HTTP 代理服务器的设计与实现

摘要：随着计算机使用的全面普及，网络的作用也越来越重要，互联网逐渐成为人们生产生活不可或缺的一部分，但目前互联网依然存在包括网络连接不安全、IPv4 地址已经被分配完、同时连接人数过多导致服务器负荷大和互联网区域限制等问题。所以代理服务器就被作为一种应用手段用于解决以上问题，可以通过部署一个代理服务器来实现突破 IP 访问限制、有效加强网络访问的效率、服务器之间负载均衡、用户权限验证以及自我保护等功能。虽然代理服务器已经是一个发展时间很长的技术了，但现在它依然在互联网中发挥着它的重要作用，并且随着互联网技术的发展持续进步。

本文根据现在互联网所面临的这些问题，设计并实现了一种 HTTP 代理服务器，其中第一部分对现在网络环境及代理服务器分类和发展现状进行了分析。第二部分对基于 HTTP 协议的代理服务器的工作原理和设计结构进行了详细介绍。第三部分介绍了如何利用 C++编程语言在 Linux 平台实现一个多线程的高性能的 HTTP 代理服务器，包括基本网络代理功能、内容过滤、权限控制和简单的反向代理等功能。第四部分为系统测试过程及结果展示。

关键词：HTTP 代理服务器；Linux；C++

Design And Implementation Of HTTP Proxy Server

Abstract: With the widespread use of computers, the role of the network is becoming more and more important. The Internet has gradually become an indispensable part of people 's production and life. However, the Internet still exists including insecure network connections, IPv4 addresses have been allocated, and the number of people connected at the same time, too much causes problems such as heavy server load and Internet zone restrictions. Therefore, the proxy server is used as an application to solve the above problems. A proxy server can be deployed to break through IP access restrictions, effectively enhance the efficiency of network access, load balancing between servers, user rights verification and self-protection. Although the proxy server has been a technology that has been developing for a long time, it still plays an important role in the Internet, and continues to progress with the development of Internet technology.

This article is based on the problems faced by the current Internet, designed and implemented a HTTP proxy server. The first part analyzes the current network environment and the classification and development status of proxy servers. The second part describes how to use the C++ programming language to implement a multi-threaded high-performance HTTP proxy server on the Linux platform, including basic network proxy functions, content filtering, permission control, and simple reverse proxy functions. The fourth part is the system test process and results display.

Key words: HTTP Proxy Server; Linux; C++

目 录

第1章 绪 论

1.1 研究背景与意义

互联网（Internet），又称国际网络，这是一个由无数个服务器相互连接而成的庞大网络体系。随着现代网络基础设施的不断完善，以及人们可连接网络设备的不断升级，人们对互联网的需求也越来越大。虽然互联网产业飞速发展，但是使用人数的快速增长，也使得现如今的网络环境面临着可用 IP 不够、区域限制、网络安全威胁、服务器负荷大等问题。

使用代理服务器把公共网络与私有网络之间连接起来，让处于局域网的用户通过代理服务器访问服务器的内容，有效解决上述问题。因为每个局域网内的用户都使用同一个代理服务器来访问外部网络，这样就有效解决了如今 IPv4 资源严重不足的问题。同时代理服务器还可以充当防火墙作用，对经过代理服务器的数据进行检查过滤，保证了连接代理服务器用户的网络安全。使用反向代理服务器来代替源服务器接收请求，使客户端请求分流，就能解决源服务器负荷过大的问题。

1.2 国内外研究现状

由于代理服务器技术自互联网诞生以来便有许多人开始研究，所以目前网络上包含着许多功能完善且种类丰富的代理服务器。下面将从代理服务器的分类和主要作用两个方法对现在代理服务器的发展进行说明。

1.2.1 代理服务器的种类划分

按代理服务器使用的协议类型进行划分，可以将现在主流的代理服务器大致分为 HTTP（HyperText Transfer Protocol）代理服务器、SOCKS 代理服务器和一些其他类型的代理服务器三类。HTTP 代理服务器工作在 OSI 模型的应用层，以 HTTP 的方式实现任意基于 TCP 的应用层协议代理，被应用于代理 WEB 浏览器和 HTTP 服务器之间的数据，使用的范围非常广泛，现在人们主要使用它来突破不同网络之间的浏览限制，可以借由 HTTP 代理服务器的网路来浏览到原来网络所不能访问的内容，同时还能增加访问的数据在传输过程中的安全性。SOCKS 代理服务器则是使用的 SOCKS 协议交换数据，SOCKS 协议是根据 Internet 跟踪协议标准，作用于网络防火墙，是会话层与会话层之间安全服务方案，并且可以对 OSI 模型应用层的数据进行过滤，即使改变应用程序协议，

也不会影响 SOCKS 协议运行。正是因为 SOCKS 协议这种只单纯传递数据包，可以过滤应用层协议的机制，所以 SOCKS 代理服务器不仅仅可以像 HTTP 代理服务器那样代理普通网页请求，而且还可以适用于更加复杂的网络环境，通过 SOCKS 代理客户端和代理服务器之间传输数据的加密和解密可以使连接更加安全，防止被监听。

1.2.2 代理服务器的作用

1、提高访问速度

每次把客户端访问的数据存放在硬盘里，这样当客户再次请求相同站点的相同数据时，就可以直接从局域网服务器中读取，比从远端服务器再次获取资源的速度会快很多。在这个过程中代理服务器起到了缓存的作用，对于客户端访问非常频繁的站点，代理服务的优势能得到显著体现。不过随着近几年互联网访问速度不断地提升，所以现在使用代理服务器缓存机制来提高访问速度的作用已经没有以前那么重要了。

