证码页面进行验证，首要的问题是如何区分页面和接口。因此需要获取 URL 的属性，确认它是否为 HTML 属性。

还记得之前谈论过的 nginx_log_analysis 日志分析系统吗？它可以记录 URL 返回的 Content-Type 响应头。Content-Type 存放的是“text/html”或“text/css”等属性，通过这些可以获取 URL 的格式（JSON、HTML 或 JPG），也就可以轻松地判断 URL 是否为 HTML 属性了。（这些数据存放在 nginx_log_analysis 系统的 MySQL 数据库中，每个 URL 都有对应的 Content-Type 属性。）

注意：如果在项目服务中存在不发送 Content-Type 响应头的服务，请务必添加上该响应头，它是一个服务的基础属性，即使没有验证码服务，此属性也是必须的。

下面是 Nginx 在验证码拦截功能中的设计流程。

- Ngx_Lua 提供共享内存 A 来存放可疑的 IP 地址、User_Agent 或 Cookie 等标识。
- Ngx_Lua 提供共享内存 B 来存放 Content-Type 标识为 HTML 的 URI。
- 风控中心将可疑的标识或 IP 地址通过 Ngx_Lua 的 API 存放到共享内存 A 中。风控中心一般用来检查请求的危险等级，例如检查来自该 IP 地址的请求次数是否过于频繁，是否有来自某 IP 地址的请求一直在尝试访问敏感服务等。
- 当共享内存 A 中记录了可疑请求的属性后，Ngx_Lua 就会访问共享内存 B，检查此 URI 是否为 HTML 属性。如果是，则跳转到验证码验证页面，要求用户输入验证码；如果不是，则返回 HTTP 状态码（如 421）。前端页面在加载过程中发现特定的 HTTP 状态

码后，就会在当前页面弹出验证码窗口，要求用户进行验证。

- 当用户输入验证码后，会执行验证码服务进行验证，如果验证成功则通知风控中心从可疑的 IP 地址中去掉该 IP 地址；如果验证不成功，就一直停留在验证码处要求当前请求输入验证码。

Nginx 验证码交互流程图如图 14-2 所示。

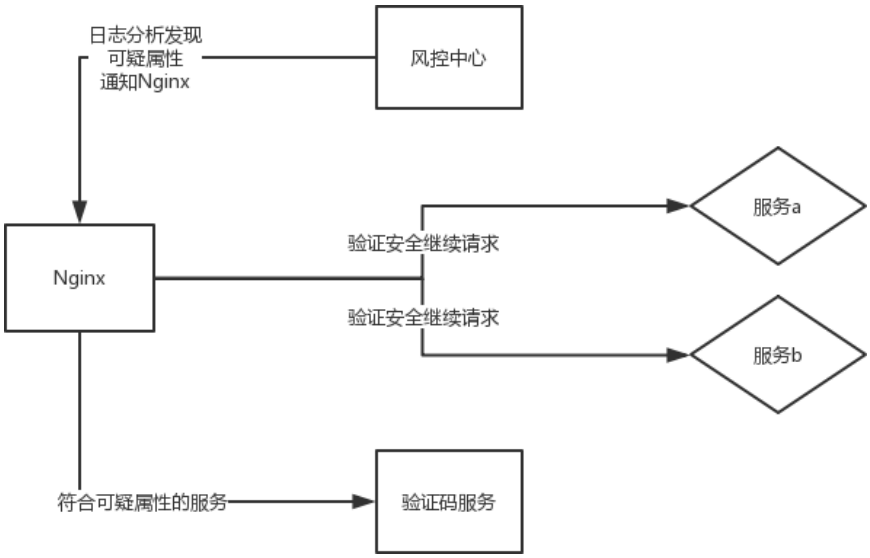


图 14-2　Nginx 验证码交互流程图

经过这样的改造，应用服务器不必直接和验证码服务打交道了，从而简化了代码逻辑，并且在验证时使用了 Ngx_Lua 的异步性能，降低了资源使用率。

14.2　鉴权管理中心

鉴权管理中心的作用是验证用户是否拥有访问某系统的权限，它的使用非常广泛。笔者最初使用鉴权管理中心时，也是让应用服务器直接和它打交道，但随着应用服务越来越多，这项工作分散在各个系统中，维护和调度的复杂度越来越大，这和 14.1 节中的验证码拦截功能遇到的问题相似，所以为了降低代码耦合度，笔者决定使用 Nginx 和鉴权管理中心交互。

14.2.1　利用 auth_request 管理鉴权

Nginx 在 1.5.4 版本中引入了 ngx_http_auth_request_module，它可以在当前请求发送给后端

服务器前，先发送子请求到另一个服务器上，如果子请求返回的状态码是 2xx，那么就会继续执行当前请求；如果子请求返回的是 401 或 403 等状态码，则当前请求会被拒绝并返回错误信息给客户端。该模块的激活方式是在编译时加入--with-http_auth_request_module。利用 auth_request 管理鉴权的代码示例如下：

```
location /order/ {
    # 在用户请求到达应用服务器前，先请求/check_auth 来验证权限
    auth_request /check_auth;
    # 代理到应用服务器上
    proxy_pass http://ups_servers;
}

location = /check_auth {
    proxy_pass http://代理到鉴权中心
    proxy_pass_request_body off;
    proxy_set_header Content-Length "";
    proxy_set_header X-Original-URI $request_uri;
}
```

通过 auth_request 模块的使用，客户端和鉴权服务的交互都交由 Nginx 来完成；又因为鉴权发送的是子请求，所以性能也十分出色。

14.2.2　利用 Ngx_Lua 子请求实现鉴权功能

在鉴权管理中，也可以利用 Ngx_Lua 来实现鉴权功能。Ngx_Lua 的配置更为灵活。下面的示例是利用 Ngx_Lua 模仿 ngx_http_auth_request_module 的方式设计的：

```
# 客户端请求/orde/下面的 URL
location = /order/ {
    access_by_lua_block {
        local ngx = require "ngx"
       -- 发送子请求到鉴权服务/check_auth
            local res = ngx.location.capture("/check_auth")
            if res.status == 200 then
       -- 如果鉴权返回 200 状态码，则鉴权成功，退出当前阶段，继续下一个 Nginx 执行阶段
                ngx.exit(ngx.OK)
            else
       -- 如果鉴权返回非 200 的状态码，则把返回鉴权中心的状态码传递给客户端，请求停止
                ngx.exit(res.status)
            end
    }
    proxy_pass http://test_servers;
}
location = /check_auth {
```

```
    # 请求鉴权中心，并将原始请求的 URI 发送过去，也可以通过请求头传送
        proxy_pass http://127.0.0.1:82/$request_uri;
        proxy_pass_request_body off;
        proxy_set_header Content-Length "";
        proxy_set_header X-Original-URI $request_uri;
    }
```

假如有 200 个 URI 分散在不同的应用服务器中，按照上面的配置就需要很多个 location 来完成，而且每次新增 URI 都需要去修改鉴权配置并重载 Nginx 进程。这看上去并不“优雅”。为了让上述配置更加灵活，可以使用下面的重构方案，该方案不仅可以解决 URI 较多时配置烦琐的问题，还可以提供降级功能（当鉴权中心出现异常时，通过降级来保证客户端的服务能够正常访问）。重构方案如下。

- 将上述配置中 access_by_lua_block 的配置从 location 块更换到 server 块或 http 块。

注意：此处不建议使用 http 块，因为在 http 块执行会导致 Nginx 下所有 server 的请求都被过滤掉，但在现实环境中不是所有域名都需要鉴权的，对不需要鉴权的域名（如静态资源 CSS、JS、JPG 等文件）进行过滤会消耗额外的 CPU 资源。因此，除非能够确认 Nginx 下所有域名的 URI 服务都需要鉴权，否则，尽量不要使用 http 块。

- 将需要鉴权的 URI 服务的数据存放在后台管理系统上（可以使用第 13 章介绍的数据存放方案），在 URI 发生变更时同步这个数据到 Ngx_Lua 的共享内存中，这样 Nginx 就可以得到需要鉴权的所有 URI 服务了，且不必去重启服务，提升了灵活性。
- 对当前客户端请求进行判断，如果请求的 URI 匹配到了共享内存中的数据，就表示需要进行鉴权，此时就发送子请求去鉴权。
- 当鉴权中心出现异常时，如果鉴权中心响应缓慢，可以采取限流功能，将需要强制鉴权的 URI 服务保留在共享内存中，其他 URI 全部删除，让核心服务优先鉴权。如果鉴权中心彻底崩溃，可以根据业务规则确认是否全部停止鉴权，全部停止的方式就是清空共享内存中的数据。

14.3 并行访问

在 7.13 节曾经讲过，在 Ngx_Lua 中可以使用子请求的指令 ngx.location.capture 和 ngx.location.capture_multi 来实现内部并发的需求，本节将会介绍一种灵活轻巧的并发方式，它就是 Ngx_Lua 模块中的轻量级线程（以下简称轻线程）。

14.3.1　轻线程的启动和终止

轻线程是 Ngx_Lua 模块中的特殊协程，称为 light thread。下面先来讲一下 Ngx_Lua 轻线程的启动和终止方式。

指令：ngx.thread.spawn

语法：co = ngx.thread.spawn(func, arg1, arg2, …)

环境：rewrite_by_lua*、access_by_lua*、content_by_lua*、ngx.timer.*、 ssl_certificate_by_lua*、ssl_session_fetch_by_lua*

含义：创建一个新的轻线程来执行 Lua 函数 func，可以加入可选参数 arg1、arg2 等供 func 函数使用。默认情况下，请求在轻线程全部执行后，或者轻线程被异常终止时才能够结束。可以在轻线程执行中使用 ngx.exit、ngx.exec、ngx.redirect、ngx.req.set_uri(uri, true)指令来终止轻线程。

请看下面的示例，并观察轻线程的使用情况：

```
location / {
    content_by_lua_block {
        local ngx = require "ngx";
        local function func_say(a)
             -- 休眠 a 秒，不会阻塞 worker 进程
             ngx.sleep(a)
             ngx.say(a)
        end
        ngx.thread.spawn(func_say,'1')
        ngx.thread.spawn(func_say,'4')
        ngx.thread.spawn(func_say,'3')
        ngx.thread.spawn(func_say,'2')
     }
}
```

执行结果返回如下：

```
[root@testnginx ~]# curl http://172.19.3.41/
1
2
3
4
```

如果按照串行输出的方式，或者按照先请求先返回就先输出的方式，得到的结果应该是“1，4，3，2”，但很显然返回的结果并非如此，它们呈现如下特点。

- 是并发执行的结果，先返回的结果会先输出，和配置函数的前后顺序没有关系。
- 请求会等所有的轻线程执行完成后才返回数据。

再看下面的例子，观察轻线程的终止操作：

```
location / {
    rewrite_by_lua_block {
        local ngx = require "ngx";
        local function func_say(a)
             ngx.sleep(a)
             -- 如果 a 等于 3，则退出当前执行阶段
             if a == '3' then
                ngx.exit(0)
             end
             ngx.say(a)
        end
        ngx.thread.spawn(func_say,'1')
        ngx.thread.spawn(func_say,'4')
        ngx.thread.spawn(func_say,'3')
        ngx.thread.spawn(func_say,'2')
    }
}
```

执行结果如下：

```
[root@testnginx ~]# curl http://testnginx.com/
1
2
```

输出结果只有“1”和“2”，当轻线程执行到 a==3 时，被 ngx.exit(0)终止了，并导致剩下的轻线程都被终止，所以 a==4 的函数也没有输出结果。

14.3.2 等待和终止轻线程

通过 ngx.thread.spawn 可以使 Ngx_Lua 完成轻线程的执行，但还需要对轻线程的执行情况进行处理，让并发执行变得更加灵活。

指令：ngx.thread.wait

语法：ok, res1, res2, … = ngx.thread.wait(thread1, thread2, …)

环境：rewrite_by_lua*、access_by_lua*、content_by_lua*、ngx.timer.*、ssl_certificate_by_lua*、ssl_session_fetch_by_lua*

含义：等待 thread1、thread2 等轻线程（在 ngx.thread.wait 中可以有一个或多个轻线程）

的执行结果，并返回第一个执行完成的轻线程的结果。

在实际使用中，很少会有只需要第一个返回结果的情况，一般都需要获取全部信息，下面的例子将会介绍如何等待并获取所有的结果：

```
location / {
    rewrite_by_lua_block {
        local ngx = require "ngx";
        local function func_say(a)
            ngx.sleep(a)
            return a
        end

        local threads = {
           ngx.thread.spawn(func_say,'1'),
           ngx.thread.spawn(func_say,'4'),
           ngx.thread.spawn(func_say,'3'),
           ngx.thread.spawn(func_say,'2')
        }
        -- 循环读取轻线程的结果
        for i = 1, #threads do
            local ok, res = ngx.thread.wait(threads[i])
            if not ok then
                ngx.say(i, ": failed to run: ", res)
            else
                ngx.say(i, " res: ",res)
            end
        end

    }
}
```

