学士学位论文

基于 IPv6 的 HTTP/2 过渡应用服务设计与实现

学　　号: 20151001270
姓　　名: 陈立翔
学 科 专 业: 计算机科学与技术
指 导 教 师: 陈伟涛 副教授
　　张峰　副教授
培 养 单 位: 计算机学院

二〇一九年五月

摘　　要

今天，网络早已成为人们生活中不可缺少的部分。同时随着我国网民人口的不断增长，互联网环境正在变得愈发复杂。在服务端，我们急需将我们旧有服务使用的 IPv4 和 HTTP/1.1 协议迭代更新至 IPv6 和 HTTP/2 协议以解决旧协议存在的一系列问题。如何将旧有的 Web 应用过渡支持 IPv6 和 HTTP/2 是我们目前面临的重要挑战。

本文对于如何让现有的 IPv4 环境下仅支持 HTTP/1.1 协议的 Web 应用能够在 IPv4 和 IPv6 双栈环境下工作并支持 HTTP/2 协议，同时尽可能提高服务端的并发性能，开展了以下研究工作：

1. 通过对 Go 语言技术的深入研究，分析其协程和并发模型的技术特点。并基于这些技术特点开发本网关。

2. 比较了 RFC 标准中 HTTP/1.1 和 HTTP/2 报文的区别，分析了如何将 HTTP/1.1 的报文加工处理成 HTTP/2 的报文。

3. 为了将 HTTP/1.1 应用过渡支持 HTTP/2 , 本文采用 Go 语言技术设计了一个 HTTP 网关。该网关通过反向代理技术，在无需对原有应用进行任何修改的情况下，将其升级为支持 HTTP/2 协议，并可在 IPv4 和 IPv6 双栈环境下工作。

4. 基于加权轮询算法和一致性哈希算法对该网关设计了负载均衡策略，使得该网关能够支持横向扩容的 Web 应用，应对更复杂的并发情况。

5. 对该网关进行了压力测试，并通过火焰图分析该网关在实际工作环境中的性能瓶颈。

综上所述，本文设计的基于 Go 语言的 HTTP 反向代理网关，能够在多平台下作为在 IPv4 和 IPv6 双栈环境下的 HTTP/2 过渡应用，并且具有较好的并发性能。其提供的负载均衡功能可以使得用户轻松对应用进行横向扩容。经过测试，具有一定实用价值。

关键词： IPv6 协议；HTTP/2 协议；HTTP 网关；反向代理；负载均衡；Go 语言；

Abstract

The Internet has become an indispensable part of people's lives nowadays. At the same time, with the continuous growth of the netizen population in China, the Internet environment is becoming more and more complicated. On the server side, we urgently need to iteratively update the IPv4 and HTTP/1.1 protocols used by our legacy services to the IPv6 and HTTP/2 protocols to resolve a series of problems with the old protocols. How to transition legacy Web applications to support IPv6 and HTTP/2 is an important challenge for us today.

In this paper, the following research work is carried out on how to make the Web application supporting only the HTTP/1.1 protocol in the existing IPv4 environment work in the IPv4 and IPv6 dual-stack environment and support the HTTP/2 protocol while improving the concurrent performance of the server. :

1. Through the in-depth study of Go language technology, analyze the technical characteristics of its coroutine and concurrency model. The gateway is developed based on these technical features.

2. Compare the differences between HTTP/1.1 and HTTP/2 messages in the RFC standard, and analyze how to process HTTP/1.1 messages into HTTP/2 messages.

3. In order to support the HTTP/1.1 application transition to HTTP/2, this article uses Go language technology to design an HTTP gateway. The gateway uses reverse proxy technology to upgrade to support the HTTP/2 protocol without any modifications to the original application, and can work in both IPv4 and IPv6 dual stack environments.

4. Based on the weighted round-robin algorithm and the consistent hash algorithm, a load balancing strategy is designed for the gateway, which enables the gateway to support horizontally expanded Web applications and cope with more complex concurrency situations.

5. The gateway was stress tested and the performance bottleneck of the gateway in the actual working environment was analyzed by flame graph.

In summary, the Go-based HTTP reverse proxy gateway designed in this paper can be used as an HTTP/2 transition application in IPv4 and IPv6 dual-stack environments under multiple platforms, and has better concurrency performance. Its load balancing feature allows users to easily scale applications horizontally. After testing, it has certain practical value.

Key Words: IPv6 Protocol; HTTP/2 Protocol; HTTP Gateway; Reverse Proxy; Load Balancing; The Go Programming Language;

目 录

第一章 绪论

1.1 研究背景及其意义

各种网络协议常常被称为互联网时代的基石。随着网络环境的变化，网络协议也需要适应时代的需求迭代升级。

从 1996 年开始，一系列关于 IPv6 协议的 RFC 标准发表出来，旨在取代原有的 IPv4 协议。IPv6 的地址长度为 128 位，能提供远多于 IPv4 的地址数量。在 IPv6 环境下，将不需要 IPv4 环境下为了解决地址数量不足而衍生出的 NAT 等技术，每个设备都能得到一个公网 IP，有利于物联网等技术的发展。但 IPv6 技术的推进还需要一段时间，在未来一段时间内我们都将面临着 IPv4 和 IPv6 共存的互联网环境。

2015 年发布的 HTTP/2 协议则旨在取代 HTTP/1.x 协议。为了提高网络传输的效率，更有效地利用网络资源，该协议增加了二进制分帧，多路复用和头部压缩等新的特性。同时目前浏览器中 HTTP/2 实现都强制要求数据通过 HTTPS 加密，而不是像 HTTP/1.x 协议那样将数据明文发送。这显著增强了数据传输的安全性，有效避免数据被劫持等安全隐患。

将现有的应用迁移升级到这些新协议上将会是一个漫长且耗资巨大的过程。如何在今天这种多种协议共存的情况下，将旧有的应用以尽可能低的成本过渡迁移到新协议上，且具有较好的性能？这是我们当前面临的一个重要问题。

1.2 国内外相关工作介绍

据统计，世界上前一百万个访问量最大的网站有 80% 以上使用的 Web Server 是 Apache HTTP Server 和 Nginx。其中 Nginx 的占有率为 44.2%，Apache HTTP Server 的占有率为 39.0%[1]。这两者都基于 C 语言编写，并提供反向代理功能。也有很多人对这些网关进行了二次开发，例如中国工程师章亦春等人基于 Nginx 开发的 OpenResty[2] 等。

这些网关在不断的迭代过程中也根据 RFC 标准陆续加入了 IPv6 和 HTTP/2 协议的支持，且能够对旧有的协议通过反向代理进行升级。它们由于久经实际生产环境考验，具有较强的稳定性。但由于一些历史遗留原因，也或多或少存在诸如开发效率较低，并发能力较差等问题。

接下来简单介绍这两个 Web Server 的优势和缺陷。

1.2.1 Apache

Apache HTTP Server（简称 Apache）是 Apache 软件基金会维护开发的一个开源 HTTP 服务器软件。它自 1995 年诞生以来，一直是世界上最被广泛使用的 HTTP Server。同时其也提供反向代理功能，并已加入 HTTP/2 和 IPv6 的支持。