2、作为网络防火墙

因为使用代理服务器的用户必须通过代理服务器所在主机才能与外部网络进行数据交换，这样代理服务器就可以充当网络防火墙的作用，用于检查连接双方的域名和 IP 是否属于黑名单，还可以对双方交换的协议内容进行检查，对存在风险的内容进行过滤处理。

3、访问一些不能访问的站点

现在的互联网结构复杂，不同国家或地区之间存在区域限制，导致很多时候我们并不能正确获取到想要的数据。这个时候就可以通过在目标网络环境下建立代理服务器，同时代理服务器又能访问到互联网，这样就能通过代理服务器绕开区域限制，以获取到我们需要的数据。同时这也是现代代理服务器最重要的作用之一。

4、隐藏 IP 信息

通过代理服务器来访问外部资源时，目标服务器得到的访问 IP 信息只是代理服务器的 IP，这样代理服务器下面的客户端就达到了隐藏自身真实 IP 的目的。使用这种方法也能达到减少 IP 地址使用量的目的。

第 2 章　HTTP 代理服务器的概述

2.1 定义

代理服务器是一种工作环境介于客户端和服务器之间的服务器，HTTP 代理服务器就属于其中的一类，是介于 Web 浏览器与 HTTP 服务器之间的特殊服务器。简单来说代理服务器就是一个网络信息中转站，对于连接到它的客户端而言，它属于服务端；对于它要连接的服务端而言，它属于客户端。它就是局域网与外部互联网之间的中间代理机构，用来实现对两个不同网络之间信息转发和内容控制等功能。

HTTP 代理服务器就是一种基于 HTTP 协议运行的代理服务器类型。根据 RFC 7230 - HTTP/1.1: Message Syntax and Routing[14]中对 Intermediaries“中间人”的定义，协议允许使用中介程序通过连接链的方式来完成请求，并将连接链的中间人分为：代理，网关和通道三种方式。在某些情况，中间人还可以根据请求内容的不同使用不同的方式来连接服务器。

图 2-1 中间人在网络中的位置

图 2-1 展示的是三个处于客户端和原始服务器之间的中间人，从客户端发起的请求以及从原始服务器发回的响应信息都将分别经历这个连接链中的四个不同的连接，最终被传送到目的地。虽然图中展示的连接链是线性的，但实际连接时，高级代理服务器是可以进行多个连接，同时代理到不同的位置。比如 B 代理服务器不仅可以接收来自 A 代理服务器的请求还可以接收其他代理服务器传来的请求信息，同时 B 不仅可以把来自 A 的请求信息发往 C 代理服务器还可以发往其他不同的服务器。而且 B 代理服务器还可以基于后一级连接的负载情况来动态判断应该把请求发往负载较低的服务器。

目前 HTTP 代理存在两种形式，下面对这两种形式进行简单介绍：

第一种是前面所提到的 RFC 7230 - HTTP/1.1 标准文档中描述的普通代理，这种代理服务器所要扮演的是中间人这个角色，对于连接到它的用户来说，它属于一个服务器；

而对于它要要连接的服务端来说，它又是客户端。主要负责在客户端和原始服务器之间来回交换 HTTP 报文。

另一种则是 Tunneling TCP based protocols through Web proxy servers（通过 Web 代理服务器用隧道方式传输基于 TCP 的协议）的隧道代理方式。这种代理使用的是 HTTP 协议中的内容部分（Body）来完成数据交换，以 HTTP 协议方式来实现基于 TCP 连接的应用层协议代理。隧道代理使用 HTTP 协议中的 CONNECT 方法来发送请求以建立连接，而 CONNECT 方法在 RFC 2616 - HTTP/1.1 文档中并没有被定义，直到后来发布的 RFC 7231 - HTTP/1.1: Semantics and Content[15]才增加了对 CONNECT 方法的描述，这也给 HTTPS（Hypertext Transfer Protocol Secure）安全链接的建立提供了协议支持。

2.2 分类

按 HTTP 代理服务器工作原理和使用方式的不同，可以把 HTTP 代理服务器分为前向代理服务器、反向代理服务器和透明代理服务器。

1、正向代理服务器

正向代理（Forward Proxy）是指客户端需要通过在浏览器上对目标代理服务器进行配置，包括 HTTP 协议发往的 IP 地址或域名、端口号、用户名和密码等。开启正向代理后浏览器就会按照配置的内容，把 HTTP 协议请求发往目标代理服务器，然后代理服务器再将请求转发给目标服务器，待到目标服务器将响应内容返回，代理服务器就把响应内容发给客户端的浏览器。这种形式的代理服务器最大的特点就是客户知道代理服务器的存在，并主动通过代理服务器获取远端信息。图 2-2 展示的是正向 HTTP 代理服务器的工作结构图。

2.、反向代理服务器

反向代理（Reverse Proxy）则是使 HTTP 代理服务器伪装成普通服务器来工作，就工作基本原理而言与正向代理服务器基本一样，不同点在于使用反向代理服务器时，浏览器不需要知道目标服务器地址，也不用对客户端进行任何配置。浏览器只需要直接向反向代理服务器请求资源，而反向代理服务器则通过自身配置，在网络上寻找最适合的资源，然后返回给浏览器。反向代理服务器的工作结构如图 2-3 所示。

从图 2-3 中可以看到用户想要通过 www.example.com 域名获取到一个叫 file 的资源文件，于是用户就访问 http://www.example.com/file 这个地址，但实际上在 www.example.com 上并没有 file 这个资源文件，于是 www.example.com 就从其后端服务器中寻找 file 资源，然后将 file 资源作为自己的内容返回给用户。这对于用户来说是完

全不知情的，就像是直接从 www.example.com 这个服务器上获取到的资源。其中 www.example.com 域名所对应的服务器便是设置了反向代理功能的反向代理服务器。