执行结果如下：

```
[root@testnginx ~]# curl http://testnginx.com/
1 res: 1
2 res: 4
3 res: 3
4 res: 2
```

在执行过程中，如果需要终止正在执行的轻线程，可以使用如下指令。

指令：ngx.thread.kill

语法：ok, err = ngx.thread.kill(thread)

环境：rewrite_by_lua*、 access_by_lua*、content_by_lua*、ngx.timer.*

含义：终止通过 ngx.thread.spawn 创建的轻线程。终止成功则返回 true，其他情况则返回错误描述信息。只有父协程可以终止轻线程，但无法终止 Nginx 下触发的子请求操作，如使用 ngx.location.capture 执行的子请求。

```
location / {
    rewrite_by_lua_block {
        local ngx = require "ngx";
        local function func_say(a)
            ngx.sleep(a)
            ngx.say(a)
        end

        local a = ngx.thread.spawn(func_say,'1')
        local b = ngx.thread.spawn(func_say,'4')
        local c = ngx.thread.spawn(func_say,'3')
        local d = ngx.thread.spawn(func_say,'2')
        -- 判断 b 轻线程是否还在运行，如果是，则终止它
        if coroutine.status(b) == 'running' then
           ngx.say(coroutine.status(b))
           local ok, err = ngx.thread.kill(b)
        -- 终止 b 轻线程后，输出 b 轻线程目前的状态
           ngx.say('kill 了 b 轻线程，b 轻线程目前的状态是',coroutine.status(b))
        end
    }
}
```

执行结果如下：

```
[root@testnginx ~]# curl http://testnginx.com/
running
kill 了 b 轻线程，b 轻线程目前的状态是 dead
1
2
3
```

上述配置使用了指令 coroutine.status，它是 Lua 用来检查协程的运行状态的，协程的运行状态有 3 种，即 dead、suspend、running。

14.3.3 URL 的外部合并和内部并发

在 HTTP 1.1 中，页面加载的请求过多会降低加载效率，因此在优化 Web 性能时通常会对 URL 的数量进行控制。

控制 URL 数量有多种方式，本节会以合并 URL 的方式来优化请求数量，将多个请求的 URL 合并成一个 URL 的好处如下。

- 减少 TCP 建立连接开销。
- 避免触发浏览器对同一个域名请求数量的限制。
- 部分 URL 的响应内容非常少，无法进行响应内容压缩，合并后响应内容的大小达到压缩条件后就会触发内容压缩，从而节省带宽。

外部 URL 的合并和业务有关，本节不做过多说明，下面主要讲解一下当请求合并进入 Nginx 后，如何实现内部并发的功能，下面是相关思路说明。

- 合并后的 URL 将每个业务需要的数据以不同的参数进行分配。
- 每个参数代表一个业务逻辑，通过 Nginx 分发给不同的应用服务器或不同的数据库。
- 应用服务器或数据库将响应内容反馈给 Nginx，Nginx 合并这些数据并返回给客户端。

那么，如何完成内部的并发请求呢？这就需要用到轻线程了，示例如下：

```
location / {
    content_by_lua_block {
        local ngx = require "ngx";
        -- 加载 lua-resty-memcached 模块，读取 memcahed
        local memcached = require "resty.memcached"
        -- 加载 lua-resty-http 模块，发起外部 HTTP 连接
        local http = require "resty.http"
        local httpc = http.new()

        -- 读取 memcahed 函数
        local function op_memcached(key,timeout)
            local memc = memcached:new()
             memc:set_timeout(timeout)
            memc:connect("127.0.0.1", 11211)
            local res, err = memc:get(key)
            local ok, err = memc:set_keepalive(10000, 10)
            return res
        end
        local function func_say(a)
            ngx.sleep(a)
            return a
        end

        -- Ngx_Lua 的子请求
        local function capture_req_http()
            local res = ngx.location.capture("/test_ngx_lua")
```

```
            return res.body
        end
        -- 发起外部 HTTP 请求的函数
        local function cosocket_req_http()
            local httpc = http.new()
            httpc:set_timeout(timeout)
            local res, err = httpc:request_uri("http://127.0.0.1:8011/
share_set", {
                method = "GET",
                body = "a=1&b=2",
                keepalive_timeout = 60,
                keepalive_pool = 10
            })
            return res.body
        end

        -- 读取 URL 中的参数
        local memc_get = ngx.var.arg_mc_get
        local la_say = ngx.var.arg_la_say
        local lb_get = ngx.var.arg_lb_get
        local lc_get = ngx.var.arg_lc_get
        -- 声明一个 table 用于存放轻线程
        local threads = {}

        -- 判断参数是否存在，如果存在就执行和此参数有关的代码逻辑
        if memc_get then
            -- 函数 op_memcached 中 memcached 的 timeout 参数值为“1”
            -- 这样就可以灵活控制请求的返回时间，避免长时间等待了
            -- 后面的几个函数与此相似，不再进行说明
            table.insert(threads,ngx.thread.spawn(op_memcached,memc_get,1))
        end
        if la_say then
            table.insert(threads,ngx.thread.spawn(func_say,1))
        end
        if lb_get then
            table.insert(threads,ngx.thread.spawn(cosocket_req_http,1))
        end
        if lc_get then
            table.insert(threads,ngx.thread.spawn(capture_req_http))
        end
        -- 循环读取轻线程的结果
        for i = 1, #threads do
            local ok, res = ngx.thread.wait(threads[i])
```

```
                if not ok then
                    ngx.say(i, ": failed to run: ", res)
                else
                    ngx.say(i, " res: ",res)
                end
            end

        }
    }
```

执行结果（涉及返回数据的业务没有进行配置，仅显示输出的内容和格式）如下：

```
    #   curl 'http://testnginx.com/?mc_get=testnginxlua&la_say=1&lb_get=1&lc_
get=1'
    1 res: bar
    2 res: 1
    3 res: share_set!!
    4 res: test_ngx_lua!!!
```

通过参数来判断所要执行的业务逻辑，可以使接口的灵活度大大提升。但如果业务逻辑变化极少，传递到后端服务的请求基本是固定的，可以不使用参数进行传递，只需要请求 URL，让 Ngx_Lua 固定分发给应用服务器即可。

14.3.4　使用 cosocket 实现外部访问

在上一小节的配置中出现了 lua-resty-http 模块，它是基于 cosocket API 开发的，在 HTTP 的使用场景中，与 Ngx_Lua 子请求的指令相比，lua-resty-http 可以直接向外部发送 HTTP 请求，并且在响应数据内容过大的情况下 lua-resty-http 性能更好，也更加灵活。

14.4　小结

本章深入挖掘了 Nginx 反向代理的一些功能，利用这些功能可以简化服务的逻辑，提升系统的稳定性。Nginx 可以使用的功能还有很多，希望各位读者能够积极进行探索。

第 15 章
爬虫

打开门来做生意，遇到爬虫是难免的。它们有的来自各大知名的搜索引擎，有的来自名不见经传的网站，也有可能来自某个人。如果遇到以上种种，可能会出现如下情况。

- 出现服务性能问题，拖慢用户访问速度，甚至出现服务崩溃、网站瘫痪的情况。
- 网站数据被窃取，公司的业务数据被暴露给竞争对手。

本章将会对爬虫的不同情况进行分析，并结合 Nginx 来介绍如何控制爬虫的行为。

15.1 区分搜索引擎爬虫和恶意爬虫

爬虫大致可分为两类：搜索引擎的爬虫和恶意爬虫。那么如何区分它们呢？可以使用排除法，搜索引擎的爬虫都带有明显的 User-Agent 特征，例如搜狗的 Sogou web spider/4.0 和 Sogou inst spider/4.0、百度的 BaiduSpider/2.0。当然标识也可以伪造，所以很多恶意爬虫会伪造 User-Agent 进行访问。如何区分伪造爬虫呢？流程如下。

- 确认搜索引擎爬虫的 User-Agent，这在各大搜索引擎的网站上都有标注，如搜狗的 User-Agent 可以查看官方 Wiki，网址为 http://help.sogou.com/spider.html。
- 从日志中获取符合这些爬虫的 User-Agent 的 IP 地址。
- 使用 Linux 命令 nslookup 获取反向解析，得到此 IP 地址对应的域名。
- 得到的域名如果不是搜索引擎公司的域名，那么这些爬虫就属于伪造爬虫。

下面是 Nginx 日志的几条爬虫记录：

```
   123.125.71.29   www.zhe800.com   -   [23/May/2018:18:59:07   +0800]   "GET
/ju_deal/doudouxiao_31986454 HTTP/1.1" 200 671 "-" "Mozilla/5.0 (compatible;
```

```
Baiduspider/2.0; +http://www.baidu.com/search/spider.html)"
   220.181.125.106  www.zhe800.com  -  [23/May/2018:18:56:20  +0800]  "GET
/concern/11923041     HTTP/1.1"     200     162     "-"     "Sogou     web
spider/4.0(+http://www.sogou.com/docs/help/webmasters.htm#07)"
   24.125.7.34   www.zhe800.com   -   [23/May/2018:18:56:20   +0800]   "GET
/concern/11923041     HTTP/1.1"     200     162     "-"     "Sogou     web
spider/4.0(+http://www.sogou.com/docs/help/webmasters.htm#07)"
```

对这些客户端的 IP 地址使用 nslookup 进行检查，以 IP 地址 123.125.71.29 为例：

```
   # nslookup 123.125.71.29
   Server:      192.168.1.4
   Address: 192.168.1.4#53

   Non-authoritative answer:
   29.71.125.123.in-addr.arpaname = baiduspider-123-125-71-29.crawl.baidu.com.

   Authoritative answers can be found from:
   125.123.in-addr.arpa nameserver = ns2.bta.net.cn.
   125.123.in-addr.arpa nameserver = ns.bta.net.cn.
   ns.bta.net.cn    internet address = 202.96.0.133
   ns2.bta.net.cn   internet address = 202.106.196.28
```

对 IP 地址 220.181.125.106 使用 nslookup 进行检查：

```
   # nslookup 220.181.125.106
   Server:      192.168.1.4
   Address: 192.168.1.4#53

   Non-authoritative answer:
   106.125.181.220.in-addr.arpa name       =       sogouspider-220-181-125-
106.crawl.sogou.com.

   Authoritative answers can be found from:
   181.220.in-addr.arpa nameserver = idc-ns3.bjtelecom.net.
   181.220.in-addr.arpa nameserver = idc-ns1.bjtelecom.net.
   181.220.in-addr.arpa nameserver = idc-ns2.bjtelecom.net.
   idc-ns1.bjtelecom.netinternet address = 218.30.26.68
   idc-ns2.bjtelecom.netinternet address = 218.30.26.70
   idc-ns3.bjtelecom.netinternet address = 211.100.2.125
```

对 IP 地址 24.125.7.34 使用 nslookup 进行检查：

```
   # nslookup 24.125.7.34
   Server:      192.168.1.4
   Address: 192.168.1.4#53
```

```
Non-authoritative answer:
34.7.125.24.in-addr.arpa name = c-24-125-7-34.hsd1.ga.comcast.net.

Authoritative answers can be found from:
125.24.in-addr.arpa  nameserver = dns105.comcast.net.
125.24.in-addr.arpa  nameserver = dns102.comcast.net.
```

通过对上述 IP 地址的检查，可以看出 IP 地址 24.125.7.34 对应的不是搜狗的爬虫，其他的两个 IP 地址经过反向解析都找到了对应的爬虫公司，因此可以认定是搜索引擎爬虫。通过此方式可以识别出目前已知的各大正规搜索引擎的爬虫，那么剩下的爬虫行为都可以理解为不安全的恶意爬虫。

15.2 应对搜索引擎爬虫

先来谈谈各大搜索引擎爬虫的情况，合理应对它们可以让服务更加稳定和安全。

15.2.1 搜索引擎的 User-Agent

搜索引擎在爬取时会带有标识性的 User-Agent，一般可在搜索引擎官网的帮助中心查到，例如，想了解 360 搜索的 User-Agent 标识可访问 https://www.so.com/help/help_3_2.html。图 15-1 所示为 360 搜索关于 User-Agent 标识的说明。

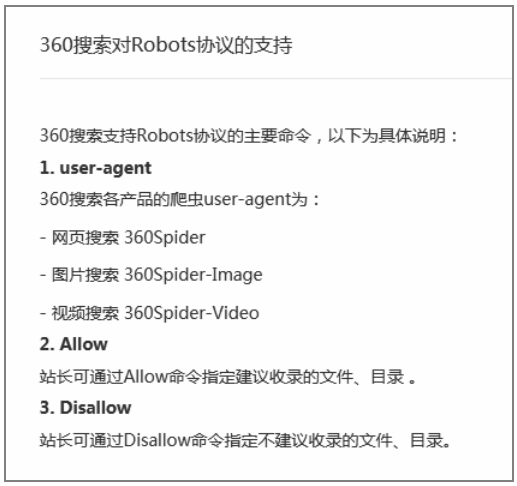


图 15-1　360 搜索关于 User-Agent 标识的说明

15.2.2　Robots 协议

Robots 协议的全称是网络爬虫排除标准（Robots Exclusion Protocol），网站通过 Robots 协议告诉搜索引擎哪些页面可以爬取，哪些页面不能爬取。