Apache 在二十多年间已迭代数个版本，具有极佳的稳定性和大量的插件，能够帮助开发者快速构建小型 Web 应用。

但是 Apache 在应对并发请求时，使用的是传统上基于进程或线程模型架构，通过多线程来处理并发请求。事实上这种模型虽然稳定，却无法发挥出现代计算机的性能，后面在讨论 Go 语言的并发模型时将会谈到这一点。

1.2.2 Nginx

Nginx 是俄罗斯工程师 Igor Sysoev 在 2004 年发布的一款异步框架的高性能 HTTP 服务器，常被用作反向代理和负载均衡器。目前绝大多数高并发量的网站都使用 Nginx 作为 HTTP 服务器。它在不断的迭代中也陆续加入了 IPv6 和 HTTP/2 的支持。

Nginx 的出现就是为了解决 Apache 的并发模型无法解决的 C10K 问题。C10K 指的是能够使单机服务器承受一万以上的并发连接请求。Nginx 创造性地将模块化加入了设计之中，非常方便于二次开发。如由中国工程师章亦春等人开发的 OpenResty，阿里巴巴公司开发的 Tengine 都是根据实际需求基于 Nginx 二次开发的 HTTP 服务器。

Nginx 采用了 epoll 多路复用机制来应对高并发请求。epoll 是 Linux 2.6 版本内核之后加入的特性。Linux 提供了三种 I/O 多路复用模型，select, poll 和 epoll。它们分别是基于数组，链表和哈希表保存文件描述符的 I/O 多路复用模型。显然 2.6 版本加入的 epoll 模型所使用的哈希表具有最低的时间复杂度，这可以大大减少操作系统切换上下文的开销，应对高并发连接。根据官方的测试结果，在理想情况下，Nginx 单机可以应对五万个并发连接，而在实际的运作中，可以支持二万至四万个连接。

但是这也给 Nginx 带来了一个问题. 在非 Linux 系统下, 无法使用这些操作系统底层的 API 来进行 I/O 多路复用，这时其并发性能会大大下降。对于情况较为复杂的跨平台服务端解决方案来说，Nginx 并不合适。

1.2.3 小结

综上两个网关都具有优异的特性，但它们分别具有两个较大的缺点：抗并发能力较差和跨平台能力较差。那么，能否设计一个支持 IPv6 和 HTTP/2 协议的反向代理网关，且具有较强的跨平台能力和并发性能? 本文针对这一问题展开了研究。

1.3 论文主要工作和章节结构

本文主要针对如何在 IPv4 和 IPv6 双栈环境下将旧有的 HTTP/1.1 的 Web 服务过渡升级支持 HTTP/2 设计了一个网关。该网关通过反向代理的形式升级协议。并且该网关通过 Go 语言编写，利用了其强大的并发模型，具有高并发和易跨平台的特性，避免了上述解决方案的一些缺点。

论文的内容和章节结构如下：

1. 第一章：绪论。主要介绍了研究背景，以及通过反向代理升级 IPv6 和 HTTP/2 协议的这方面国内外的一些成熟解决方案及其存在的问题，以及本文主要工作安排。

2. 第二章：Go 语言服务端相关技术介绍。简单介绍了 Go 语言的背景和一些内置库实现，并分析了其并发模型，说明了 Go 语言应用在本项目的一些优势。

3. 第三章：HTTP/1.1 到 HTTP/2 的转换实现。深入分析了 RFC 标准中 HTTP/1.1 协议和 HTTP/2 协议的差异，根据这些差异拟定了程序中通过怎样的流程去升级 HTTP 协议。同时也对广泛使用的 Websocket 协议做了处理。

4. 第四章：IPv4 和 IPv6 双栈环境下反向代理功能的实现。介绍了本设计的核心功能，反向代理功能是如何实现的，以及程序如何在 IPv4 和 IPv6 双栈环境下工作。并介绍了通过加权轮询算法和一致性哈希两种算法设计的两种负载均衡策略。

5. 第五章：性能分析。通过压力测试和火焰图两种形式对本设计的性能进行了分析。

6. 第六章：总结和展望。总结了本设计的功能、特性和缺点，阐述了一些辄需改进的地方。

第二章 Go 语言相关技术介绍

2.1 Go 语言简介

Go 语言（简称 Go）是由美国谷歌（Google）公司发布的一种静态强类型、编译型、并发型并具有垃圾回收功能的编程语言[3]，于 2009 年推出最初版本并开源。其作者 Rob Pike 和 Ken Thompson 等人均曾经在美国贝尔实验室工作，参与了 C 语言和 Unix 系统的开发工作。

Go 的发明是为了解决 C/C++ 在服务端开发时的一些问题。在面向高并发环境的程序开发时，C++ 由于其过多的语言特性，导致程序员对语言的学习成本极高；C 语言则恰好相反，过于简单的语言特性和较低的抽象层次，使得编写代码本身就十分复杂。同时它们较为落后的包管理机制不利于对代码的高效复用。依赖于系统底层 API 的网络操作等也使得其跨平台能力不强。

Go 本身语法大量借鉴了 C 语言，并简化了一些部分，其关键字数量比 C 语言还要更少。对于有一定 C 语言开发经验的开发者来说学习成本较低。Go 内置大量网络编程相关的库，可以大大简化网络编程的工作，更重要的是，其实现不依赖于任何操作系统底层 API，这使得 Go 编写的程序可以较为容易地交叉编译并在多平台运行。

Go 也在一定程度上支持一些面向对象编程的方法。其程序以包的形式组织，包实质上是一个包含 Go 文件的文件夹。包内的代码共享相同的命名空间，为区分包内包外的私有成员，Go 将以大写字母开头的对象（包括函数，变量等）对其它包可见，其它的则不可见。

Go 支持在多平台运行和交叉编译。Go 语言在设计时尽可能避免使用操作系统的底层 API，在语言层面实现了并发模型，因此在不同平台和不同架构的处理器上具有一致的表现。目前 Go 支持包括 Linux，Windows，FreeBSD，Sun 在内的多种操作系统，也支持 X86，MIPS，ARM，RISC-V 在内的多种处理器架构。

2.2 Go 语言的并发模型

Go 语言最为重要的特性来源于其在语言层面对并发的支持。这里讨论的是 Go 1.7 版本及之前的经典并发模型。虽然在后期版本增加了一些新的功能或语法糖，但本质上还是离不开我们后面讨论的这些并发模型。接下来简单介绍一些并发模型的用法。

2.2.1 协程（Coroutine）

我们知道，线程和进程都是在操作系统层面上实现的。相对于进程来说，线程可以减少调用时的上下文开销，具有更小的颗粒度。

但是对于服务端接受 HTTP 请求这样的需求来说，线程的粒度依然过大。对每个请求或 TCP 连接开一个线程来处理的话，会有大量的 I/O 阻塞占用 CPU 时间，造成资源的极大浪费。

Go 在语言层面上实现了协程，通过关键字 go 来进行调用。协程拥有自己的 Runtime，每个协程都有自己的堆栈，Go 封装了自己的一套协程调用方法，实际还是通过线程来执行任务。Go 的协程在理想状态下可以达到百万级并发[4]。

2.2.2 通道（Channel）

协程是在语言层面上实现的，因此我们不能使用操作系统中实现的诸如信号量这样的 API 对其做并发操作。Go 语言是在语言层面上实现了一种称之为通道（Channel）的数据结构，来进行协程间通信的操作。