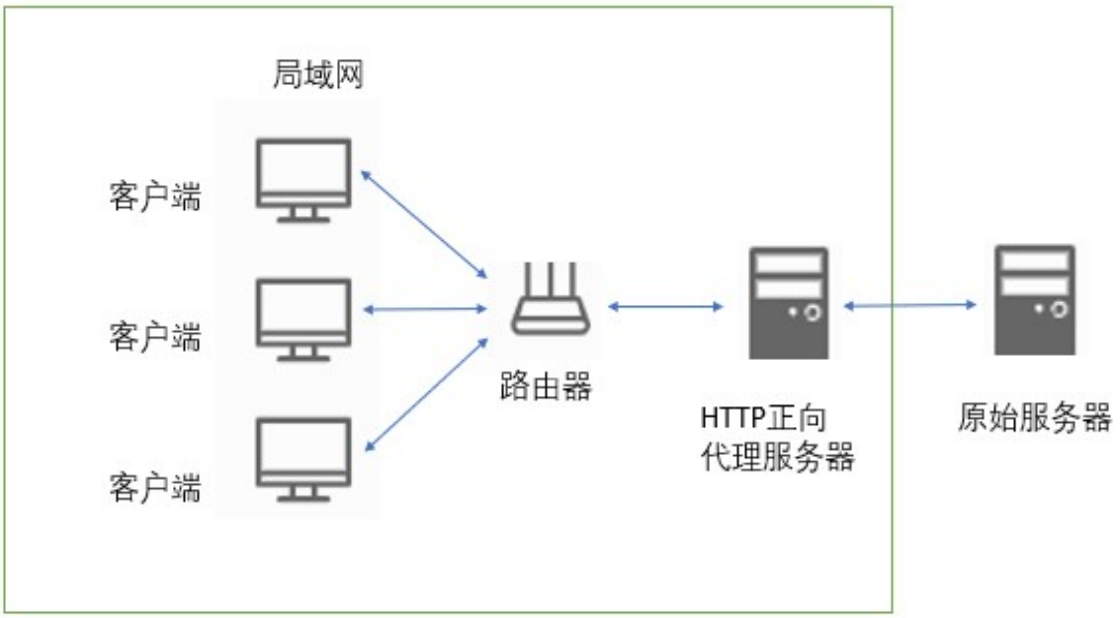

图 2-2 正向代理服务器结构图

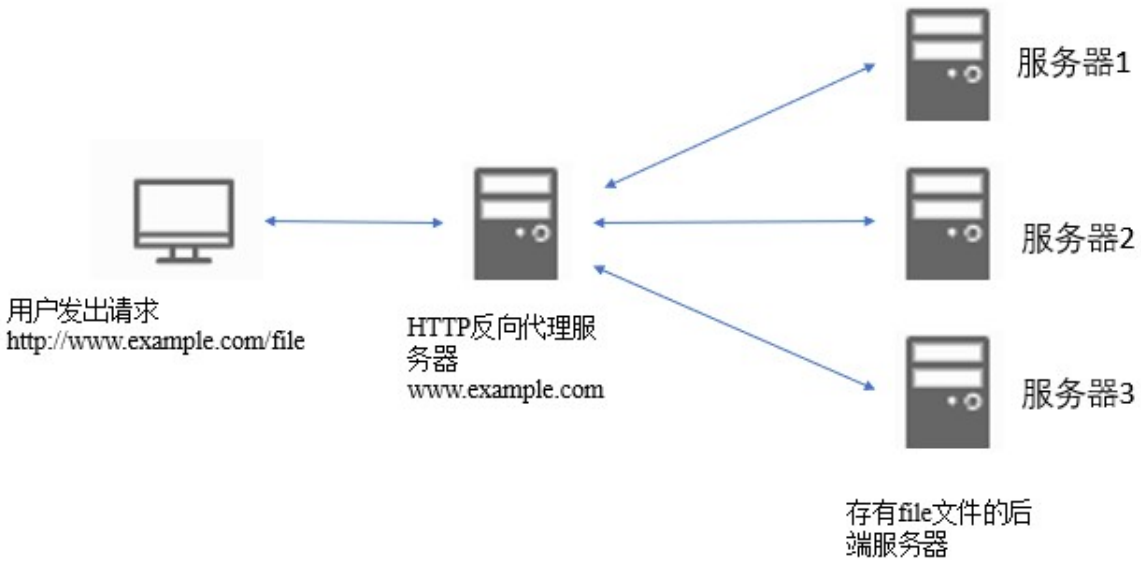

图 2-3 反向代理服务器工作结构图

3、透明代理服务器

透明代理（Transparent Proxy）是指将代理服务器搭建在客户端路由的必经路径上，通过拦截用户的请求报文，并作为实际的请求方，向服务器发送请求，最后再把服务器返回的内容返回给用户。透明代理与前两种代理方式的最大不同就是其相对于用户而言是完全隐藏的，用户不知道代理服务器的存在。透明代理多被网络提供商用于解决 IP 地址不足的问题。图 2-4 展示的是透明代理服务器的工作结构图。