搜索引擎访问站点时会先去网站的根目录下查找文件 robots.txt，如果有这个文件就会根据文件上的配置进行访问。通过文件 robots.txt 就可以控制爬虫的爬取行为，图 15-2 所示为 robots.txt 文件的作用。

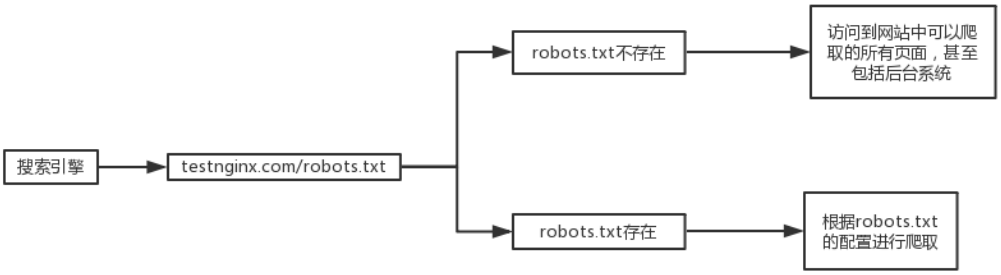


图 15-2　robots.txt 文件的作用

robots.txt 文件的配置说明见表 15-1。

表 15-1　robots 文件的配置说明

语　　法	说　　明
User-agent: *	允许所有搜索引擎爬取数据，如果要对某个搜索引擎进行限制，则需要单独配置
Disallow: /	禁止爬取全部的 URL，一般用于限制部分搜索引擎，如禁止某个搜索引擎的访问
Disallow: /login/	禁止爬取/login 下的目录
Allow: /abc/	允许访问/abc 下的目录
Allow: .html$	允许访问以“.html”为后缀的 URL

robots.txt 文件内容示例如图 15-3 所示。

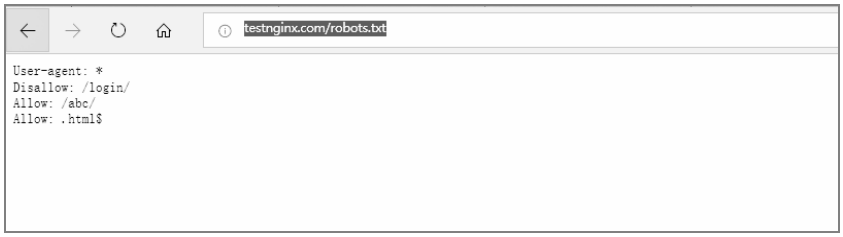


图 15-3　robots.txt 文件内容示例

如需关闭某个搜索引擎的爬取权限，单独进行配置即可，关闭 TestSpider 访问权限的配置

示例如图 15-4 所示。

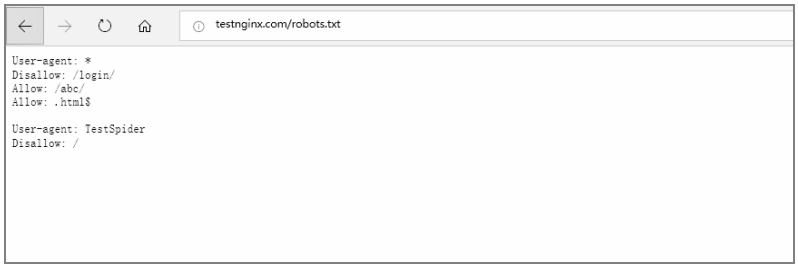


图 15-4　关闭 TestSpider 访问权限的配置示例

注意：robots.txt 文件一般是由开发人员和 SEO（Search Engine Optimizers，指专门从事搜索引擎优化的技术人员）制定出可以爬取的规则后，再上传给 robots.txt 的。

15.2.3　控制搜索引擎爬虫实战

读者已经知道如何区分搜索引擎、如何控制搜索引擎的访问了，但如果爬虫太多、请求太频繁，会不会导致服务器出现性能问题呢？试看图 15-5 所示的搜索引擎爬取网站的示例，会不会感觉“压力山大”？

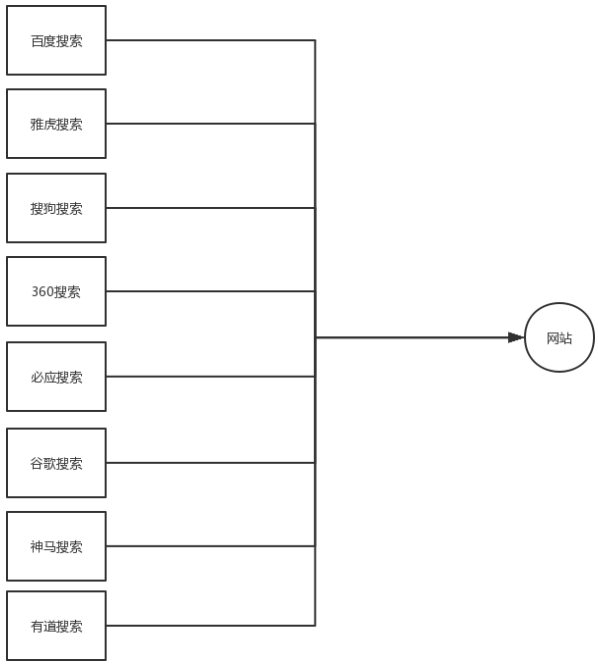


图 15-5　搜索引擎爬取网站的示例

其实，当网站的性能受到影响时，可以通过如下方式进行解决。

1．确认目前有多少家搜索引擎在爬取数据。通过 Nginx 的数据分析可以得到，也可以与 SEO 确认目前已知的搜索引擎公司。

2．与 SEO 确认必须要给哪些搜索引擎网站提供爬取权限，对于不需要提供爬取权限的搜索引擎可在 robots.txt 文件上禁止其访问。

3．禁止后观察几天，如果性能问题依然存在，可在各大搜索引擎官网进行爬虫降权限速的配置。

4．如果第 3 步效果仍然不明显，或不太愿意去各大搜索引擎官网配置限速，也可以为爬虫提供定制信息。

给爬虫提供定制信息的方案，属于业务范畴，不在本书所讲范围内，但是通过 Nginx 可以快速部署分流方案，将爬虫的请求发送到单独的服务器上，以减少对用户请求的干扰。以百度搜索和 360 搜索为例，将其爬虫的请求转发到单独的服务器上，代码如下：

```
# cat nginx.conf

user  webuser webuser;
worker_processes  1;

worker_rlimit_nofile 102400;

events {
    use epoll;
    worker_connections  102400;
}

http {
    include       mime.types;
    default_type  application/octet-stream;
    sendfile        on;
    keepalive_timeout  65;

    upstream  test_12 {
        server 127.0.0.1:81  weight=20  max_fails=300000 fail_timeout=5s;
        server 127.0.0.1:82  weight=20  max_fails=300000 fail_timeout=5s;
    }

    upstream spider_cache {
         server 127.0.0.1:8000;
    }
```

```
map $http_user_agent  $server {
    # 默认 user_agent 对应的结果
    default             test_12;
    # 判断 user_agent 是否包含 360 搜索的标识
   ~360Spider          spider_cache;
    # 判断 user_agent 是否包含百度搜索的标识
   ~BaiduSpider        spider_cache;
}
server {
    listen       80;
    server_name  testnginx.com;
    location  /  {
    # $server 就是 map 指令获取的变量
    proxy_pass http://$server;
    }
}
```

15.3 应对恶意爬虫

爬虫对服务的恶劣影响更多的是来自恶意爬虫，它们一方面会窃取网站信息，另一方面会影响服务的性能。那么该如何减少这类问题的发生呢？（根据以往的经验，很难彻底屏蔽所有的恶意爬虫，只能通过一些方案来减少它们的袭击。）

15.3.1 发现恶意爬虫

首要任务是找到这些恶意爬虫，下面是笔者根据多年经验整理出来的一些方法。

1．恶意爬虫喜欢伪造 User-Agent，假装自己是搜索引擎，通过 15.1 节所讲的方法可以找出这些恶意爬虫。

2．比较初级的恶意爬虫的 User-Agent 可能会带有 LWP.*Simple、BBBike、Python-urllib、^trasin、^curl、*libcurl、WWW.*Mechanize 等和编程语言相关的内容。

3．某些恶意爬虫会模拟用户真实的 User-Agent，从而逃过前两种检查方案。但它的访问量会很大，所以可以计算某段时间内某 IP 地址的访问次数，次数越多，则是恶意爬虫的可能性就越大。然后再分析此 IP 地址下对应的用户行为，恶意爬虫一般会带有如下特点。

- 访问的 URL 是持续性的顺序访问。
- 访问的 URL 种类不多，可能主要集中在某几个类型上。
- 访问时都没有带 Cookie，或者所有 Cookie 都是一样的。

- 访问的 Cookie 解析出来的用户是假的，或者 Cookie 是随机的。
- 访问的 IP 地址属于云厂商或机房的出口 IP 地址。

4．某些恶意爬虫会模拟用户的真实行为，访问的频率很低，缓慢地爬取网站信息，这样做是为了逃过第 3 种检查方案，对此可以做如下处理。

统计每天的访问次数，如果访问次数大于一个正常用户的访问量，则筛选出对应的 IP 地址，采取第 3 种检查方案验证其访问特点，确认是否为正常用户的操作。

5．如果经过第 3 种和第 4 种检查方案仍然无法判断是否为恶意爬虫，可以给该 IP 地址的请求发送一个特殊的 Cookie。如果是正常用户，下一次请求时就会带上这个 Cookie 来访问服务器；如果是恶意爬虫，则一般是带不了这个 Cookie 的。原因在于大部分爬虫不会启动浏览器来模拟访问，所以“种不上”Cookie。但恶意爬虫也可以对返回的 Cookie 进行获取并“种植”在下一次请求中，以混淆判断。

注意：如果 App 客户端有恶意爬虫，其判断方式会有变化，由于 4G 网络的特殊性，很多用户的 IP 地址都是一样的，因此不能单纯通过 IP 地址的访问次数来判断，一定要结合 URL 的访问特性、Cookie 和用户访问行为来判断。例如真实的用户会有进入个人中心，增、删购物车或支付等操作，爬虫是很难模拟这些的，所以也可以结合自身的业务特性进行判断。

15.3.2　抵御恶意爬虫之禁止访问

虽然不能发现全部恶意爬虫，但现在已经有了发现大部分恶意爬虫的能力，是时候开始抵御这些恶意爬虫了，下面是禁止恶意爬虫访问的代码：

```
    server {
        listen       80;
        server_name  testnginx.com;

        deny xxx.xxx.xxx.xxx;    # 将伪造成搜索引擎的 IP 地址配置到 deny，禁止其访问
        deny 124.124.124.123;    # 将伪造成搜索引擎的 IP 地址配置到 deny，禁止其访问

        if  ($http_user_agent  ~*  LWP.*Simple|BBBike|Python-urllib|^trasin|^
curl. *libcurl|WWW. *Mechanize) {
            return 403;          # 禁止不合法的 User-Agent 的访问
        }
        location  /  {
          proxy_pass http://test_12;
        }

    }
```

15.3.3 抵御恶意爬虫之验证码拦截

前面对已确认为恶意爬虫的请求进行了配置，但有些 IP 地址只是存在可疑性，尚不能认定为恶意爬虫，总不能对可疑目标也“一刀切”地禁止访问。这种情况其实可以采用验证码功能。将可疑的 IP 地址传入 Nginx 的 Lua 共享内存中，并通过 Nginx 进行拦截（Nginx 的验证码拦截功能详见第 14 章）。

15.4 小插曲——使用假数据迷惑恶意爬虫

本节分享一个在实战中对抗恶意爬虫的案例。

有一次，笔者和同事通过分析日志发现大量的 IP 地址爬取了本公司网站详情页的销量和评论的数据，初步怀疑是竞争对手在搞鬼。于是，我们将异常的 IP 地址在 Nginx 中全部配置为 deny，禁止其访问。可是第 2 天，恶意爬取的团队发现被封，又换了一批 IP 地址继续进行爬取，然后我们再次封掉这批新的 IP 地址，但对方很快又换了一批。

如果这样下去受苦受累的还是负责运维的同胞们，而且对手仍然可以通过不断更换 IP 地址的方式爬取到我们所有的信息，长期下去数据就被“爬”完了。最后我们想到一个办法，给了对手一个措手不及，具体操作如下。

首先，获取到对手的爬取规律，具体如下。

- 请求量很大，已经触发了限速规则，时不时会出现 503 错误，对方也知道这种情况，但只要还有不少请求返回 200，他们就认为服务是正常的，就会继续进行爬取。
- 同一批请求的 IP 地址非常多，并且 User-Agent 是伪造的。
- 对手有足够多的 IP 地址可以轮换使用。
- 爬取的内容基本是与销量、好评率、评论等有关的 JSON 格式的数据。