Channel 本质上是一个先进先出队列（FIFO），协程内可以通过 select 关键字监听 Channel 中是否有元素可以做出队操作，当 Channel 中没有元素时协程阻塞。协程中同样也可以对 Channel 直接做写入操作，当队列满时也发生阻塞。通过这样的机制就能实现协程之间的通信。

2.2.3 上下文（Context）

上下文（Context）在与 API 和慢处理交互时可以派上用场，特别是在生产级的 Web 服务中。

Context 实现的主要功能为在协程之间共享状态变量，和在被调用的协程外部，通过设置 ctx 变量值，将“过期或撤销这些信号”传递给“被调用的程序单元”。在网络编程中，若存在 A 调用 B 的 API, B 再调用 C 的 API，若 A 调用 B 取消，那也要取消 B 调用 C，通过在 A、B、C 的 API 调用之间传递 Context，以及判断其状态，就能解决此问题。

第三章 HTTP/1.1 到 HTTP/2 的转换实现

要想实现 HTTP/1.1 到 HTTP/2 的转换，我们首先要了解这两个协议的特性及其异同，针对性地设计转换的方法。

世界上最早的 HTTP 协议标准源于 1991 年发布的 HTTP/0.9 标准。该版本及其简单，只有一个 HTTP 方法 GET。它具有以下特点：

- 客户端/服务端响应都是 ASCII 字符
- 客户端请求由一个回车符（CRLF）结尾
- 服务器响应的是超文本标记语言（HTML）
- TCP 连接在文档传输完毕后断开

显然这个协议是十分简陋的。它无法处理用户登录等稍微复杂一点的操作，以及各种富文本。1996 年 IETF 发布了正式的 HTTP/1.0 标准，相较于之前的版本它有一些明显的特性：

- 服务端响应增加了响应状态
- 请求和响应增加了多行首部
- 响应内容不再局限于 HTML

HTTP/1.0 依然存在很多问题和值得优化的地方。比如对 TCP 连接的复用，大文件传输的优化，浏览器缓存较差等等。1999 年发布的 HTTP/1.1 就是为了解决这些问题的[5]。该协议一直到 2014 年，依然在不断地进行修订。它具有以下几个显著特性：

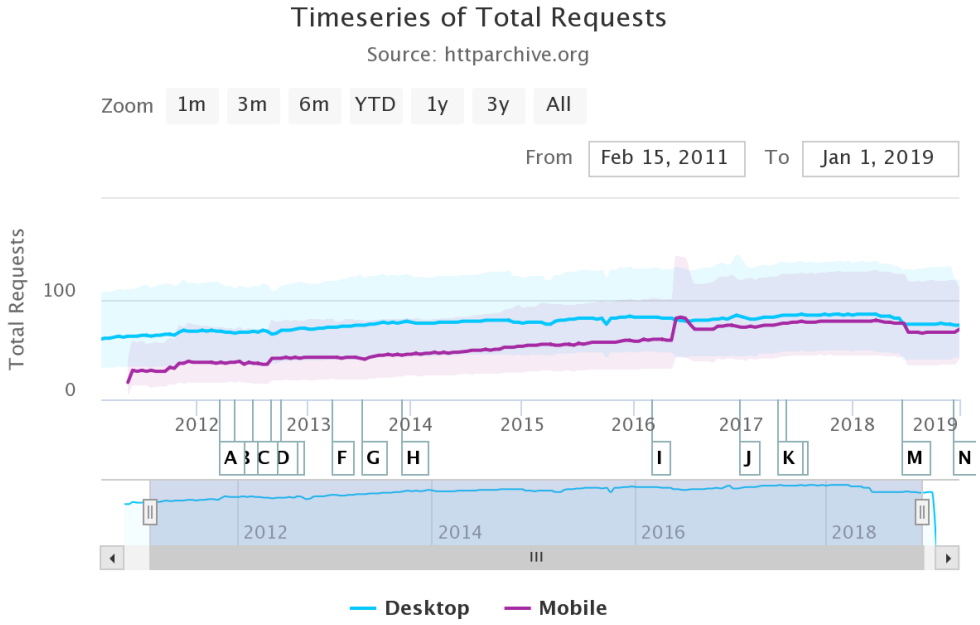
- 持久连接（Keep-Alive 机制）
- 传输编码（Chunk 机制）
- 字节范围请求（Accept-Ranges）
- 增强的缓存机制（协商缓存和强缓存）

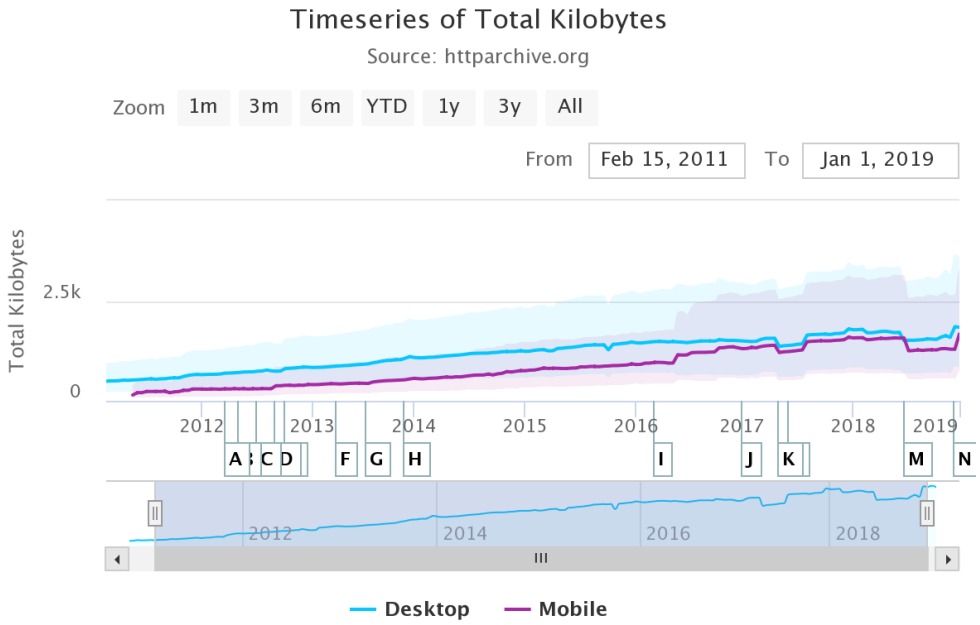
HTTP/1.1 已经有二十年的历史，并且依然在被广泛使用，十分稳定。但 2015 年，IEFT 依然发布了 HTTP/2 用于取代 HTTP/1.1。这源于互联网快速的发展，使得其无法适应当前的需求。

本章介绍 HTTP/1.1 存在的问题，HTTP/2 的新增特性和在 Go 语言下将 HTTP/1.1 报文转换为 HTTP/2 的方法。以及浏览器和服务端之间协议的协商机制和对目前广泛存在的 WebSocket 协议的处理。