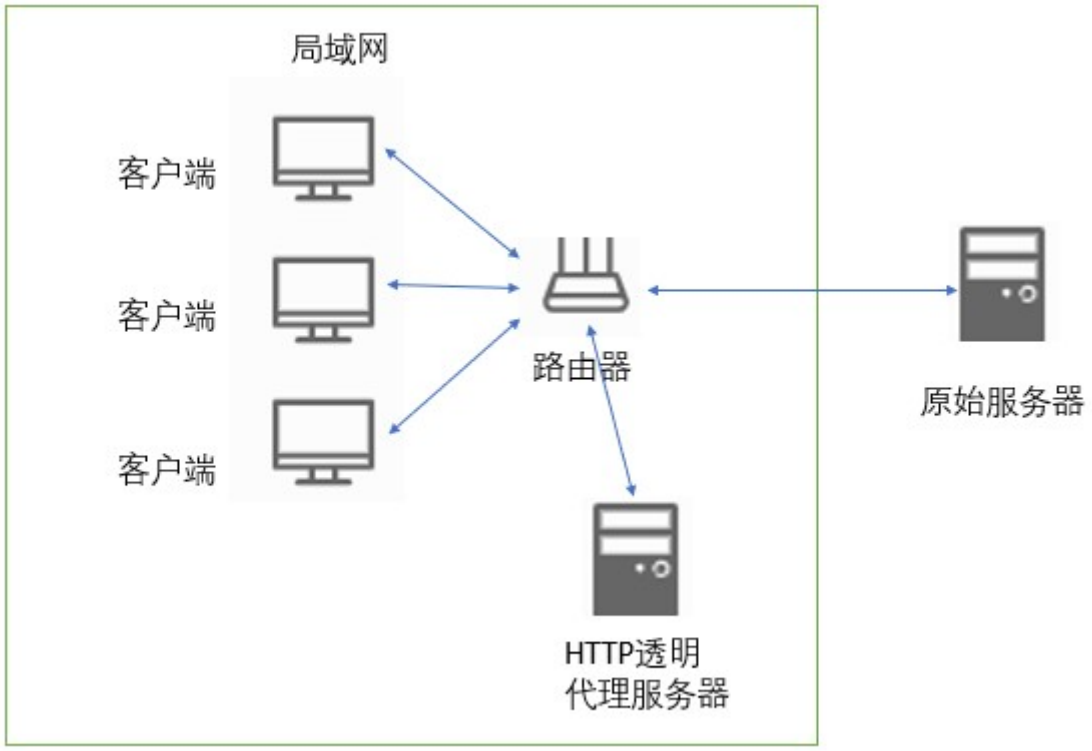

图 2-4 透明代理服务器工作结构图

2.3 基本工作流程

HTTP 代理服务器作为一种工作在客户端浏览器与原始服务器之间的特殊服务器，不管是那种类型的 HTTP 代理服务器都遵循着一个相同的基本工作流程，下面对 HTTP 代理服务器的基本工作流程进行介绍：

1、配置阶段

这个阶段处于代理服务器刚启动时，用于对代理服务器的基本工作方式进行配置，例如代理服务器需要监听的网口和端口，以及使用正向代理工作还是反向代理工作等。虽然有些简单的代理服务器使用缺省的方式省略了这一步，但对于一个功能完善的 HTTP 代理服务器而言这是必不可少的。

2、等待用户请求阶段

当服务器完成基本配置后，服务器就根据配置内容开始等待用户对自己开放的端口发起 TCP 连接，在 TCP 三次握手确认连接后进入下一个阶段，对于任务分配模型比较高级的代理服务器的进入下一阶段的同时，依然能继续监听下一个用户的请求。

3、处理请求阶段

在收到用户发送给代理服务器的请求内容后，代理服务器第一步要做的就是解析请求的内容。根据 HTTP 协议的定义将请求内容解析为请求方法、目标 URL、HTTP 协议的版本和请求头这几部分，然后根据代理服务器运行的方式对请求内容进行处理，最后把处理过的请求内容发给上层服务器。

4、处理响应阶段

在代理服务器将请求内容发给上层服务器后，就会收到来自上层服务器的响应协议内容。一般代理服务器也会对响应内容进行解析并处理，然后把处理过的内容返回给客户端的浏览器。这样代理服务器便是完成了一次最基本的代理请求。

2.4 网络模型

每一个需要连接网络的程序都需要一个设计成熟的网络，代理服务器也不例外。网络模型的优劣直接决定的一个代理服务器的工作效率以及稳定性等问题。下面对现在主流的网络模型进行介绍：

1、迭代模型

这种模型早在 20 世纪 50 年代便已经问世，属于最原始的模型之一。使用迭代模型运行的程序只有一个线程，网络信号接收以及业务处理都在一起进行，并且每次只能对一个用户的请求报文进行处理，只有等待上一个用户的请求报文处理完成并断开连接后才能开始对下一个的用户的请求报文进行处理。这种模型的代理服务器经常被开发者用于 DEMO 的开发中，演示预计的功能是否具有可行性。图 2-5 对迭代模型的工作方式进行了简单描述。

2、并发模型

并发属于对进程资源的较高级使用，它可以把一些需要等待响应的程序操作闲置起来，腾出更多的 CPU 使用空间给一些需要及时处理的程序，当上一个等待的程序响应后才对其进行处理，这样便能实现同时处理请求的目的。并发模型便是使用这种原理，当代理服务器程序收到来自客户端的请求时，便开启一个额外的子线程用来处理请求，通过新建多个子线程的方式来同时处理多个用户请求。图 2-6 对并发模型的代理服务器