最后我们决定迷惑对手，给他们制作假数据，于是在 Nginx 上做了如下配置：

```
server {
    listen       80;
    server_name  testnginx.com;

    # /a.json 和 b.json 是爬虫要访问的接口
    location ~ ^(/a.json|/b.json) $ {
    # $remote_addr 配置的 IP 地址是恶意 IP 地址，如果 IP 地址过多，建议使用 Lua 的共
    # 享内存或存放在 Redis 上，或者 geo 模块也可以，这些方式本书都有介绍。if 语句是为了
    # 方便读者阅读。请注意如果 IP 地址是通过 CDN 或代理服务器传递的，则需要使用 realip
    # 来获取真实的 IP 地址
```

```
        if ($remote_addr ~ (xxx\.xxx\.xxx\.xx|xxx\.xxx\.xxx\.xxx)) {
        # 将请求跳转到 /make_false/的 location
            rewrite ^/(.*) /make_false/$1 last;
        }
        proxy_pass http://test_12;
    }

    location ~ ^/make_false/ {
        # 因为用了$request_uri，请求会使用原始的 URL，而不会新增/make_false/ 前缀
        proxy_pass http://err_data_servers$request_uri;
    }
}
```

其中 upstream 的 err_data_servers 服务里存放的是错误的销量数据，这样对方爬取的数据也就是错误的了，同时，服务的压力也小了很多。通过这种方式，我们“忽悠”了对方很长一段时间。当然，这并不是很好的选择，但和恶意爬虫打交道是“持久战”，就看谁能坚持到最后。直到后来全站接入验证码功能，我们才从此类问题中解脱出来。

15.5　小结

本章讲解了爬虫的应对方案，虽然不能完全杜绝恶意爬虫，但至少不会让恶意爬虫轻轻松松地获取到信息了。

第 16 章
性能分析和优化

Nginx 如果只是用来做反向代理的话，那么，它的优化方式简单且有效，例如增加 worker 进程、增加长连接、减少硬盘中临时文件的存储和优化内核等。但 Nginx 被当作开发利器后，代码复杂度也在逐步增加，不合理地使用 Nginx 也会造成性能问题。本章会介绍多个开源工具，利用它们可以查找 Nginx 的性能问题并进行优化处理。

注意：本章包含与 Ngx_Lua 有关的性能分析但不限于此，相关指令都可以在 Nginx 和 OpenResty 两个平台上进行测试。部分工具在 LuaJIT 2.0 和 LuaJIT 2.1 中会有区别，这些都会在讲解时进行说明。

16.1 性能分析场景搭建

性能分析一般会在测试环境下使用（如果线上服务有性能问题且在测试环境下无法复现时，则另当别论），这样可以确保在服务上线前做好性能优化。性能分析需要开启 Debug 模式，所以需要先搭建 Debug 环境，该环境需要依赖很多包，请读者按照步骤一步步来。

16.1.1 安装 SystemTap

SystemTap 是用来分析 Linux 系统性能的工具，通过它提供的接口可以开发出调试和分析性能的代码，本章介绍的性能分析方式都是依赖于 SystemTap 的。

首先需要确认系统版本，推荐 Linux 内核版本要大于 3.5，如果低于此版本，官方也会在其内核中提供 utrace 补丁，本测试安装在 CentOS 6.4 版本上，安装方式如下：

```
# uname -r
```

```
2.6.32
```

安装对应版本的内核包前需要先确认是否安装了 kernel-devel：

```
#  rpm -qa |grep   kernel-devel
kernel-devel-2.6.32-696.30.1.el6.x86_64
```

如果没有，则先安装此包：

```
# yum install kernel-devel
```

然后，提供 Debug 支持：

```
# wget -S http://debuginfo.centos.org/6/x86_64/kernel-debuginfo-common-x86_64-2.6.32-696.30.1.el6.x86_64.rpm
# wget -S http://debuginfo.centos.org/6/x86_64/kernel-debuginfo-2.6.32-696.30.1.el6.x86_64.rpm
# rpm -ivh kernel-debuginfo-2.6.32-696.30.1.el6.x86_64.rpm kernel-debuginfo-common-x86_64-2.6.32-696.30.1.el6.x86_64.rpm
```

安装 SystemTap：

```
# yum install systemtap
```

验证是否安装成功：

```
# stap -ve 'probe begin { log("Test Nginx Systemtap!")exit() }'
Pass 1: parsed user script and 117 library script(s) using 213788virt/41172res/3232shr/38628data kb, in 330usr/20sys/348real ms.
Pass 2: analyzed script: 1 probe(s), 2 function(s), 0 embed(s), 0 global(s) using 214580virt/42280res/3536shr/39420data kb, in 10usr/0sys/8real ms.
Pass 3: translated to C into "/tmp/stapldNGqo/stap_193cfbe6fbce06a86a9581c649f20084_960_src.c" using 214580virt/42668res/3892shr/39420data kb, in 0usr/0sys/0real ms.
Pass 4: compiled C into "stap_193cfbe6fbce06a86a9581c649f20084_960.ko" in 1040usr/210sys/1281real ms.
Pass 5: starting run.
Test Nginx Systemtap!
Pass 5: run completed in 0usr/10sys/345real ms.
```

16.1.2　LuaJIT 的 Debug 模式

需要在安装 LuaJIT 时启动 Debug 模式，即 make CCDEBUG=-g，方法如下：

```
# wget http://luajit.org/download/LuaJIT-2.1.0-beta3.tar.gz
# tar -zxvf LuaJIT-2.1.0-beta3.tar.gz
# cd  LuaJIT-2.1.0-beta3
# make CCDEBUG=-g
```

```
# make install
# export LUAJIT_LIB=/usr/local/lib
# export LUAJIT_INC=/usr/local/include/luajit-2.1
```

16.1.3 开启 PCRE 的 Debug 模式

有时需要对 Nginx 正则表达式的使用情况进行分析，确保服务在正则操作方面的性能。

如果 Nginx 在静态编译中使用了--with-pcre=的方式，请重新编译并开启 Debug 模式：

```
# cd nginx-1.12.2
# ./configure --with-pcre=/path/to/my/pcre-8.39 \
    --with-pcre-jit  --with-pcre-opt=-g \
```

如果 PCRE 的包是动态链接的，则直接安装 rpm 即可：

```
#   wget   http://debuginfo.centos.org/6/x86_64/pcre-debuginfo-7.8-7.el6.
x86_64.rpm
# rpm -ivh pcre-debuginfo-7.8-7.el6.x86_64.rpm
```

需要确保如下所示的 3 个包都存在：

```
# rpm -qa |grep pcre
pcre-debuginfo-7.8-7.el6.x86_64
pcre-7.8-7.el6.x86_64
pcre-devel-7.8-7.el6.x86_64
```

16.1.4 分析工具下载

基础环境搭建好后，就可以使用 openresty-systemtap-toolkit 和 stapxx 这两个工具进行性能分析了，这两个工具都有各自的使用场景，其中 stapxx 是对 openresty-systemtap-toolkit 的简单扩展，更侧重于进行与 Ngx_Lua 有关的性能分析。下面会混用这两个工具，它们的下载方式如下：

```
# git clone https://github.com/openresty/openresty-systemtap-toolkit
# git clone https://github.com/openresty/stapxx.git
```

然后，下载 FlameGraph（FlameGraph 是将上面两个工具采集到的性能数据生成火焰图的工具）：

```
# git clone https://github.com/brendangregg/FlameGraph.git
```

注意：Nginx 编译时需要开启 Debug 模式，即编译时加上--with-debug，并且 Linux 系统要求 Perl 至少是 5.6.1 以上的版本（默认情况下已经安装）。

16.1.5　找出不支持 Debug 模式的 lib 库

如果 lib 库不支持 Debug 模式，而需要测试的功能又和这个 lib 库有关，则测试时会报错。可以先使用命令 check-debug-info -p pid 找出不支持 Debug 模式的 lib 库。其中 pid 可以是 Nginx 的 worker 进程的 PID，也可以是其他 Linux 下的进程（意味着此工具不只可以用来检测 Nginx 进程）PID。该命令是 16.1.4 中提到的 openresty-systemtap-toolkit 目录中的命令，用法如下：

```
# cd openresty-systemtap-toolkit
# ./check-debug-info -p  30396
File /lib64/ld-2.12.so has no debug info embedded.
File /lib64/libc-2.12.so has no debug info embedded.
File /lib64/libcrypt-2.12.so has no debug info embedded.
File /lib64/libdl-2.12.so has no debug info embedded.
File /lib64/libfreebl3.so has no debug info embedded.
File /lib64/libm-2.12.so has no debug info embedded.
File /lib64/libnss_files-2.12.so has no debug info embedded.
File /lib64/libpthread-2.12.so has no debug info embedded.
File /lib64/libresolv-2.12.so has no debug info embedded.
```

16.2　流量复制

性能分析的环境已经搭建完成，现在需要有流量进入才可以验证这些代码的使用情况。可以用 goreplay 或 Nginx 的镜像来模拟真实的请求和并发情况。它们在 13.5 节有介绍，这里就不重复说明了。

如果是新服务，且 URL 也是新的，可以理解为线上还没有流量可复制，那么可以用压测工具，如 AB（Apache Benchmark， Apache 自带的压力测试工具）、Webbench 等。

16.3　各项指标分析和优化建议

现在性能测试环境全部就位，可以进行性能分析了。

注意：*所有的分析操作都是在 Nginx 的 worker 进程 PID 上执行的。*

16.3.1　连接池使用状态分析

前面介绍过 lua-resty-redis 和 lua-resty-mysql，它们都使用过连接池的配置，具体如下：

```
local ok, err = db:set_keepalive(10000, 100)
if not ok then
```

```
        ngx.say("failed to set keepalive: ", err)
        return
    end
```

连接池可以极大地减少 Linux 系统的 timewait 及 Nginx 和数据库建立连接的开销，但如何才知道连接池的数量配置是否满足需求呢？可用如下方式：

```
# cd openresty-systemtap-toolkit
# ./ngx-lua-conn-pools -p  30396 --luajit20
Tracing 30396 (/usr/local/nginx/sbin/nginx) for LuaJIT 2.0···

pool "172.16.1.51:6301"
    out-of-pool reused connections: 1
    in-pool connections: 2
        reused times (max/avg/min): 29/18/7
    pool capacity: 100

pool "172.16.13.171:6398"
    out-of-pool reused connections: 0
    in-pool connections: 1
        reused times (max/avg/min): 0/0/0
    pool capacity: 100

pool "172.16.1.55:8186"
    out-of-pool reused connections: 0
    in-pool connections: 1
        reused times (max/avg/min): 377/377/377
    pool capacity: 30

pool "172.16.1.7:5689"
    out-of-pool reused connections: 0
    in-pool connections: 1
        reused times (max/avg/min): 1/1/1
    pool capacity: 100

For total 4 connection pool(s) found.
122 microseconds elapsed in the probe handler.
```

指令：ngx-lua-conn-pools

语法：ngx-lua-conn-pools -p $pid (--luajit20 或--lua51)

含义：获取指定 worker 进程中的 Ngx_Lua 的连接池状态。如果 Nginx 编译时使用的是 Lua 5.1，就用--lua51；如果是 LuaJIT 2.0，就用--luajit20，目前 LuaJIT2.1 也可以用--luajit20。

连接池信息说明见表 16-1。

表 16-1　连接池信息说明

结　　果	说　　明
pool "172.16.1.51:6301"	连接池所连接的服务，如 Redis、MySQL
out-of-pool reused connections: 1	外部连接池重用数量
in-pool connections: 2	连接池内的连接数量
reused times (max/avg/min): 29/18/7	重用次数、最大值、平均值、最小值
pool capacity: 100	连接池容量，即文件内配置连接池的数量

通过此命令可以获取连接池的使用数量和配置数量，可以用来在压力测试中观察连接池的使用情况，以确认是否需要调整连接池的配置。

16.3.2　找出读/写频繁的文件

找出被读次数较多的文件的名称，默认输出排名前 10 的文件名：

```
# cd openresty-systemtap-toolkit
# ./accessed-files -p 48070 -r
Tracing 48070 (/usr/local/nginx/sbin/nginx)···
Hit Ctrl-C to end.
^C
=== Top 10 file reads ===
#1: 1 times, 2033 bytes reads in file middle_page.lua.
#2: 1 times, 2374 bytes reads in file rand_str.lua.
```

找出被写入次数较多的文件的名称，默认输出排名前 10 的文件名：

```
# ./accessed-files -p 48070 -w
Tracing 48070 (/usr/local/nginx/sbin/nginx)···
Hit Ctrl-C to end.
^C
=== Top 10 file writes ===
#1: 1976 times, 3353103 bytes writes in file access.log-2018-06-28-20-
14-01.log.
#2: 1975 times, 841837 bytes writes in file wireless.access.log.
#3: 1360 times, 2206474 bytes writes in file  access.log.
#5: 17 times, 5598 bytes writes in file error.log.
```