3.1 HTTP/1.1 的困境

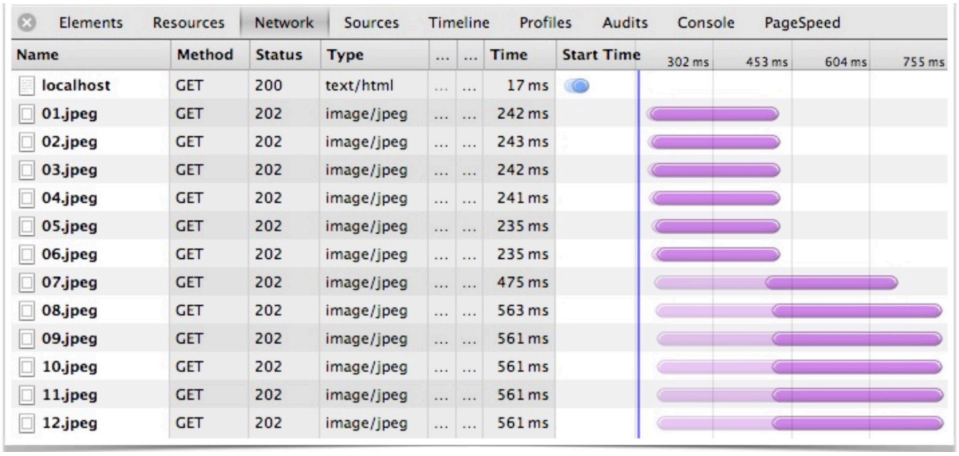
如今的 Web 应用已经大大超出了人们在最开始对 Web 的想像。在早期，一个网站往往只有一个简单的 HTML 页面。但是在今天，任意一个门户网站都有大量的流媒体，AJAX 请求，图片……人们也不再仅仅使用 PC 来浏览网页，移动端和物联网设备逐渐成为了主流。这种情况下，一个 Web 页面中的 HTTP 请求数和网站所占用的通信和存储空间会大大增加。httparchive.org 对目前互联网的的大量网站做了监测，结果如图所示[6]。





从图中可以看出，随着时间的推移，浏览一个网站平均需要的 HTTP 请求数量和资源的大小都在稳步上升。目前在没有缓存的情况下，在 PC 端浏览每个网站平均需要下载 2000kb 以上的资源，发生 70 个以上的 HTTP 请求。在移动端这个数字还要更高。这给了网络传输以很大的挑战。事实上，用户浏览网站时，在等待的过程中，平均有 69.5% 的时间是耗费在网络操作的阻塞上的。

而 HTTP/1.1 协议本身对并发连接并没有做很好的优化。Keep-Alive 机制本身只能复用连接，但是对并发连接并没有什么帮助。早期的标准 RFC2616 甚至规定浏览器中同域名只能有两个连接[7]。RFC7230 虽然去掉了这一限制，但现代浏览器中依然限制同域最多只能有 6 个并发连接。在浏览器中打开一个 HTTP/1.1 的网站，我们很容易在控制台中看到网络资源的加载时序，如图所示。



3.2 HTTP/2 的新增特性及转换实现

针对上文提到的 HTTP/1.1 的问题，2015 年 IETF 发布的 HTTP/2 协议新增了若干特性：

- 引入了流（Stream）的概念，取消了 Keep-Alive 而采用多路复用机制。
- 引入二进制分帧，将消息分割为更小的帧，并采用二进制格式进行编码。其中头（Header）会被专门封装到 Headers 帧中。
- 引入头部压缩机制，将一些常用头部用单字节表示，缩减头部大小。

这些机制解决了并发连接的问题，并在一定程度上缩减了每次 HTTP 传输的大小，显著提高了响应速度和 TCP 传输的效率[8]。

当然，HTTP/2 还不只增加了这些特性。HTTP/2 设计时还考虑到其作为一个多用途的网络通讯协议，因此还增加了诸如服务端主动推送这样的特性。目前现代浏览器还没有实现这些特性，暂不在本设计的考虑范围之内。特别需要注意的是，HTTP/2 目前有 H2C 和 H2 两种实现类型[9]。H2C 是目前主流浏览器所实现的

类型，该实现较为强调传输的安全性，强制通过 TLS 加密传输的数据。H2 多用于非 Web 等对安全性要求较低的应用场景，例如 RPC 调用等。下文所讨论的均默认为 H2C 的实现。

对于协议的处理，本网关最重要的功能就是将用户发送的 HTTP/2 的请求修改为 HTTP/1.1 的，以及将 HTTP/1.1 的响应修改为 HTTP/2 的。本节讨论的就是如何对报文处理以达到我们的目的。

HTTP/2 的报文格式在设计时尽可能地兼容了 HTTP/1.1，请求报文基本上只需要修改头部请求的协议即可。

对于响应的报文则需要做如下操作：

- 去除 Keep-Alive 相关的头部，包括 Connection 和 Keep-Alive。
- 对头部采用 HPACK 算法[10] 进行压缩。

之后将处理的报文按照 HTTP/2 的二进制帧的格式写入即可。

3.3 HTTP/2 协商机制

在 HTTP/1.1 和 HTTP/2 共存的环境下，浏览器和客户端要怎么样才能知道到底应该选择哪个协议来沟通？毕竟还有很多低版本浏览器不支持 HTTP/2。本节将介绍目前主流的客户端与服务端的协商机制，这一机制也用在了我们设计的网关当中。

这就要谈到 HTTP/1.1 中的 Upgrade 机制，这一机制能够使得客户端和服务端之间借助已有的 HTTP 语法升级到其它协议。对于 H2C 的升级，正是借助 Upgrade 来实现的[11]。

如果客户端（浏览器）支持 HTTP/2，就必须通过 Upgrade 头部列出要升级的协议和版本，并且还要包含一个 HTTP2-Settings 的头部，示例报文如下：

```
GET / HTTP/1.1
Host: www.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64url encoding of HTTP/2 SETTINGS payload>
```

如果服务器不支持 HTTP/2 便会忽略 Upgrade 头部，直接响应 HTTP/1.1 的报文。示例报文如下：

```
HTTP/1.1 200 OK
Content-Length: 1234
Content-Type: text/html
...
```

反之，如果服务端支持 HTTP/2，那就可以回应 101 状态码及其对应头部，并且在响应正文中直接传送 HTTP/2 的二进制帧。示例报文如下：

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c
HTTP/2 connection...
```

以上就完成了客户端和服务端协商使用哪种协议的流程。该流程能够确保在协议不冲突的情况下尽可能地强制客户端和服务端通过 HTTP/2 进行通信。

3.4 WebSocket 支持

WebSocket 协议本身不是 HTTP 标准的一部分。HTTP/1.1 协议的重要缺陷之一就是难以进行交互式通信，对于实时聊天和长轮询等功能只能通过开启大量 HTTP 请求来解决。为了解决这个问题，IETF 专门设计了一个新的协议 WebSocket 用于交互式通信的场景，其协议标准在 RFC 6455 中规定[12]。

3.2 节中提到，HTTP/2 引入了服务端推送机制，如果能够在应用开发中使用这一机制，那么是能够有效替代 WebSocket 协议的。但目前不论是客户端还是我们所要升级的目标服务，都是采用 WebSocket 协议。因此我们要在网关中加入对反向代理 WebSocket 协议的支持。这需要我们分析 WebSocket 协议的建立与传输流程，并针对性地设计反向代理的方法。

在 HTTP 网关的部分，我们要解决的问题是客户端与服务端之间如何从 HTTP 协议切换到 WebSocket 协议。3.3 节中提到的 HTTP/1.1 的 Upgrade 机制同样适用于此处。需要注意的是，HTTP/2 不提供 Upgrade 机制，因此下文所讨论的依然是基于 HTTP/1.1 展开。

通过 Upgrade 机制升级 WebSocket 协议的示例报文如下：