进行了描述。

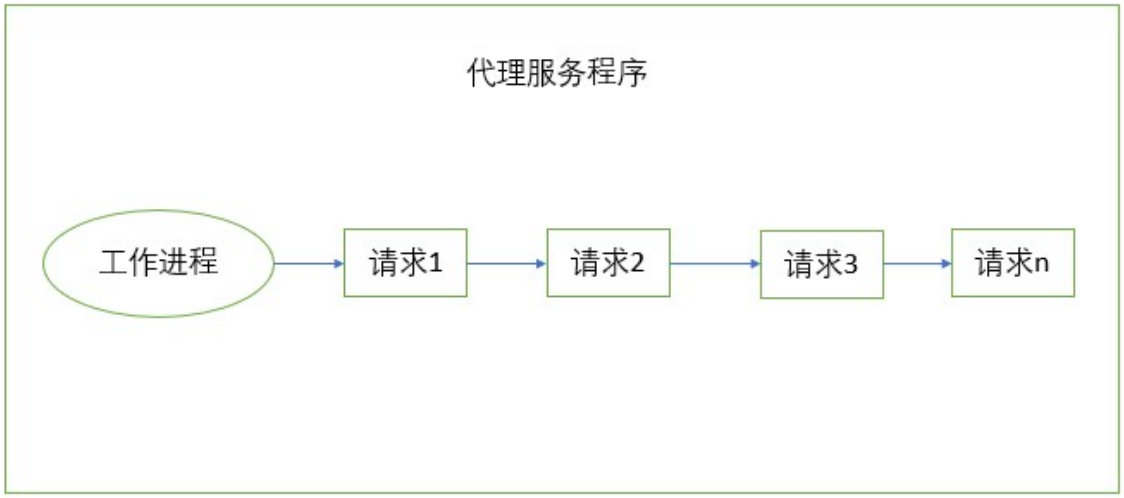

图 2-5 迭代模型

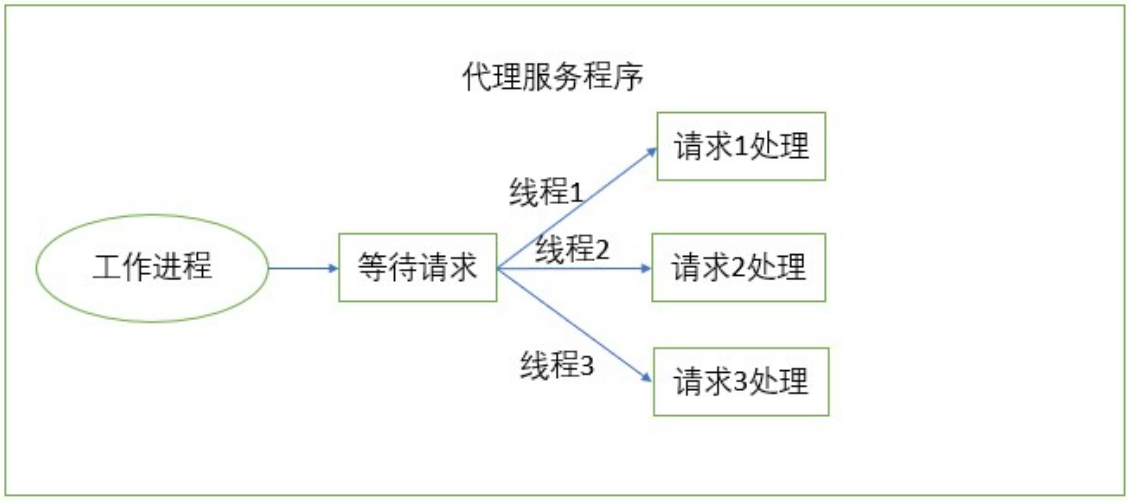

图 2-6 并发模型

3.分发模型

分发模型是指代理服务器主线程负责分发任务，包括来自用户的请求内容和来自服务器的响应内容，将这些任务分发给单个的线程进行处理。这样就可以使网络服务处理与程序逻辑处理分开来进行，代理服务器就能同时处理大量的用户。图 2-7 展示了分发模型的结构示意图。

4.线程池模型

线程池就是在程序开始运行时就按照配置定义创建一定数量空闲的线程，这些线程集合就被称为线程池，在程序需要运行一个任务时，就在线程池的空闲线程中选择一个来运行任务，任务运行完毕后，属于线程池的线程不会像普通线程一样被系统回收，而是再次被放回到空闲线程池中等待下一次任务调用。使用线程池模型时不会因为频繁的去进行系统调用来申请新线程空间和销毁线程而过渡消耗系统资源，也不会因为过渡切换线程而错误，致使系统意外崩溃，所以这可以很好的提高执行任务时的系统运行效率和系统稳定性。图 2-8 对线程池模型的工作流程进行了描述。