16.3.3　执行阶段耗时分析

客户端请求会在 Nginx 的多个执行阶段被处理，获取每个执行阶段的执行效率就可以定位优化范围，从而提升响应速度了，代码如下：

```
# cd stapxx
```

```
# export PATH=$PWD:$PATH
# ./samples/ngx-single-req-latency.sxx -x  18993
Start tracing process 18993 (/usr/local/nginx/sbin/nginx)···

[1530153801023919]  pid:18993  GET  /zhe800_n_api/search/hot_words?new_
user=1&user_id=0&user_type=0&callback=hot_words
      total: 1235us, accept() ~ header-read: 65us, rewrite: 23us, pre-
access: 23us, access: 26us, content: 985us
      upstream: connect=270us, time-to-first-byte=496us, read=0us
```

使用./samples/ngx-single-req-latency.sxx -x $pid，获取 worker 进程的单个请求在各个阶段的消耗时间（上述代码中时间的单位是微秒，即 μs，）。默认只输出该指令获取到的第一条请求的消耗时间，所以如果要测试某个请求的阶段耗时，可以使用 goreplay 复制流量进行过滤，只需发送指定的 URL 来验证即可。

16.3.4 HTTP 连接数和文件打开数分析

通过分析 worker 进程的连接数量和文件打开数量，可以知道系统配置的资源是否够用，避免出现文件数或连接数不够用的情况。

```
# cd stapxx
# export PATH=$PWD:$PATH
# ./samples/ngx-count-conns.sxx -x 18993
Start tracing 18993 (/usr/local/nginx/sbin/nginx)···

====== CONNECTIONS ======
Max connections: 102400      # 设置最大连接数
Free connections: 101854     # 还可以接受的连接数
Used connections: 546        # 已经使用的连接数

====== FILES ======
Max files: 102400            # 设置最大文件打开数
Open normal files: 20        # 当前打开的文件数量
```

16.3.5 找出 CPU“偷窃者”

通过下面的命令可以获取抢占指定 PID 进程的 CPU 频率，该命令也可以用来分析其他非 Nginx 的进程。

```
# cd stapxx
# export PATH=$PWD:$PATH
# ./samples/cpu-robbers.sxx -x  30396
Start tracing process 30396 (/usr/local/nginx/sbin/nginx)···
Hit Ctrl-C to end.
```

```
^C
#1 consul: 38% (5 samples)
#2 events/0: 30% (4 samples)
#3 kblockd/0: 15% (2 samples)
#4 watchdog/0: 7% (1 samples)
#5 zabbix_agentd: 7% (1 samples)
```

16.3.6　正则表达式耗时分析

在 Nginx 中会用到大量的正则表达式，特别是 Ngx_Lua 会设置很多动态路由，如果正则表达式耗时过长对请求响应速度和 CPU 都会造成影响。使用下面的命令可以捕获 Nginx 中耗时最长的正则操作：

```
# cd stapxx
# export PATH=$PWD:$PATH
# ./samples/ngx-pcre-top.sxx --skip-badvars -x  18999
Found exact match for libluajit: /usr/local/lib/libluajit-5.1.so.2.1.0
Found exact match for libpcre: /lib64/libpcre.so.0.0.1
Start tracing 18999 (/usr/local/nginx/sbin/nginx)
Hit Ctrl-C to end.
^C
Top N regexes with longest total running time:
1. pattern //*/: 7921us (total data size: 17947)
2. pattern //address/[a-zA-Z]+[0-9]+/view/edit$/: 6801us (total data
size: 14684)
3. pattern //address/[a-zA-Z]+[0-9]+/view/default$/: 6555us (total data
size: 14088)
4. pattern //address/[a-zA-Z]+[0-9]+/queryAddressById$/: 6253us (total
data size: 14684)
5. pattern //address/[a-zA-Z]+[0-9]+/view/add$/: 6161us (total data
size: 14684)
6. pattern //address/[a-zA-Z]+[0-9]+/view/query$/: 5834us (total data
size: 14684)
7. pattern //address/[a-zA-Z]+[0-9]+/view/delete$/: 5634us (total data
size: 14088)
8. pattern //address/[a-zA-Z]+[0-9]+/get_default$/: 5475us (total data
size: 14088)
9. pattern /0/: 3521us (total data size: 7258)
10. pattern //orders/*/: 3088us (total data size: 7530)
```

使用--arg utime=1 可以提高获取到的耗时时间的准确性，它可以在 Ngx_Lua、lua-resty-core 的 API 中使用，但某些平台可能不支持本参数。如下是加入--arg utime 参数后的输出结果：

```
# cd stapxx
# export PATH=$PWD:$PATH
```

```
   # ./samples/ngx-pcre-top.sxx --skip-badvars -x  18999 --arg utime=1
   ^[[AFound   exact   match   for   libluajit:   /usr/local/lib/libluajit-
5.1.so.2.1.0
   Found exact match for libpcre: /lib64/libpcre.so.0.0.1
   Start tracing 18999 (/usr/local/nginx/sbin/nginx)
   Hit Ctrl-C to end.
   ^C
   Top N regexes with longest total running time:
   1. pattern //*/: 3000us (total data size: 20418)
   2. pattern //address/[a-zA-Z]+[0-9]+/view/add$/: 2000us (total data
size: 11302)
   3.  pattern  //address/[a-zA-Z]+[0-9]+/view/query$/:  2000us  (total  data
size: 11302)
   4.  pattern  //address/[a-zA-Z]+[0-9]+/view/manage$/:  2000us  (total  data
size: 5428)
   5.  pattern  //mz/catelist/[a-zA-Z]+[0-9]+$/:  1000us  (total  data  size:
1188)
   6. pattern //mz/baoyou/[a-zA-Z]+[0-9]+$/: 1000us (total data size: 594)
   7.  pattern  //m/list/baoyou/[a-zA-Z]+[0-9]+$/:  1000us  (total  data  size:
594)
   8. pattern //m/detail/*/: 1000us (total data size: 718)
   9. pattern //cn/z_key/*/: 1000us (total data size: 997)
   10.  pattern  //nnc/orders/[a-zA-Z]{14}.[a-zA-Z]{4}$/:  1000us  (total  data
size: 1445)
```

可以使用下面的命令获取所有正则表达式消耗的时间分布：

```
   # cd stapxx
   # export PATH=$PWD:$PATH
   # ./samples/ngx-pcre-dist.sxx -x  18999
   Found exact match for libpcre: /lib64/libpcre.so.0.0.1
   Start tracing 18999 (/usr/local/nginx/sbin/nginx)
   Hit Ctrl-C to end.
   ^C
   Logarithmic  histogram  for  data  length  distribution  (byte)  for  196882
samples:
    (min/avg/max: 0/22/4749)
   value |-------------------------------------------------- count
       0 |                                                    1255
       1 |                                                    1272
       2 |@                                                   3336
       4 |                                                    1481
       8 |@@@@@@@@@@@@@                                      50690
      16 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@                  113128
      32 |@@@@@@                                              23912
      64 |                                                    1341
```

```
     128 |                                                    419
     256 |                                                     30
     512 |                                                     12
    1024 |                                                      4
    2048 |                                                      0
    4096 |                                                      2
    8192 |                                                      0
   16384 |                                                      0
```

从上面的执行结果可知，大多数请求的耗时时间是 16μs，耗时 16μs 的请求有 113128 个。

./samples/ngx-pcre-dist.sxx -x $pid 也支持使用--arg utime=1，用来提高时间的准确性，如下是加入--arg utime 参数后的输出结果：

```
   # cd stapxx
   # export PATH=$PWD:$PATH
   # ./samples/ngx-pcre-dist.sxx -x  18999 --arg utime=1
   Found exact match for libpcre: /lib64/libpcre.so.0.0.1
   Start tracing 18999 (/usr/local/nginx/sbin/nginx)
   Hit Ctrl-C to end.
   ^C
   Logarithmic  histogram  for  data  length  distribution  (byte)  for  46247
samples:
   (min/avg/max: 0/23/805)
   value |-------------------------------------------------- count
       0 |                                                    215
       1 |                                                    214
       2 |@                                                   715
       4 |                                                    349
       8|@@@@@@@@@@@@@@@@@@                                 10117
      16 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@                 28346
      32 |@@@@@@@@@@                                          5973
      64 |                                                    247
     128 |                                                     66
     256 |                                                      3
     512 |                                                      2
    1024 |                                                      0
    2048 |                                                      0
```

16.3.7　找出消耗 CPU 资源较多的指令

如果发现 Nginx 在 CPU 资源使用上消耗过多，可以利用 lua-stacks.sxx 进行分析，它可以获取在 Nginx 的 worker 进程中各项指令占用 CPU 资源的比例。

如果是 LuaJIT 2.1，则使用的指令如下：

```
# cd stapxx
# export PATH=$PWD:$PATH
# ./samples/lj-lua-stacks.sxx --arg time=10 --skip-badvars -x 36984
  ./samples/lj-lua-stacks.sxx --arg time=10 --skip-badvars -x  48070
```

如果是 LuaJIT2.0 或 Lua5.1，则用下面的指令：

```
# cd openresty-systemtap-toolkit
# ./ngx-sample-lua-bt -p 9768 --luajit20 -t 10   # 如果是 Lua5.1, 将--
luajit20 换成--lua51 即可
```

这两种方式得到的输出结果是一样的，如下所示：

```
Found exact match for libluajit: /usr/local/lib/libluajit-5.1.so.2.1.0
WARNING: Start tracing 48070 (/usr/local/nginx/sbin/nginx)
WARNING: Please wait for 10 seconds···
WARNING: Time's up. Quitting now···
match
builtin#88
@/usr/local/nginx/conf/lua/log_jk/utils/utils.lua:4
@/usr/local/nginx/conf/lua/log_jk/utils/utils.lua:9
@/usr/local/nginx/conf/lua/log_jk/utils/utils.lua:29
@/usr/local/nginx/conf/lua/log_jk/log_to_influxdb.lua:1
    131
match
builtin#84
@/usr/local/nginx/conf/lua/log_jk/utils/utils.lua:9
@/usr/local/nginx/conf/lua/log_jk/utils/utils.lua:29
@/usr/local/nginx/conf/lua/log_jk/log_to_influxdb.lua:1
    114
compile_regex
C:ngx_http_lua_ngx_re_find
@/usr/local/nginx/conf/lua/log_jk/utils/utils.lua:29
@/usr/local/nginx/conf/lua/log_jk/log_to_influxdb.lua:1
    75
match
C:ngx_http_lua_ngx_exit
    26
lj_str_new
C:ngx_http_lua_var_get
@/usr/local/nginx/conf/lua/log_jk/log_to_influxdb.lua:1
    21
max_expand
builtin#88
@/usr/local/nginx/conf/lua/log_jk/utils/utils.lua:4
```

```
    @/usr/local/nginx/conf/lua/log_jk/utils/utils.lua:9
    @/usr/local/nginx/conf/lua/log_jk/utils/utils.lua:29
    @/usr/local/nginx/conf/lua/log_jk/log_to_influxdb.lua:1
```

很显然这种输出方式不便于观察 CPU 资源的消耗情况，这就需要用到 FlameGraph 了。

16.3.8　利用火焰图展示和分析数据

FlameGraph 是将采集到的性能数据生成火焰图的工具，可以让数据表达显得更直观，其使用方式如下。

首先，将 16.3.7 节中输出的内容存放到 tmp 目录下的 a.bt 文件中：

```
    # cd stapxx
    # export PATH=$PWD:$PATH
    #  ./samples/lj-lua-stacks.sxx  --arg  time=10  --skip-badvars  -x  36984
>/tmp/a.bt
```

然后，使用 openresty-systemtap-toolkit 中的 fix-lua-bt 获取 a.bt 中的 Lua 代码，并将其存放到 a_new.bt 文件中：

```
    # cd openresty-systemtap-toolkit
    # ./fix-lua-bt /tmp/a.bt > /tmp/a_new.bt
```

再使用 FlameGraph 根据 a_new.cbt 文件生成火焰图文件 a.svg：

```
    # cd FlameGraph
    # ./stackcollapse-stap.pl /tmp/a_new.bt > /tmp/a_new.cbt
    # ./flamegraph.pl /tmp/a_new.cbt > /tmp/a.svg
```

复制 a.svg 文件，并粘贴到浏览器中打开，会看到如图 16-1 所示的 worker 进程的火焰图。

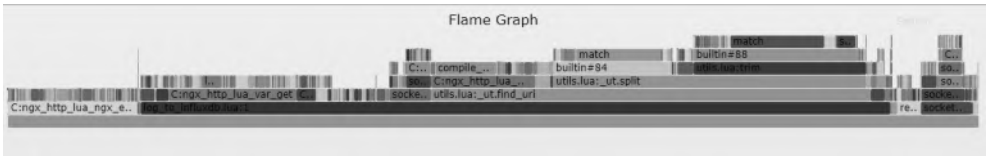


图 16-1　worker 进程的火焰图

对 a.svg 分析如下。

1．对于 16-2 所示的火焰图，图中垂直的数轴即 *y* 轴，表示调用栈的深度。顶部是正在执行的函数，下方都是它的父函数，在 *y* 轴上的高度越高，表面调用的层次越深。