```
GET /chat HTTP/1.1
Host: www.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat
```

Sec-WebSocket-Key 是一个客户端随机生成的值，它用于唯一标识一个 WebSocket 连接。Sec-WebSocket-Protocol 表示最终使用的协议。如果服务端支持 WebSocket 的协议的话，将会返回 HTTP 101 响应，示例报文如下：

```
HTTP/1.1 101 Switching Switching Protocols
Connection: Upgrade
Upgrade: websocket
...
```

至此，HTTP 网关本身负责的 HTTP 协议的部分已经结束了。接下来数据帧的格式直接交给源站处理即可，我们的网关只需开辟一条从客户端到源站的 TCP 连接即可完成源站与客户端接下来的 WebSocket 协议通信。

3.5 HTTPS 支持

HTTPS（Hypertext Transfer Protocol Secure）是以安全为目标的 HTTP 通道。它通过多种对称加密和非对称加密算法对 HTTP 传输内容进行加密。

HTTP/2 协议本身并不要求是加密的，但目前所有的浏览器都强制要求 HTTP/2 协议传输的数据是 HTTPS 加密的。在具体实现中，我们可以认为浏览器中 HTTP/2 是在应用层中的 TLS（Transport Layner Security）[13] 层之上进行传输的，HTTP/2 和 TLS 共同构成了 HTTPS。

为了使得我们的网关能够支持现代浏览器，在本设计中加入 HTTPS 支持。在 TCP 传输时增加 TLS 操作，加密数据。本设计是通过 Go 语言中 crypto 库中已经实现的 TLSv1.2 标准进行加密的，目前包括 Chrome，FireFox 在内的主流浏览器最近的几个大版本均支持这一标准。在运行程序之前，用户也需要先做好 HTTPS 证书相关的配置。

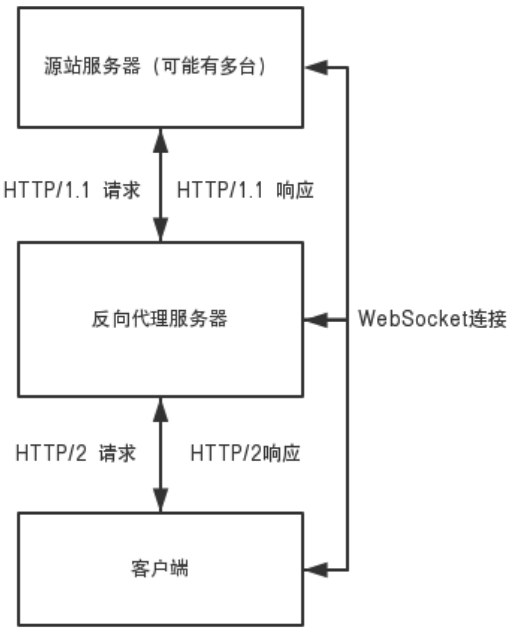
第四章 IPv4 和 IPv6 双栈环境下反向代理功能的实现

设计的网关是在反向代理的过程中完成前一章所讨论的对 HTTP/2 协议的处理，本章讨论了反向代理的实现和其如何在 IPv6 和 IPv4 双栈环境下工作。并设计了负载均衡策略以满足高并发环境下的业务需求。

4.1 反向代理的实现

前面提到，本网关通过反向代理功能升级 HTTP 服务。反向代理本身的功能十分简单，服务器根据客户端的请求，从其关系的一组或多组提供 HTTP 服务的源站服务器上获取数据，然后再将这些数据返回给客户端。客户端只会得知反向代理服务器的 IP 地址，而不知道在代理服务器后面的服务器集群的存在[14]。

前一章提到的 HTTP/1.1 报文转换为 HTTP/2 报文就是在反向代理过程中实现的。反向代理服务器收到客户端的 HTTP 请求后，将报文修改为 HTTP/1.1 格式的，发送到源站服务器；源站服务器响应后再将源站返回的 HTTP/1.1 的报文修改为 HTTP/2 格式的，返回给客户端。当然，这其中还需要注意 HTTP 超时和源站崩溃的问题，设计中通过上文提到的 Context 监听这种异常情况。对于出现异常情况，根据 HTTP 状态码的定义响应对应的状态码即可。流程图如图所示：



4.2 负载均衡策略

高并发环境下，为了减少由于访问量过多引起服务器系统崩溃的情况，我们常常需要对源站进行动态扩容。这其中最简单的方式就是横向扩容，通过多个相互独立的服务器组成一个集群服务器系统，从而较好地应对高负载情况。

为了达到这一目的，我们设计的网关需要提供负载均衡的功能。即把转发的 HTTP 请求分摊到不同的源站进行处理，降低单一机器的负载压力[15]。为了尽可能地利用有限的服务器资源，我们的负载均衡策略均要求具有一定的平滑性和均衡性。

本设计针对两种较常出现的业务场景，根据目前较为成熟的算法，设计了两种不同的负载均衡策略。一种是业务简单且对负载均衡性能要求极高的场景，采用加权轮询算法实现；一种是存在 HTTP 会话的场景，即用户可能需要通过 Cookie 等手段与同一台源站服务器保持一段时间的会话。这时如果将同一用户的不同 HTTP 请求发送到不同服务器会导致我们的业务流程无法正常完成。这时采用一致性哈希算法，将同一 IP 来源的用户的请求发送到同一台服务器，并使得请求尽可能均摊到不同服务器上，且在出现节点故障时能够以最小的损耗将请求迁移到其它机器。本节讨论了这两种策略的具体实现。

4.2.1 加权轮询算法（Weighted Round Robin）

轮询（Poll）本身是一种非常简单的算法。假设有我们有三台源站服务器 ServerA，ServerB，ServerC，那么我们可以生成一个序列 {ServerA, ServerB, ServerC}，对这个序列进行不断地遍历来获得轮询结果。

当我们需要的源站服务器具有不同的权重时，情况就会比较复杂了。假设对于上面三台服务器，分别具有 3，2，1 大小的权重，那么我们通过轮询算法生成的序列就是 {ServerA, ServerA, ServerA, ServerB, ServerB, ServerA }。对这个序列遍历获取轮询结果确实能达到一定的负载均衡的效果，但这种负载均衡策略不具有平滑性和均衡性，在不同的时间不同机器的负载是有很大差别的。

Nginx 对上述的加权轮询算法进行了改进，设计了一种平滑的加权轮询算法[16]。本设计使用了与 Nginx 中一致的加权轮询策略。

该加权轮询策略首先需要给每个源站服务器分配三个权重。weight 是每个源站服务器恒定不变的负载权重；effective weight 是实际有效权重，一开始与 weight 大小相同，用于根据实际情况动态调整的权重；current weight 是一个在运行过程中动态调整的值。该策略的具体步骤如下：

- 对每个请求，遍历所有源站服务器，对每个源站服务器的 current weight 的值

自增 effective weight 值的大小。同时累加所有源站的 effective weight，保存为 total。

- 从所有源站服务器中选出 current weight 最大的一个，作为本次命中的源站服务器。
- 对本次选中的源站服务器，对其 current weight 的值自减 total 的值。

使用这一策略对上文提到的权重分别为 3，2，1 的三个源站服务器，其生成的轮询序列示例如下：