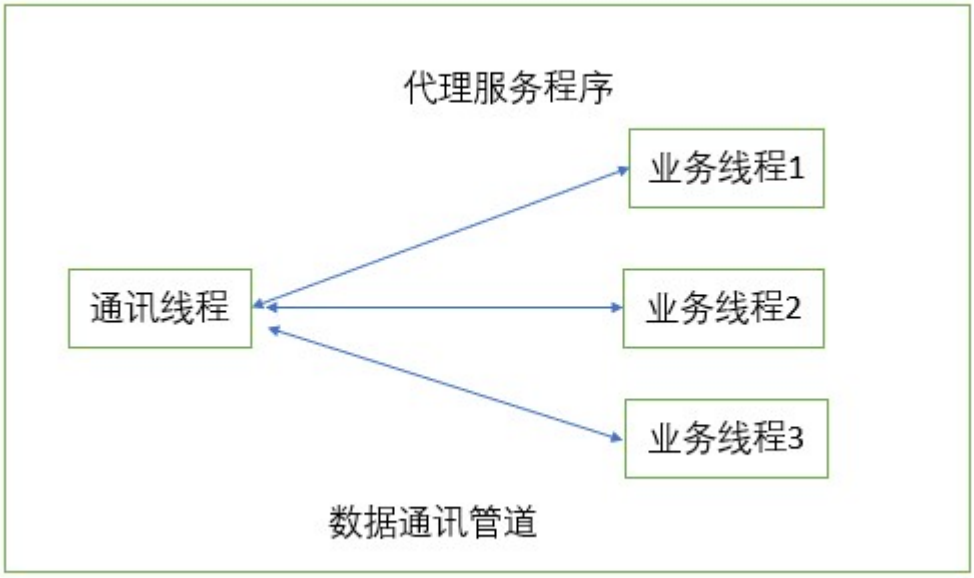

图 2-7 分发模型

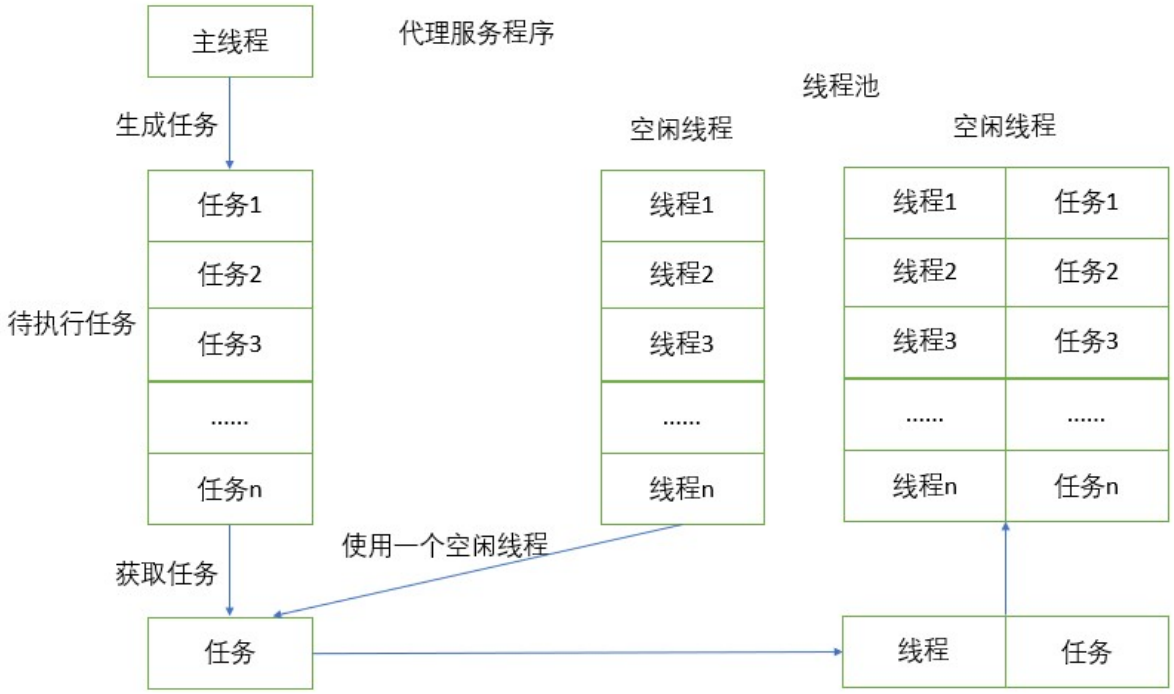
主线程
代理服务程序
线程池
生成任务
空闲线程
空闲线程
任务1
任务2
任务3
.....
任务n
待执行任务
线程1
线程2
线程3
.....
线程n
线程1
任务1
线程2
任务2
线程3
任务3
.....
.....
线程n
任务n
使用一个空闲线程
获取任务
任务
线程
任务

图 2-8 线程池模型

第 3 章　HTTP 代理服务器的设计与实现

3.1 方案选择

3.1.1　设计平台与编程库的选择

下面列出几个设计初期拟定的施行方案：

1、在 Windows 平台使用 C++编程语言进行编程，网络编程库使用 Windows 下的 C 语言的库函数，并设计可以直观表达各项功能的界面，通过 GUI（Graphical User Interface）面板来编辑基本参数，控制的各项功能。

2、在 Linux 平台上进行 C++编程，使用可跨平台运行的 Qt 库来实现代理服务器的基本功能。同时使用 Qt 库制作一个控制界面，通过 GUI 完成代理服务器人机交互。

3、在 Linux 平台上进行 C++编程，使用基于 Linux 的原生 C/C++网络编程库来开发代理服务器的各项功能。使用配置文件和运行命令的方式来完成代理服务器的控制。

4、在 Linux 上进行编程，使用可跨平台的 Asio 网络库来开发服务器的网络连接部分，程序的其他重要模块也使用基于 Boost 的库进行开发，不制作 GUI 界面，所有服务器控制都基于配置文件和控制台命令来完成，花费更多时间来提升服务器的稳定性和安全性，实现更多有用的功能。

通过查阅相关资料，对现在网络上普遍流行的代理服务器的设计结构进行学习，下面将对上面提出的几个拟定方案进行选择。

首先排除第一个方案，因为在 Windows 上进行 C++服务器类编程开发都比较困难，开发周期会大大加长，而且基于 Windows 开发的服务器性能和安全性方面都不是很高。第二个方案使用 Linux 作为编程和运行平台，因为 Linux 对服务器性能和安全方面的保护是非常到位的，其次 Linux 下的 C++编程更加方便可以使用的资源更加多，而且 QT 库也是非常优秀的跨平台 C++库，所以这个方案相对与方案一有很大提升。但是往往决定一款服务器程序性能好坏的因素，GUI 界面的优劣并不在其中，而是其长期运行的稳定性，处理客户请求的效率和并发性等因素。所以设计代理服务器程序更应该着重考虑它在后台的运行情况和资源消耗。方案三就是考虑到了这一点，舍弃了复杂的 GUI 界面设计，花更多时间来提升后台性能，开发更多实用的功能。方案四是在方案三的基础上进一步提高，使用先进的 Boost 库来开发程序。由于 C++官方标准库中没有对网络部分

进行补充，所以在编写 C++程序的网络连接部分时总是要使用最基本的 C 语言库中的语义，这是相当复杂且很不安全的，所以 Boost 库中有 Asio 对 C++网络编程部分进行了第三方补充。Asio 库的使用不仅简单容易上手，而且稳定性非常高。Boost 库中还存有其他方便好用的库，也可以添加到代理服务器的编程中，以降低开发难度。综上，采用方案四作为程序开发的最终方案。