图中水平的数轴即 *x* 轴，表示请求占用 CPU 资源的情况，在 *x* 轴上的宽度越宽，则表示请求的次数越多，占用耗时也就越长。

如果最上面是“平顶”的样子，则可能存在性能问题，正常的火焰图呈现的效果应该有很多尖刺，而不应该是“平顶”。

2．通过浏览器打开后，单击图中所示的 CPU 资源占比较高的位置，会看到具体是哪个函数执行的操作。

如图 16-2 所示为 CPU 资源占用率较高的火焰图，在火焰图的顶端，可以看到这个函数最后执行的命令是 match，match 采样 1144 次，CPU 资源占比 8.75%，是正则匹配操作。

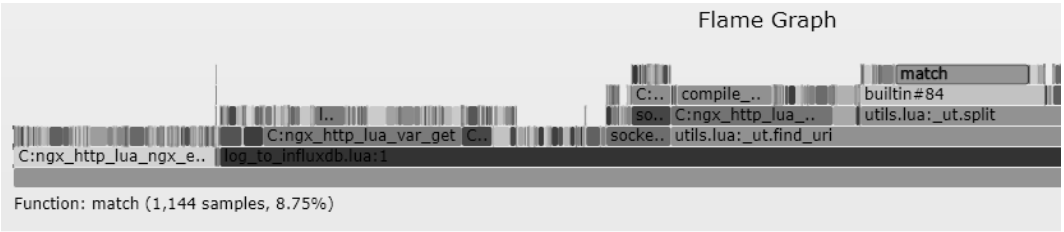


图 16-2　CPU 资源占用率较高的火焰图

3．该火焰图“平顶”过多，存在优化空间，根据图中定位的函数进行优化即可。

图 16-3 是 OpenResty 官网提供的正常火焰图模型。

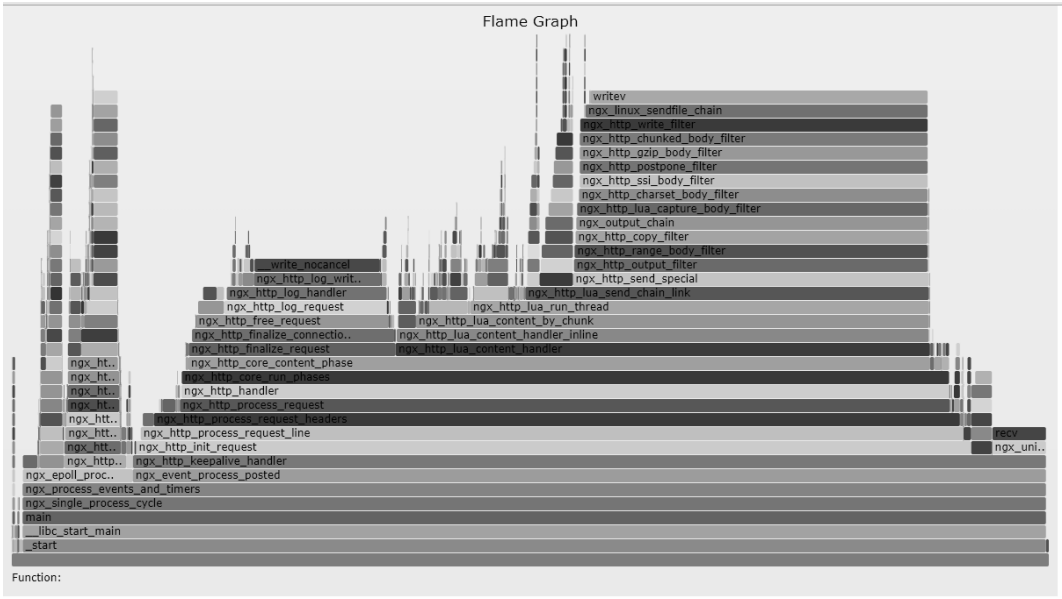


图 16-3　OpenResty 官网提供的正常火焰图模型

16.4　检查全局变量

在前面的章节中曾多次提到，在 Ngx_Lua 中要避免使用全局变量，因为它会带来很多意想不到的麻烦，使用检查工具 luacheck 可以检查变量是全局变量还是局部变量。

首先，安装检查工具 luacheck，代码如下：

```
# yum install luarocks
# luarocks install luacheck --deps-mode=none
```

--deps-mode=none 是可选参数，如果在安装过程中报错，则可以加上此参数再试试。

如果需要一次性检查目录下所有 Lua 脚本的变量，请安装 LuaFileSystem，luacheck 依赖于它，安装方式如下：

```
# luarocks install luafilesystem
```

一般情况下 Ngx_Lua 的代码都是基于 LuaJIT 2.0/2.1 开发的，那么检查时需要加入参数--std Ngx_Lua；如果是基于 Lua5.1 开发的，则需要使用--std lua51。--std 的作用是设置指定格式的全局变量的环境。

执行如下检查：

```
# luacheck  --std ngx_lua  /tmp/ab_version.lua

Checking /tmp/ab_version.lua                      2 warnings

    /tmp/ab_version.lua:1:7:  value  assigned  to  variable  version  is
unused
    /tmp/ab_version.lua:2:1: setting non-standard global variable xx

Total: 2 warnings / 0 errors in 1 file, couldn't check 1 file
```

根据检查结果发现了变量 version 和 xx 的问题，那么，现在去排查问题出现的原因吧。

16.5　小结

本章讲解的只是 openresty-systemtap-toolkit 和 stapxx 的部分命令，它们还提供了更多的命令来完善性能分析，如分析 off-cpu、内存泄漏、请求的队列等。有兴趣的读者可以访问相关 GitHub 的 Wiki。

第 17 章
值得拥有的 OpenResty

下面会介绍一些玩转 Nginx 的利器，这些利器会丰富我们的开发环境。

在前面的章节中，曾提到过 OpenResty，它自带丰富的组件，与 Nginx 相比，OpenResty 在开发中更灵活、更简便、也更稳定。

注意：本章的模块都运行在 OpenResty 下，使用 Nginx 的读者须手动安装这些模块，具体安装方式可以参考该模块的官网 Wiki。部分组件对 Nginx 的版本是有要求的，但 OpenResty 因为自带这些组件所以不需要考虑版本兼容的问题。

下面就来介绍一下 OpenResty 到底默认支持哪些组件。

首先，安装 OpenResty，方式如下：

```
# wget https://openresty.org/download/openresty-1.13.6.2.tar.gz
# tar -zxvf openresty-1.13.6.2.tar.gz
# cd openresty-1.13.6.2
# ./configure
# make
# sudo make install
```

安装完成后，prefix 的默认路径是/usr/local/openresty/，可以使用参数-V 查看 OpenResty 的默认编译内容，如下所示：

```
# /usr/local/openresty/nginx/sbin/nginx  -V
nginx version: openresty/1.13.6.2
built by gcc 4.8.5 20150623 (Red Hat 4.8.5-28) (GCC)
built with OpenSSL 1.0.2k-fips  26 Jan 2017
TLS SNI support enabled
configure arguments:
```

```
   --prefix=/usr/local/openresty/nginx
   --with-cc-opt=-O2
   --add-module=../ngx_devel_kit-0.3.0
   --add-module=../echo-nginx-module-0.61
   --add-module=../xss-nginx-module-0.06 --add-module=../ngx_coolkit-0.2rc3
   --add-module=../set-misc-nginx-module-0.32
   --add-module=../form-input-nginx-module-0.12
   --add-module=../encrypted-session-nginx-module-0.08
   --add-module=../srcache-nginx-module-0.31
   --add-module=../ngx_lua-0.10.13 --add-module=../ngx_lua_upstream-0.07
   --add-module=../headers-more-nginx-module-0.33
   --add-module=../array-var-nginx-module-0.05
   --add-module=../memc-nginx-module-0.19
   --add-module=../redis2-nginx-module-0.15
   --add-module=../redis-nginx-module-0.3.7
   --add-module=../rds-json-nginx-module-0.15
   --add-module=../rds-csv-nginx-module-0.09
   --add-module=../ngx_stream_lua-0.0.5
   --with-ld-opt=-Wl,-rpath,/usr/local/openresty/luajit/lib
   --with-stream --with-stream_ssl_module --with-http_ssl_module
```

可以看出 OpenResty 安装了很多组件，其中有些组件在前面的章节中已经介绍过。

17.1　OPM

本书所讲的大部分内容是基于 Nginx 环境的，且安装 Ngx_Lua 模块时都采用拷贝的方式，很显然这种方式并不“优雅”。如果是 OpenResty，那么它已经实现了利用 OPM（OpenResty Package Manager）来管理安装包的功能，可以很方便地安装各种模块，前提是该模块已经上传到了 OPM 平台上。图 17-1 所示为 OPM 的官方页面。

可以通过 http://opm.openresty.org/查看目前的模块列表和更新状态。下面介绍一下 OPM 的基本指令。

1．搜索想要安装的模块的包，以一次性搜索 2 个包为例，代码如下：

```
   # /usr/local/openresty/bin/opm  search lua-resty-http
   pintsized/lua-resty-http       Lua   HTTP   client   cosocket   driver   for
OpenResty/ngx_lua
   agentzh/lua-resty-http         Lua   HTTP   client   cosocket   driver   for
OpenResty/ngx_lua
   # /usr/local/openresty/bin/opm  search lua-resty-url
   3scale/lua-resty-url
```

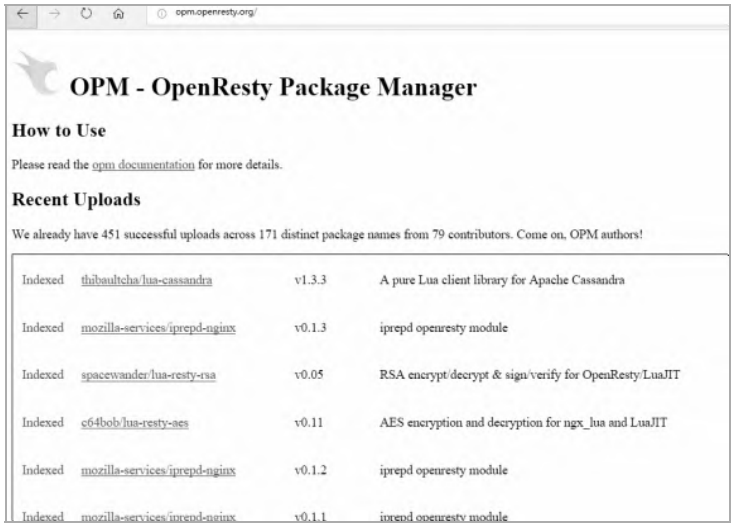


图 17-1　OPM 的官方页面

2．安装指定模块的包，代码如下：

```
    # /usr/local/openresty/bin/opm get pintsized/lua-resty-http
    * Fetching pintsized/lua-resty-http
      Downloading    https://opm.openresty.org/api/pkg/tarball/pintsized/lua-
resty-http-0.12.opm.tar.gz
      % Total     % Received % Xferd  Average Speed    Time     Time      Time
Current
      Dload  Upload   Total   Spent    Left  Speed
    100 19953  100 19953    0     0   134k      0 --:--:-- --:--:-- --:--:--
134k
    Package  pintsized/lua-resty-http  0.12  installed  successfully  under
/usr/local/openresty/site/
```

3．更新全部已经安装的模块的包，代码如下：

```
    # /usr/local/openresty/bin/opm update
    * Fetching pintsized/lua-resty-http > 0.12
    Package pintsized/lua-resty-http 0.12 is already the latest version.
    * Fetching 3scale/lua-resty-url > 0.3.3
    Package 3scale/lua-resty-url 0.3.3 is already the latest version.
```

4．更新指定模块的安装包，代码如下：

```
    # /usr/local/openresty/bin/opm upgrade  pintsized/lua-resty-http
    * Fetching pintsized/lua-resty-http > 0.12
    Package pintsized/lua-resty-http 0.12 is already the latest version.
```

5．移除已安装模块的包，代码如下：

```
#  /usr/local/openresty/bin/opm remove pintsized/lua-resty-http
Package pintsized/lua-resty-http 0.12 removed successfully.
```

更多指令请参考 OPM 的 Wiki，网址为 https://github.com/openresty/opm。

17.2　使用 DNS 提升访问效率

在实际开发中经常会有调用 API 的操作，它包含内部系统和公网的 HTTP 服务，通常情况下 API 是以域名的方式提供的，Nginx 在解析域名时为了提升效率会使用 resolver 指令来缓存解析到的 IP 地址，但 resolver 指令的功能较少，且需要在 Nginx 的配置中写“死”，用起来不够灵活，所以这里推荐使用 lua-resty-dns 模块。