请求序号	选择前的 current weight	选中源站服务器	选择后的 current weight
1	{3, 2, 1}	ServerA	{-3, 2, 1}
2	{0, 4, 2}	ServerB	{0, -2, 2}
3	{3, 0, 3}	ServerA	{-3, 0, 3}
4	{0, 2, 4}	ServerC	{0, 2, -2}
5	{3, 4, -1}	ServerB	{3, -2, -1}
6	{6, 0, 0}	ServerA	{0, 0, 0 }

可以看出该策略生成的序列较为平滑，且满足负载均衡中均衡分布的要求。6 次轮询过后 current weight 回到了 0, 0, 0，意味着若继续执行该算法，将不断生成相同的序列。

4.2.2 一致性哈希算法（Consistenting Hashing）

一致性哈希算法由美国麻省理工学院在哈希算法的基础上提出[17]，它的目标是解决动态系统中均衡分布的问题。与直接采用一般哈希算法取模的做法比起来，该算法具有更强的平衡性、单调性、平滑性和分散性，适用于因特网中的热点问题，同样也适合本设计中的负载均衡策略。本设计就以用户 IP 作为 Hash Key，通过一致性哈希算法解决本设计中特定场景的负载均衡问题。

本设计根据实际需求实现了如下描述的一致性哈希算法中的六个步骤，实现了取模、映射、删除服务器节点的功能，达到了均衡和单调的要求。

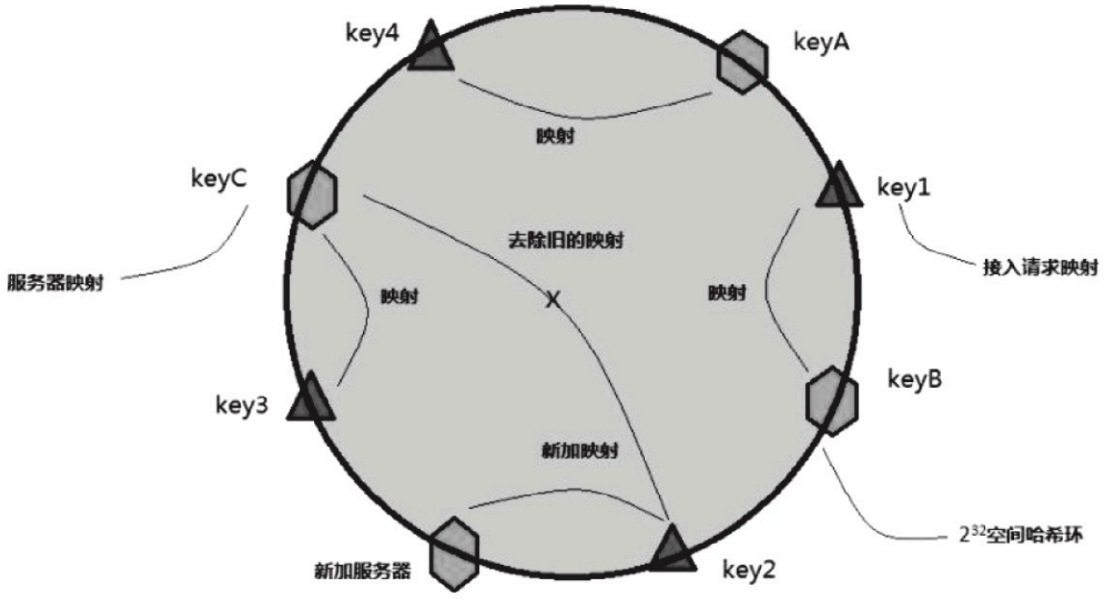
构造环形哈希表 首先创建一个 32 位大小的 key 值，然后将这个数据空间首尾相连，形成一个环形哈希环。

将请求映射到哈希环 将用户请求时的 IP 地址作为字符串处理，采用一般哈希算法，将这个字符串计算得到一个 32 位的值，对于请求 ReqN 我们得到 KeyN，然后映射到哈希环上。本例中采用的是 Go 语言内置库 crypto 中实现的 MD5 算法。

将源站服务器映射到哈希环 如同上一步骤一样，通过一般哈希函数，以服务器 IP 作为 Hash Key 将服务器也处理成一个 32 位的值，映射到同一个哈希环中，对于 ServerM 我们可以得到一个 KeyM。增加和减少源站服务器时，在环形哈希环上作适当的增删。

将请求映射到源站服务器 顺着哈希环的顺时针方向，从请求 Key1 开始查找，一直到遇到第一台服务器 Server1 为止，将该请求 Key1 映射到 Server1 上。由于这两者我们计算出来的哈希函数是固定的，因此它们的映射关系是唯一的，可以保证同一 IP 的用户会一直映射到同一台源站服务器上。以此类推，对应映射 ReqN 到 ServerM 上，一直到所有的请求和服务器映射关系建立完成。

动态增减源站服务器 一致性哈希算法的重要特点之一就是支持对源站服务器的动态增减，使得我们在特定场景下能够以较少的损耗动态迁移源站服务器。当要增加服务器 Server(M+1) 时，通过一般哈希算法得出 Key(M+1)，该 Key 数值介于原来的两台服务器的值之间，虽然新增了服务器，但只需要重新映射 Key(M+1) 逆时针出发遇到的第一台服务器 Key 之间的请求，其余的接入请求和服务器保持原有的映射关系不变。移除服务器 KeyB 时，同理，受影响的也只是逆时针出发达到的第一台服务器 KeyA 的接入请求，也只需要将 Key1 和新服务器进行重新映射即可，如图所示。



通过上述六个步骤，即可完成基于一致性哈希算法的负载均衡策略。

4.3 IPv6 环境下工作

IPv6 是网际协议的最新版本，用作互联网的网络层协议。用它来取代 IPv4 主要是为了解决 IPv4 地址枯竭问题，同时它也在其他方面对于 IPv4 有许多改进。

在网络上，数据以分组的形式传输。IPv6 定义了一种新的分组格式，目的是为了最小化路由器处理的消息标头。由于 IPv4 消息和 IPv6 消息标头有很大不同，因此这两种协议无法互操作。但是在大多数情况下，IPv6 仅仅是对 IPv4 的一种保守扩展。除了嵌入了互联网地址的那些应用协议（如 FTP 和 NTPv3，新地址格式可能会与当前协议的语法冲突）以外，大多数传输层和应用层协议几乎不怎么需要修改就可以在 IPv6 上运行[18]。

我们开发的网关是在应用层开发的，因此对于 IPv6 不需要过多的处理。事实上，Go 的内置的 net 库所有的操作均已支持对 IPv6 地址的处理。只需要让设计的网关支持在多 Host 和多端口监听即可。

第五章 性能分析

本章对设计的网关模拟高并发场景，进行系统测试。主要分两部分进行测试。一部分是压力测试，测试了系统极吞吐量和响应时间。另一部分是对网关内各个调用开销的分析，使用了火焰图进行可视化分析。

本次测试使用的计算机的基本配置如下：

- CPU：Intel(R) Xeon(R) CPU E5-2660 v3
- 物理核数：8
- 逻辑核数：8
- 内存：32G
- 操作系统：Ubuntu 17.10 artful, Linux Kernel 4.13.0-46-generic

5.1 压力测试

5.1.1 分析内容

压力测试主要分析本文设计的网关在不同的极限条件下和线上环境下本网关的性能。分别测试网关处理请求的极限，吞吐量极限和具有一定延迟的线上环境下的表现。主要指标是请求的最小，最大和平均响应时间。吞吐量的指标是每秒传输的流量大小。

5.1.2 方案设计

压力测试主要使用 nghttp2 这个工具来完成。该工具通过简单的命令行参数就能完成不同的测试方法。测试的具体方案如下：

1. 源站放在服务器本地，源站提供简单的静态页面的访问，单个资源大小均小于 20KB。nghttp2 随机访问任意 URL。nghttp2 使用 10 个线程进行请求，共计请求一万次。测试网关能够处理的并发请求极限。

2. 源站放置在服务器本地，源站提供几个 4GB 以上的大文件的静态资源访问，开启十个线程 GET 访问的资源。并比较关闭 HTTPS 支持时的差别。测试网关的极限流量吞吐量。

3. 源站放置在远程服务器上，模拟线上环境，源站返回响应存在一定延时。源

站的资源与测试 1 相同，但是平均有 200ms 左右的延时且较不稳定。测试方法与测试 1 也相同。测试线上环境下的表现。

压力测试在同等环境下对 Nginx，Apache 和本网关进行比较。Nginx 使用的 1.13.0 stable 版本，Apache 使用 2.2.0 版本。Nginx 和 Apache 配置较为复杂，测试时尽量设置成本网关类似的配置，例如 TLS 协议使用 TLSv1.2 版本。