下面列出最终采用方案的系统构成为：

开发平台：Linux

编程语言：C++

编译器：GCC

编辑器：qtcreator

使用的第三方库：Boost

程序类型：后台服务程序

根据以上预设计方案，想要完整开发一款 HTTP 代理服务器，其中需要涉及的相关技术包括：Linux 系统使用方法、C++的新特性、Boost 编程库的使用、HTTP 协议的基本内容、HTTP 代理服务器的运行流程等。

3.1.2 网络模型的选择

一个高性能且运行稳定的HTTP代理服务器必须拥有一个优秀的网络模型作为程序开发的基础，下面对第二章中提到的 4 种基本网络模型进行优劣分析：

1、迭代模型

优点：设计简单，系统稳定，适合系统模型开发时，对方案可行性验证时使用。

缺点：因为迭代模型每次只能处理一个任务，当任务较多时工作效率将会非常低，所以实际开发时一般不会选择使用。

2、并发模型

优点：多线程处理，有效提高了任务处理速度。当并发量比较低时，稳定性高。

缺点：不适合大量连接，当并发量提高时，资源消耗显著提高，系统会因为过渡使用线程而崩溃甚至出现死机。

3、分发模型

优点：适合大量用户接入，将通信处理与业务逻辑区分开来，使得系统结构逻辑严密，可扩展性强。

缺点：模型复杂度高，设计难度大，并且容易因为线程逻辑而导致的内存调用错误，

使得系统崩溃。

4、线程池模型

优点：合理设置初始线程数，可以有效利用多个 CPU 资源，极大提高任务处理速度，系统稳定较高。

缺点：与分发模型类似，因为涉及到多线程资源共享问题，如果程序流程设计不当，容易导致程序内存调用错误。为了解决资源同步共享问题，使用互斥锁等线程同步机制也会浪费一些系统资源。

根据对以上 4 种网络模型的优缺点分析，最后得出使用线程池模型可以很好的设计出一个高性能且稳定的 HTTP 代理服务器。并且 Asio 网络库与线程池模型的兼容性比较良好，减少了程序开发过程中可能会遇到的复杂问题。

3.2 对相关技术的概述

3.2.1 HTTP 协议

HTTP 协议主要是用于 Web 客户端与 Web 服务器之间进行数据交换而被定义出来的。编写 HTTP 代理服务器首先需要弄明白的就是 HTTP 协议的基本结构，标准的 HTTP 协议包含请求报文与响应报文两部分内容，请求报文首先包含一个请求行，请求行中有请求的类型、请求的内容和协议版本，然后就是多行请求头，最后是请求的消息体。响应报文的结构与请求报文类似，首先是一个状态行包含响应码、响应消息和协议版本，然后是多行响应头，最后是响应的消息内容。图 3-1 和 3-2 分别展示了 HTTP 协议请求报文与响应报文的基本格式。

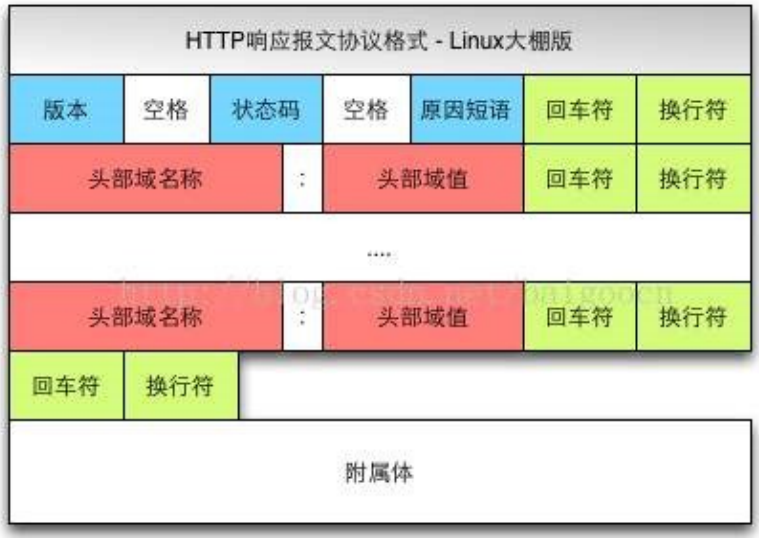

图 3-1 HTTP 协议请求报文标准格式

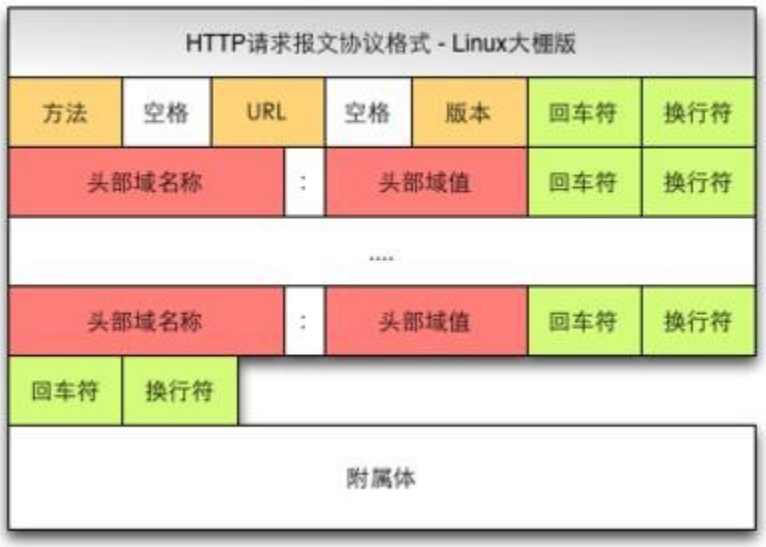

图 3-2 HTTP 协议响应报文标准格式

HTTP 协议工作时的基本流程为：首先 Web 浏览器向 Web 服务器发送包含请求内容与目标信息的请求报文，然后 Web 服务器解析请求报文内容，根据请求内容向 Web 浏览器返回响应报文。由于 HTTP 代理服务器的请求格式也符合 HTTP 协议，因此代理服务器在收到客户端请求时的第一步也是按照协议格式解析出请求的内容，以此才能判断客户端想要请求的内容以及目标服务器等重要信息。