注意：内网使用域名的好处在于，DNS 可以方便地切换访问后端服务的 IP 地址，减少应用服务器的耦合配置。

下面简单地介绍一下 lua-resty-dns 的使用方式和场景，示例代码如下：

```
content_by_lua_block {
            -- OpenResty 自带的模块
            local resolver = require "resty.dns.resolver"
            local r, err = resolver:new{
                nameservers = {"192.168.1.2"},
                retrans = 3,
                timeout = 2000,
            }
            if not r then
                ngx.say("failed to instantiate the resolver: ", err)
                return
            end
            -- 获取 test1.zhe800.com 的 A 记录的地址
            local  answers,  err,  tries  =  r:query("test1.zhe800.com",
{qtype = r.TYPE_A})
            if not answers then
                ngx.say("failed to query the DNS server: ", err)
                ngx.say("retry  historie:\n   ",  table.concat(tries,  "\n
"))
                return
            end
            if answers.errcode then
```

```
                ngx.say("server returned error code: ", answers.errcode,
                        ": ", answers.errstr)
            end
            -- 获取的数据是 table 类型的，输出 DNS 解析到的结果
            for i, ans in ipairs(answers) do
                ngx.say(ans.name, " ", ans.address or ans.cname,
                        " type:", ans.type, " class:", ans.class,
                        " ttl:", ans.ttl)
            end
        }
```

执行结果如下（会显示有多个 A 记录）：

```
# curl  http://testnginx.com/test
test1.zhe800.com 192.168.1.12 type:1 class:1 ttl:60
test1.zhe800.com 192.168.1.11 type:1 class:1 ttl:60
test1.zhe800.com 192.168.1.10 type:1 class:1 ttl:60
```

DNS 解析有一个让人头疼的问题，就是如果每次请求都需要解析，反复的 DNS 解析会影响请求的响应速度，一般情况下这个影响不大，但如果能够不受影响当然更好了；并且如果 DNS 服务崩溃的话，又该如何保证请求能够正常访问呢？这都可以使用如下方案解决。

- lua-resty-dns 支持配置多个 DNS 地址，因此可以配置 DNS 集群来降低单点故障的发生概率。
- 使用 ngx.timer.every 创建定时任务去解析 DNS，将解析到的数据缓存到 Ngx_Lua 的共享内存中，并设置为永不过期。
- 如果 DNS 服务器全部发生异常，定时更新缓存的操作不会被执行，而会继续使用旧的解析地址。
- Ngx_Lua 使用 DNS 解析的地址时，会去共享内存中读取地址，如果地址有多个，可以设置成随机读取。

注意：随机读取，可以使用生成随机数的方式，具体可参考 https://github.com/bungle/lua-resty-random。

17.3 TCP 和 UDP 服务

Nginx 从 1.9 版本开始，新增了支持 TCP/UDP 的 ngx_stream_core_module， Ngx_Lua 中的 ngx_stream_lua_module 也支持 TCP/UDP。使用 ngx_stream_core_module 的示例如下：

```
# stream 用来配置 TCP/UDP 服务
stream {
    server {
        listen 1111;

        # 使用 ngx.socke.* 来完成查询 memcached 的操作
        content_by_lua_block {
            local sock = ngx.socket.tcp()
            sock:settimeout(1000)
            local ok, err = sock:connect("127.0.0.1", 11211)
            if not ok then
               ngx.say("failed to connect: ", err)
               return
            end
            -- 读取 memcached 中 testnginxlua 的值
            local bytes, err = sock:send("get testnginxlua\r\n")
            if not bytes then
                ngx.say("failed to send query: ", err)
                return
            end
            local line, err = sock:receive()
            if not line then
                ngx.say("failed to receive a line: ", err)
                return
            end
            ngx.say("result: ", line)

        }
    }
}
```

使用 telnet 命令请求配置好 TCP 服务的 Nginx，执行结果如下：

```
# telnet  127.0.0.1 1111
Trying 127.0.0.1···
Connected to 127.0.0.1.
Escape character is '^]'.
result: VALUE testnginxlua 0 3
Connection closed by foreign host.
```

如果要使用 UDP，只需将配置“listen 1111”改成“listen 1111 udp”即可。

ngx_stream_core_module 在 TCP/UDP 应用中可以支持 Ngx_Lua 中的部分指令，具体请参考相关 Wiki，网址为 https://github.com/openresty/stream-lua-nginx-module。

17.4 多层级缓存

第 10 章介绍过 Ngx_Lua 的缓存功能，主要有 lua_shared_dict 和 lua-resty-lrucache 两种缓存方式，在使用上它们各有利弊，那么，如果能够取两家之长岂不妙哉？lua-resty-mlcache 就可以达到这样的效果，图 17-2 所示为 lua-resty-mlcache 在其 Wiki 中的缓存流程图。

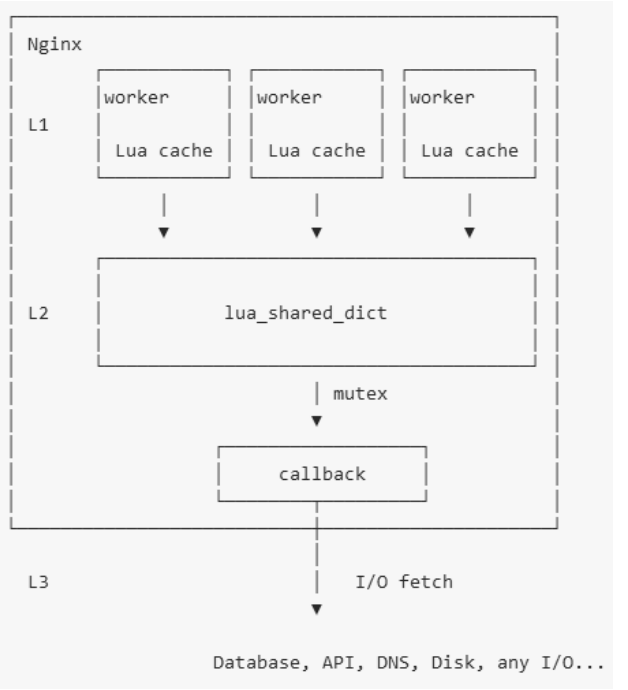


图 17-2　lua-resty-mlcache 在其 Wiki 中的缓存流程图

从图中可以看出，lua-resty-mlcache 将 lua-resty-lrucache 放在缓存的第 1 层，将 lua_shared_dict 放在缓存的第 2 层，后端的应用服务是第 3 层，这样做的好处如下。

- 第 1 层使用 lua-resty-lrucache，避免了 lua_shared_dict 缓存的锁竞争，同时能够提供热点数据，性能更好。
- 第 2 层使用 lua_shared_dict，如果第 1 层缓存未命中，则会到第 2 层中查询。
- 如果第 2 层仍然未命中，就会到后端服务中去获取缓存，此时需要使用 lua-resty-lock 来创建一个锁，控制同一个请求的并发访问情况，避免出现因缓存失效引起的“风暴”。
- 通过这种方式，可以让缓存提供更为稳定的读取效果，有兴趣的读者可以访问 https://github.com/thibaultcha/lua-resty-mlcache 进行了解。

17.5　lua-resty-core 扩展

lua-resty-core 是 OpenResty 的组件，与 Ngx_Lua 相比，它新增了很多指令，在功能上有所加强。下面将介绍 lua-resty-core 的两个简单但很常用的指令。想要了解更多的读者也可以访问 https://github.com/openresty/lua-resty-core。

17.5.1　字符串分割

使用 lua-resty-core 中的 ngx.re.split 指令可以对字符串进行分割，示例代码如下：

```
location / {
    content_by_lua_block {
       local ngx_re = require "ngx.re"
       -- 用空格来分割字符串，得到的 res 是 table 类型的
       local res, err = ngx_re.split("test nginx 123", " ")
       -- 循环输出分割后的数据
 for i, v in ipairs(res) do
           ngx.say(v)
       end
    }
}
```

执行结果如下：

```
# curl  http://127.0.0.1:80/t?sa
test
nginx
123
```

17.5.2　Nginx 进程管理

lua-resty-core 提供了 ngx.process 指令，它可以开启有特殊权限的 nginx agent 进程，该进程拥有和 master 进程相同的系统启动账号，例如 Nginx 是由 root 账号启动的，那么 master 进程和 agent 进程都是 root 账号的进程。agent 进程可以让 Nginx 的进程实现自我控制，如发送关闭 worker 进程的信号，示例代码如下：

```
init_by_lua_block {
local process = require "ngx.process"
-- 启动 agent 进程
    local ok, err = process.enable_privileged_agent()
    if not ok then
        ngx.log(ngx.ERR, "enables privileged agent failed error:", err)
    end
```

```
        ngx.log(ngx.INFO, "process type: ", process.type())
    }
    server {
        listen       80;
        server_name  localhost;
        location = /t {
            content_by_lua_block {
                local process = require "ngx.process"
                -- 发送信号让 worker 进程退出
                process.signal_graceful_exit()
                ngx.say("process type: ", process.type())
            }
        }
    }
```

配置好后，重载 Nginx 配置，会看到 Nginx 进程中多了一个 agent 进程：

```
    # ps ax |grep nginx
    10204   ?                    Ss            0:00   nginx:   master   process
/usr/local/openresty/nginx/sbin/nginx
    -c /usr/local/openresty/nginx/conf/nginx.conf
    10620 pts/0    R+     0:00 grep --color=auto nginx
    27634 ?        S      0:00 nginx: privileged agent process
    27642 ?        S      0:00 nginx: worker process
```

请求/t 的 URL，响应如下：

```
    # curl  http://127.0.0.1:80/t
    process type: worker
```

请求后，再次观察 Nginx 的进程，会发现 worker 进程的 PID 已经发生了变化：

```
    # ps ax |grep nginx
    10204   ?                    Ss            0:00   nginx:   master   process
/usr/local/openresty/nginx/sbin/nginx
    -c /usr/local/openresty/nginx/conf/nginx.conf
    11907 ?        S      0:00 nginx: worker process
    11909 pts/0    S+     0:00 grep --color=auto nginx
    27634 ?        S      0:00 nginx: privileged agent process
```

从上述示例可知，当 worker 进程关闭后，master 进程会重新启动新的 worker 进程，因此这是一种“友好”地重载 Nginx 配置的方式。使用过 Passenger 管理 Ruby 或 Node.js 进程的读者应该知道，重启 Ruby 进程或 Node.js 进程只需一个 touch 文件即可。在该文件的属性发生变化后，Nginx 会自动将 Ruby 或 Node.js 的进程重启。如果使用 Nginx 或 OpenResty 进行开发，在更新代码后可以执行 HTTP 请求来重启服务，也可以利用 touch 文件来重启服务，如果使用

touch 文件，则需要定时检查某个文件的创建时间是否发生了变化。

注意：在 Nginx 1.13.8 以上的版本和 lua-resty-core 0.1.14 版本上，可以使用 get_master_pid 指令来获取 master 进程的 PID。

17.6　全局唯一标识符 UUID

有时需要获取与某些请求相关的所有数据以完成全链路监控，类似于阿里巴巴的鹰眼系统。Nginx 作为前端入口可以生成一个全局唯一标识符 UUID（Universally Unique Identifier，通用唯一识别码），并将 UUID 通过请求头或其他方式传递给后端服务。可使用 lua-resty-jit-uuid 模块来生成 UUID 标识符。

首先，使用 OPM 安装 lua-resty-jit-uuid 模块：

```
# export PATH=$PATH:/usr/local/openresty/bin/
# /usr/local/openresty/bin/opm  get thibaultcha/lua-resty-jit-uuid
```

然后，修改 OpenResty 配置来生成 UUID：

```
init_worker_by_lua_block {
        local uuid = require 'resty.jit-uuid'
        uuid.seed()
    }
    server {
        listen 80;
        location / {
            content_by_lua_block {
                local uuid = require 'resty.jit-uuid'
                ngx.say(uuid())
            }
        }
    }
```

执行结果如下（每次输出结果都不相同）：

```
[root@VM_3_41_centos /]# curl  http://127.0.0.1:80/t?sa
b3ab29cb-dde8-4af9-9d86-2b18c3eae198
[root@VM_3_41_centos /]# curl  http://127.0.0.1:80/t?sa
7acd4fdc-e11b-4d5e-adea-620a5a234463
[root@VM_3_41_centos /]# curl  http://127.0.0.1:80/t?sa
90cceee4-d53d-43ea-a41c-598bfa6ba2c9
[root@VM_3_41_centos /]# curl  http://127.0.0.1:80/t?sa
18532eec-8af6-4301-8565-47abc9771666
```

17.7 “全家福”awesome-resty

随着 OpenResty、Nginx 开发的流行，相关模块也越来越多，而 awesome-resty 收集了很多优秀的 OpenResty、Nginx 模块，其中包括与 Web 开发框架、路由、中间件、API、身份验证、数据库访问、缓存等相关的各式各样的工具，

想使用这些模块的读者可以访问 https://github.com/bungle/awesome-resty 搜索需要的模块，以避免重复“造轮子”。

17.8 OpenResty，未来!

近年来，OpenResty 发展速度惊人，感谢各位互联网大咖对这门技术的支持和贡献，让我们这些使用者受益匪浅。但受限于各种因素，OpenResty 目前的使用人数仍不能与 Java、Python、PHP 这些语言相比，这也是 Ngx_Lua 难以作为团队开发工具的原因之一，但庆幸的是，基于 Ngx_Lua 的开发入门比较容易，代码量也小，是进行独立开发的良好选择。