5.1.3 结果分析

本网关测试结果：

1. 测试 1 总计在 2.85s 完成了所有请求，平均每秒完成 3507.9 个请求，失败请求数为 0。其它统计数据如下：

	Min	Max	Mean
time for request	1.15ms	4.13ms	2.83ms
time for connect	3.54ms	11.27ms	8.16ms
time to 1st byte	9.43ms	13.07ms	10.39ms

2. 运行测试 2 时平均传输流量速率为 124.78M/s。但关闭 HTTPS 支持之后则能够达到 2000M/s 以上，不难发现流量瓶颈主要来源于网关加密资源带来的开销。

3. 测试 3 总计在 296.02s 完成了所有请求，平均每秒完成 337.5 个请求，失败请求数为 2（发生了超时，响应 500 状态码）。反向代理产生的延迟和直接请求源站并没有显著差别，反向代理产生的损耗可以忽略不计，统计数据如下：

	Min	Max	Mean
time for request	76.23ms	1566.30ms	218.10ms
time for connect	17.52ms	19.17ms	18.39ms
time to 1st byte	115.38ms	619.48ms	234.98ms

Nginx 测试结果：

1. 测试 1 总计在 1.99s 完成了所有请求，平均每秒完成 5025.9 个请求，失败请求数为 0。其它统计数据如下：

	Min	Max	Mean
time for request	0.98ms	5.34ms	2.76ms
time for connect	3.85ms	10.98ms	8.01ms
time to 1st byte	8.30ms	14.76ms	9.98ms

2. 运行测试 2 时平均传输流量速率为 137.82M/s。

3. 测试 3 总计在 313.86s 完成了所有请求，平均每秒完成 318.61 个请求，失败请求数为 4（发生了超时，响应 500 状态码）。具体统计数据如下：

	Min	Max	Mean
time for request	70.23ms	1578.31ms	209.10ms
time for connect	15.57ms	20.01ms	16.77ms
time to 1st byte	99.37ms	657.23ms	220.47ms

Apache 测试结果：

1. 测试 1 总计在 8.17s 完成了所有请求，平均每秒完成 1230.67 个请求，失败请求数为 0。其它统计数据如下：

	Min	Max	Mean
time for request	1.34ms	5.90ms	2.99ms
time for connect	3.72ms	13.45ms	9.96ms
time to 1st byte	6.71ms	17.98ms	11.35ms

2. 运行测试 2 时平均传输流量速率为 127.18M/s。

3. 测试 3 总计在 296.02s 完成了所有请求，平均每秒完成 337.5 个请求，失败请求数为 2（发生了超时，响应 500 状态码）。反向代理产生的延迟和直接请求源站并没有显著差别，反向代理产生的损耗可以忽略不计，统计数据如下：

	Min	Max	Mean
time for request	85.37ms	1756.99ms	232.54ms
time for connect	14.86ms	39.24ms	19.90ms
time to 1st byte	156.38ms	703.48ms	254.65ms

从上述数据可以看出，本网关具有较强的并发性能，且能够高效处理存在 I/O 阻塞的异步事件。与 Nginx 和 Apache 的性能比较起来，在流量吞吐能力上与这两者差别不大；在处理 I/O 阻塞严重的异步事件时性能也没有太大差别；但是在应对大规模并发请求时，本网关性能明显优于 Apache 但相比 Nginx 稍差。

5.2 火焰图

5.2.1 分析内容

火焰图（Flame Graph）是一种用来分析程序内部调用开销的方法[19]。它通过可视化地展示各个事件占用的 CPU 时间比例来帮助我们分析程序的性能瓶颈，能够帮助我们捕捉到压力测试下可能发现不了的问题。

火焰图中 y 轴表示调用栈，每一层都是一个函数。调用栈越深，火焰就越高，顶部就是正在执行的函数，下方都是它的父函数。x 轴表示抽样数，如果一个函数在 x 轴占据的宽度越宽，就表示它被抽到的次数多，即执行的时间长。我们要寻找是否存在某个顶层调用宽度过大，这可能是我们网关存在的潜在的性能瓶颈。

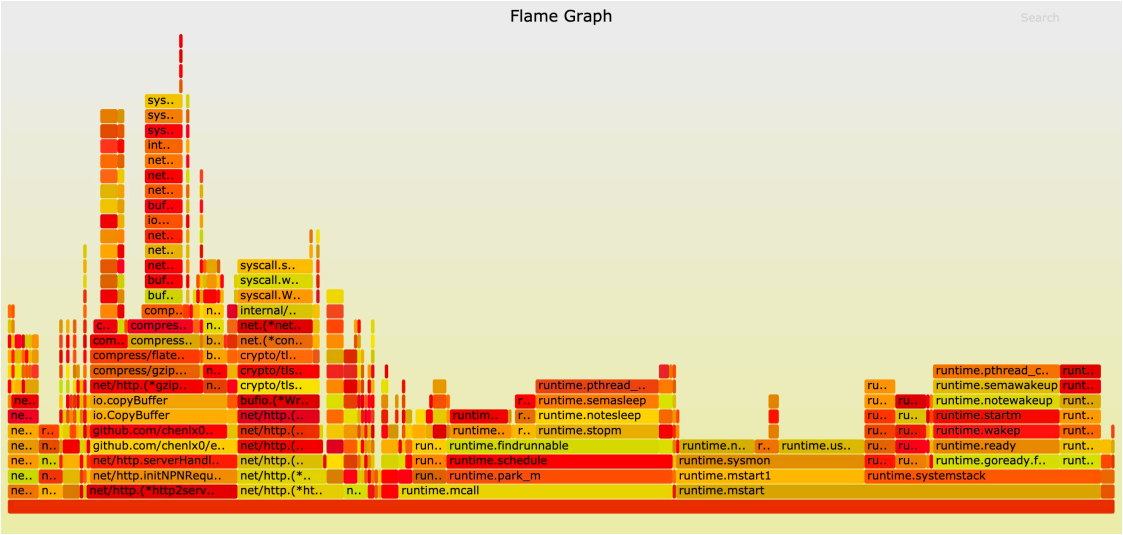
5.2.2 方案设计

Go 语言内置了 pprof 包用于监测性能，配合一个使用 Perl 语言编写的开源火焰图可视化工具 FlameGraph 即可对运行中的程序绘制火焰图。

要使用 pprof 包，需要在网关内插入一段代码，在一个新的端口打开一个 Web Server。FlameGraph 能够实时通过这个端口拉取各个调用的使用状况，并取得一段时间内的平均值（测试中是 10s）。最后根据这些数据绘制出火焰图。实际测试时通过 nghttp2 工具模拟了线上环境。

5.2.3 结果分析

对本网关绘制的火焰图如下：



从图中可以看出，各调用之间分布较为均匀。runtime 开头的调用多为 Go 底层对协程上下文切换等操作的调用，除了这之外，占用资源较多的调用为网关监听端口进行数据拷贝的调用和 HTTP/2 数据帧写入和加密的调用。总体来说没有哪一个调用过多占用了 CPU 资源，调用比例较为合理，符合本设计的最初的设想。

第六章　总结与展望

在当前服务端运维开发中,对性能和各个方面的要求越来越高。原有的HTTP/1.1协议和IPv4协议已经无法满足要求，升级到HTTP/2协议和IPv6协议已经刻不容缓。通过反向代理技术将旧有的应用升级协议是一种常用手段，但目前常用的反向代理服务器Apache HTTP Server具有并发性能较差的问题，而Nginx具有跨平台能力较差的问题。

本文设计了一个基于IPv6和IPv4双栈环境的HTTP/2反向代理网关，能够帮助用户以较低的成本将原有的IPv4环境下HTTP/1.1的应用通过本网关以反向代理技术升级支持IPv6和HTTP/2协议。