OpenResty 的创始人章亦春及其合伙人已经成立公司并组成了一个开发团队对 OpenResty 技术提供商业支持，这让没有专业 OpenResty 开发团队的公司也可以享用强大的 OpenResty 的服务，对 OpenResty 的传播推广也有很大帮助。

目前，OpenResty 主要由 OpenResty 软件基金会和 OpenResty Inc.公司提供支持，走的是规范的、非盈利的模式，OpenResty 社区希望以此来保证该技术的不断更新和稳定发展，给使用该技术的开发人员以信心。

第 18 章
开发环境下的常见问题

在使用 Nginx 的过程中，经常会遇到一些让人头疼的小麻烦，时不时干扰着开发人员的前进步伐。本章将会介绍几个有代表性的常见问题。

18.1 被截断的响应体

如果安装 Nginx 来代理开发环境，在使用中可能会出现一个诡异的现象：当使用代码调用 Nginx 代理时，只会输出响应数据的一部分；但使用浏览器打开时，数据却是完整的。

这是因为使用浏览器打开时带有支持压缩的请求头，响应数据恰好在 Nginx 内存中被压缩后完整地发给了客户端；而使用代码调用时没有提供压缩头，响应数据只能以非压缩的状态返回，又由于在传输过程中响应内容的大小大于 Nginx 的 proxy_buffering 的值，需要用到临时文件存储响应数据，但启动 Nginx 时的用户是 nobody，该用户没有写硬盘的权限，所以就出现了这种现象。

建议：通过 error.log 可以快速定位此类问题，所以遇到这种现象要优先去查看 error.log 的信息；另外启动 Nginx 时需要指定用户，并给该用户读/写目录的权限。

18.2 “邪恶”的 if

当 if 身处 location 配置中时，往往会变得很“邪恶”。想必经常使用 Nginx 的读者遇到过这种情况，如果把 if 和 try_files 放在一起，try_files 可能无法正常运行。

示例如下：

```
location / {
   root html;
   try_files /ab.html /a.html @test;
 #  if ($arg_a = "1") {
 #      set $args "a=3&b=4";
 #  }
}
location @test {
    return 200 $args;
}
```

假设上述代码中的文件/abxx.html 和/dasda.html 都是不存在的，如果访问这两个文件，就会触发 try_files 到名为@test 的 loaction 中，此时的响应结果会返回 test 的输出。

那么，如果当前请求包含参数 a=1，并启用上述代码中被注释掉的代码。根据 if 判断，会将整个参数替换成 a=3&b=4。执行整段代码，结果是否与刚才一样呢？

并不一样，执行代码后，返回了 404 错误，原因如下。

if 会在自己的处理阶段创建一个隐式 location，执行完 if 里面的指令后，会执行 proxy 逻辑，但这个 if 中没有配置响应内容的执行代码，也就只能报 404 错误了。可以采用如下方式规避这种现象的发生。

- 少用 if 语句，Nginx 提供了多种配置方式，找到合适的替换方案并不困难。
- if 在 location 中使用容易出现上面的错误，它更适合在 server 块中进行配置。
- 可以使用 Ngx_Lua 来实现复杂的逻辑。

18.3 “贪婪”的正则匹配

location 支持正则表达式的匹配，这让 Nginx 的灵活度大大提升，但如果在使用过程设计不好，也容易出现一些问题。

如果有一个需求，需要迁移 testnginx.com/abc/x、testnginx.com/abc/c 和 testnginx.com/abc/[0-9]+的请求到新的应用服务器上。有的读者可能立马会想到将这些路由指向新服务，于是配置成下面的格式：

```
location ~ /abc {
}
```

但其实这种格式包含了多种可能，会匹配到不该匹配的内容，如/x/abc、/abcde、/abc/s、

/abc.html 等。

建议在每个 location 中严格定义正则匹配的范围，避免出现因“贪婪”匹配导致的异常，例如可以使用下面的定义范围：

```
location ~ ^/abc(/|$)   # 匹配开头是/abc/的目录或开头是/abc 的 URL
location ~  /abc$       # 只匹配结尾是/abc 的 URI
location /abc/          # 匹配整个目录
location ~ ^/abc/(x|c|[0-9]+)$ #这才是需要匹配的内容
```

18.4　规范 HTTP 状态码

如果在项目中需要对一些操作进行限制，不满足条件的操作会被禁止，最容易想到的是用不同的状态码来区分不同的情况，但结果可能会动用 401、403、404、405 等各种状态码。如果项目组中的每个人都这样做，那么自定义的错误码就会变得乱七八糟。

解决该问题的方法是制定统一的 HTTP 状态码使用规范，状态码和作用需要一一对应，并建立严格的配置审查制度，禁止开发人员私自使用未定义的状态码。

18.5　规范 URL

当需要提供一套 API 给调用方时，如果使用如下的 URL 格式：

```
testnginx.com/abc/[0-9]+/[a-z]-[0-9]+/api/f.json,
testnginx.com/[a-z]+/api/x.json
```

这里将正则表达式写在了 URL 中，看上去好像没什么问题，但是否真的如此呢？先来分析一下。

- 为了监控 URL 服务，需要做大量的正则判断来确认这些正则表达式是否属于同一个 URL 服务。
- 过多地使用正则表达式会让 Nginx 的配置变得臃肿，可读性不高。
- 使用第三方工具进行监控时也会很麻烦，大量的正则表达式会导致 URL 服务的监控被分散化，无法集中在同一个服务清单中，从而失去监控的意义。

解决思路如下。

- 减少正则表达式的使用，将与业务逻辑有关的信息存放在参数中。
- 如果必须使用正则表达式，尽量把它放到最后，并使用目录的形式来表示，因为使用目录查询的方式可以减少正则表达式的压力。

18.6 proxy_set_header 的误操作

如果需要在请求从 Nginx 进入后端服务时新增一个请求头，我们可能会这样配置：

```
server {
    listen 80;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_pass http://f_servers;

    location /abc {
        proxy_set_header Test_a 'a';
        proxy_pass http://f_servers;
    }
}
```

但结果很不幸，会返回错误的响应状态码 404，后端服务的日志显示没有找到对应路由，请求头 Host 不见了，为什么会这样呢？

原来，proxy_set_header 操作是在清空当前全部 proxy_set_header 的内容后再赋值的，如果上一层有 proxy_set_header 操作，在执行下一层代码时，需要把上一层的内容再补上，除非不再需要这些内容了，解决方案如下。

- 将 server 块的头在 location 中重新生成一次。
- 采用 Ngx_Lua 或 more_set_input_headers 来设置请求头。

18.7 开发环境下的证书问题

开发人员可能经常遇到这样的需求，新开发的 API 服务须提供 HTTPS 访问。出于安全考虑，开发环境不会有真实的 HTTPS 证书，虽然可以伪造证书，可手机 App 认证严格，有时伪造的证书并不好用，何况每个开发人员都去伪造证书，也不怎么“优雅”。有没有更好的解决方案呢？

当然有，不过此方案只适合用于 Nginx 是 HTTPS、请求代理到应用服务器时是 HTTP 的情况。如果前后端都是 HTTPS，可能要用到 HTTP 2.0，这不在本书讨论范围内。

首先，开发人员需要安装大于 1.10 的 Nginx 版本（因为它支持 TCP 代理），编译时开启--with-stream，配置 HTTPS 开发环境的代码如下：

```
user webuser webuser;
worker_processes 1;

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

events {
    worker_connections  1024;
}
# 开启 TCP 代理模式，将请求指向拥有真实证书的 Nginx
stream {
upstream backend {
    # HTTPS 证书管理平台
        server 192.168.100.22:443;
    }

    server {
        listen 443 so_keepalive=on;
        # proxy_protocol on;
        # proxy_bind $remote_addr transparent;
        proxy_connect_timeout 300s;
        proxy_timeout 300s;
        proxy_pass backend;
    }
}

http {
    include       mime.types;
    default_type  application/octet-stream;
    server {
   # 这里是配置开发环境的位置，所有的请求都只需配置 80 端口即可
   # HTTPS 请求会被代理到这里
        listen       80;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_hide_header X-Runtime;

        location / {
            proxy_pass  http://127.0.0.1:8081;
        }
    }

}
```

当开发人员的 Nginx 配置好后，开发人员的 443 端口会代理到拥有真实 HTTPS 证书的

Nginx 服务器，这台带有 HTTPS 证书的 Nginx 服务器（即上述配置中的 upstream backend 的地址 192.168.100.22）配置如下：

```
server {
    listen              443 ssl ;

    ssl_certificate /path/xxx.crt;
    ssl_certificate_key /path/xxx.key;
    location / {
        # $remote_addr 记录的是开发人员服务器的访问地址，将从 443 端口进入的请求解
        # 析后再反向代理回$remote_addr 的服务器，请求回源时已经是 HTTP 请求了
        proxy_pass http://$remote_addr:80;
    }

}
```

配置完成后，192.168.100.22 的服务器是由维护证书的运维人员管理的，其他人没有登录权限，下面给出 HTTPS 代理开发环境流程图，如图 18-1 所示。

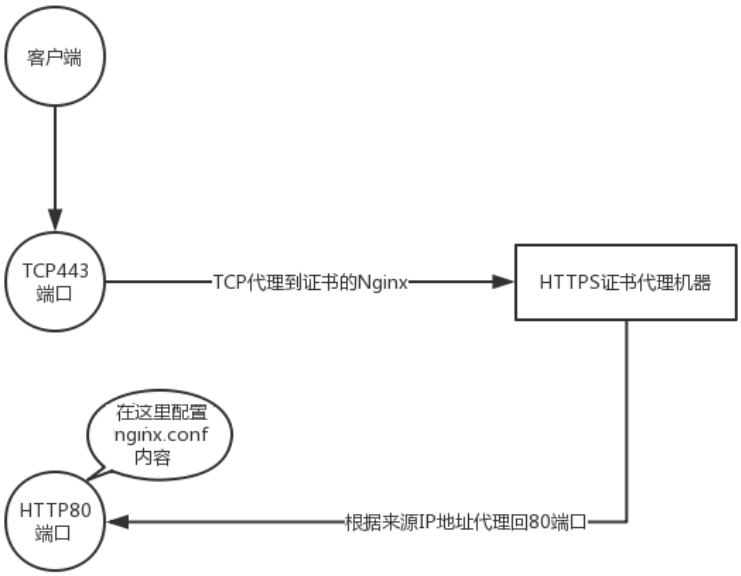


图 18-1　HTTPS 代理开发环境流程图

使用 TCP 代理，让 443 端口的数据透明地传送给 HTTPS 证书代理服务器，将请求解析后再代理回开发人员的 80 端口，这样开发人员只需要配置 80 端口的 Nginx 就可以了。

18.8　深层次的错误重定向

当代码无法提供服务时，可以给客户端提供一个友好的提示页面。这时如果使用 error_page 指令去完成错误重定向，可能并不会生效，原因是没有配置 proxy_intercept_errors 指令，它允许 error_page 拦截所定义的错误码的请求，须在 Nginx 的 http 块中激活该指令，方法如下：

```
proxy_intercept_errors on;
```

但如果重定向一次仍然无法满足需求，可以进行多次错误重定向，配置方式如下：

```
recursive_error_pages on;  # 允许 error_page 递归使用
proxy_intercept_errors on;
```

18.9　压测环境下的限速和短连接

代码开发完成后，就是压测环节了，把代码提交到压测平台上，并尽量让 Nginx 配置与线上环境保持一致。但压测数据出来后，可能会在返回的响应数据中发现大量的 503 错误状态码，查看代码却并无异常，查看 Nginx 日志会发现大量的限速信息。原来在压测环境下，Nginx 同步时将限速配置加了进来。当处在高并发下进行测试时，应该能够看到请求确实被限速了。对此有两种解决方案。

- 配置限速白名单，详见第 3.7 章。
- 将压测环境的限速关闭。

在压测时，还可能遇见另一种现象：每隔一段时间，服务的 QPS 就会下降到接近于 0，然后过一会儿又会恢复过来，并一直很有规律地持续着这种状态。这又是为什么呢？查询服务器资源，会发现原来是短连接太多，消耗了太多的 Linux 服务器端口资源，解决方案是加上长连接。

18.10　小结

本章介绍了在开发过程中经常遇到的一些问题，从中可以学到，该如何一步一步去深挖技术细节，解决现实问题。希望本章内容能够对读者有所帮助。

反侵权盗版声明

Nginx实战

本书主要讲解了Nginx在反向代理和应用开发中的作用，阅读本书可以了解Nginx在互联网开发中扮演的多个角色，充分利用这些角色的各项功能有助于提升服务的整体性能。该书所介绍的大部分功能是通过Nginx+Lua进行开发和配置的，但并不要求读者精通Lua，在必要的位置，会对Lua进行选择性讲解。

ISBN：978-7-121-35460-1

书价：79.00