本文分析了Go语言的技术特点和其并发模型，阐明了通过Go语言开发网关的技术优势。接着通过对RFC标准的解读，分析了HTTP/1.1升级HTTP/2的方法，根据目前主流浏览器客户端的实现针对性地添加了HTTP/1.1到HTTP/2的协商方法，添加了对WebSocket协议本身的支持和基于TLSv1.2协议的HTTPS支持。

本设计还提供了两种负载均衡策略，能够帮助原有的应用在不同的业务场景下进行横向扩容。一种是基于加权轮询算法的负载均衡策略，适合在高并发环境下作为负载均衡策略；另一种是基于一致性哈希算法的，适合在用户和服务器存在cookie等持久交互时的负载均衡策略。

综上所述，本文设计的这一网关适合作为当前多种协议共存当前多种协议共存环境下过渡升级新协议的服务端应用。

但它依然有需要改进的地方：

- 直接使用Go内置已经实现的TLSv1.2加密方法固然方便，但效率较低，在大流量情况下容易出现瓶颈。最好能够自己实现最新的TLSv1.3加密方法，能够大幅度提高加密效率。
- 缺少对源站的实时监控机制，源站崩溃时不容易发现问题。
- 缺少对客户端请求的过滤机制，遭遇恶意请求容易崩溃。

致谢

大学四年转瞬即逝，四年以来在中国地质大学（武汉），结交了许多老师和同学，得到了很多帮助，让我从一个懵懂的高中生成长为了一个合格的本科生。

感谢我的两位导师，陈伟涛老师和张峰老师，给我的毕业论文和研究方向给了非常多有价值的意见和指导。同时感谢中国地质大学（武汉）网络中心和帅赟老师，给了我在大学生涯中以大量的计算机软件开发的相关实践机会和资源。

感谢中国地质大学（武汉）计算机学院，给了我们良好的学习和生活环境，感谢每一位辛勤付出的老师和职工。

感谢大学期间实习的两个单位，一个是大三暑假实习的武汉微派网络有限公司，让我接触到了 Go 语言及相关的技术；另一个是毕业实习的单位，网易（杭州）网络有限公司网络管理部 CDN 组。使我了解了相关领域业界的发展方向，提高了自己的相关技能。

最后感谢我的父母，是他们让我有一个温暖的家庭，让我无忧无虑地完成了十几年的学业。感谢大学期间帮助我的各位同学，特别是信工学院 08 级的学长吴永城，无私地帮助和引导我，带我走进了计算机行业的大门。还有姜瑞，夏丁，郑健等同学，感谢你们与我走过了一个最有价值的本科生生涯。

参考文献

[1] MARQUES D B C. Learning from HTTP/2 encrypted traffic: a machine learning-based analysis tool[J], 2018.

[2] SCHWARTZ M, MACHULAK M. Proxy[G] . Securing the Perimeter. [S.l.] : Springer, 2018 : 205 – 229.

[3] DONOVAN A A, KERNIGHAN B W. The Go programming language[M]. [S.l.] : Addison-Wesley Professional, 2015.

[4] DESHPANDE N, SPONSLER E, WEISS N. Analysis of the Go runtime scheduler[J]. URl: http://www. cs. columbia. edu/~ aho/cs6998/reports/12-12-11_DeshpandeSponslerWeiss_GO. pdf (visited on 2016-12-19), 2012.

[5] DARWISH N R, ABDELWAHAB I M. Impact of implementing http/2 in web services[J]. International Journal of Computer Applications, 2016, 147(9).

[6] HTTPARCHIVE. The number of resources requested by the page.[R]. .

[7] FIELDING R, GETTYS J, MOGUL J, et al. Hypertext transfer protocol–HTTP/1.1, 1999[J]. RFC2616, 2006.

[8] de SAXCÉ H, OPRESCU I, CHEN Y. Is HTTP/2 really faster than HTTP/1.1?[C] . 2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). 2015 : 293 – 299.

[9] BELSHE M, THOMSON M, PEON R. Hypertext transfer protocol version 2 (http/2)[J], 2015.

[10] PEON R, RUELLAN H. Hpack: Header compression for http/2[R]. 2015.

[11] FIELDING R, RESCHKE J. Hypertext transfer protocol (HTTP/1.1): Semantics and content[R]. 2014.

[12] FETTE I, MELNIKOV A. The websocket protocol (rfc 6455)[J]. Internet Engineering Task Force, 2011.

[13] RESCORLA E. Http over tls[R]. 2000.

[14] 郑光勇, 尹军, 朱贤友. 用反向代理技术保护 Web 服务器的实现 [J]. 计算机安全, 2010, 5 : 30 – 32.

[15] 夏斌, 杜守国. 云服务平台负载均衡器的网络设计与实践 [J]. 计算机应用与软件, 2014, 31(8) : 121 – 125.

[16] 杜晋芳, 徐胜国. 一种动态 Nginx 负载均衡算法 [C] . 第十届中国通信学会学术年会. 2014 : 258 – 262.

[17] KARGER D, LEHMAN E, LEIGHTON T, et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web[C] . STOC : Vol 97. 1997 : 654 – 663.

[18] 伍佑明, 杨国良, 丁圣勇. IPv6 技术及其在移动互联网中的应用 [J]. 电信科学, 2009, 25(6) : 18 – 22.

[19] GREGG B. Visualizing Performance with Flame Graphs[J], 2017.