3.2.2 Linux 操作系统

该操作系统的内核是由林纳斯·托瓦兹在 1991 年 10 月 5 日首次发布的，这个系统完全是开源自由的且非常安全可靠，所以目前世界上大部分的服务器程序都是由 Linux 系统在运行。本次设计开发所使用的系统是Linux 系统发行版中的ArchLinux 系统，HTTP 代理服务器的代码编写、运行和调试都将在这个系统中完成，代码编写完成后可直接发布到 Linux 云服务器中运行起来，这样系统便能起到代理服务器的作用。刚上手使用 Linux 系统一定要学会使用控制台和几个基本指令。Linux 的控制台是系统的灵魂所在，只要学会使用控制台，基本就可以操作系统的所有功能了。下面列出几个在编程过程中将会频繁使用的基本命令：

1、ls

ls 命令的全称是 list，也就是列出当前目录下文件的意思。ls 命令常用的参数搭配有-a 和-l。-a 列出目录下所有的文件，包含被隐藏的文件。-l 列出除了文件名以外，还列出文件权限、所有者、文件大小等更加详细的文件详细。

2、cd

cd 命令的作用是切换当前工作目录，配合 ls 命令就可以用来查看文件夹的各种文件了。

3、rm

rm 命令的全称是 remove，也就是用来删除文件的。可以使用-f 参数来强制删除文件。

4、mkdir

mkdir 命令的作用是创建文件夹。

5、touch

touch 命令可以用来创建一个文件。

3.2.3 C++编程语言

C++是一种通用程序设计语言，其使用非常广泛。目前 C++已经更新了很多版本了，最新版本为 C++20 是第六个 C++标准，而这次的设计只需要学习到 C++的第三个标准即 C++11。下面列出几个编程中常用的新特性：move 移动语义、可变参数模板、初始化列表、auto 关键字、lambda 表达式、类型获取等等。C++11 中向 C++标准库更新了很多非常实用的功能，极大的提高了 C++编程开发的效率。

3.2.4 Boost 库

Boost 库是 C++的第三方代码库，提供免费同行评议。Boost 库与 C++标准库的兼容性非常强，所以其中有很多优秀的设计都被选入使用到了 C++标准库之中。由于 C++标准库中不提供关于网络编程方面的接口，要使用网络编程就必须使用原始的 C 语言提供的库函数，这是非常不安全而且繁琐的。所以本次设计重点使用的是 Boost 库中的 Asio 库，Asio 提供了一个便携式的网络接口和底层 I/O 操作，包括套接字的封装、计时器、主机名解析、套接字流、串行端口、文件描述符等操作。这些操作在 HTTP 代理服务器的编写中都是非常有用的，不仅提高了代码的安全性，而且减少了代码的编写难度。同时程序还会用到另一个库 Program Options，这个库以键值对的方式提供了更加方便的控制台选项设计功能和配置文件读取功能。对于无界面服务器程序的开发具有很大的帮助。

1、Asio

Asio 是一个跨平台的 C++网络编程库，为 C++编程者提供了底层 I/O 程序的异步易用模型的现代 C++编程接口。大部分程序在如外部世界相连接时，无论使用文件、网络、

电线还是控制台都和网络连接非常相似，这些连接在等待数据读写时都将花费很长的时间来完成，Asio 就是为这些读写等待提供了一个统一的平台。使用 Asio 库来管理这些长时间连接的接口，可以不需要基于线程的并发模型和显性的互斥锁操作。总而言之 Asio 有如下优点：便携性、可扩展性、效率高、可基于 API 编程、容易使用、可进一步抽象。

2、ProgramOptions

这个库主要是用于管理控制台服务程序的配置文件和命令行输入，使得程序可以用键值对的方式很方便的读取配置文件和命令行输入的信息。使用 ProgramOptions 库有如下几个优点：1）程序更易编辑。声明语法简单，库自身容量小，转换过程中的变量类型都是自动处理的。2）错误报告功能强大。3）读取到的配置选项可以在任何时候被被读取。

3、Thread

这个库的作用是使程序能在同一个进程里面进行多线程操作，同时能在各个线程之间共享数据。它提供了用来管理线程的类和方法，在线程之间同步数据的功能等。虽然现在 C++标准库函数已经包含 Thread 线程库，但为了更好的参考 Boost 官方的标准文档，依然使用 Boost 里面的 Thread 库来编写多线程部分。

3.3 程序输入输出框架

HTTP 代理服务器的信息输入主要包括配置文件、控制台选项、来自客户端的请求报文和来自服务器的响应报文。配置文件和控制台属于程序基本配置类，它们都可以通过 ProgramOptions 库进行处理，从而产生可以被程序读取的键值对变量。来自客户端的请求报文在通过 Asio 网络库读取为字节流后，可以用专门处理 HTTP 请求报文的程序段进行解析，从中可以提取出客户端请求的路径和目标服务器地址等重要信息。来自上游服务器的响应报文也需要通过专门处理 HTTP 响应报文的程序段进行解析，得出本次请求的响应结果，然后再将这个结果返回给客户端的浏览器。

HTTP 代理服务器的信息输出主要包括发向客户端浏览器的响应报文和发向上游服务器的请求报文。其中发给上游服务器的请求报文是通过来自客户端请求报文解析处理后重新组装而成的，发送请求报文是为了向上游服务器请求客户端需要的内容。发送给客户端的响应报文是通过来自上游浏览器响应报文解析处理后重新组装而成的，响应报文是为了响应来自客户端的请求。

图 3-3 对 HTTP 代理服务器可能存在的输入输出内容进行了展示。

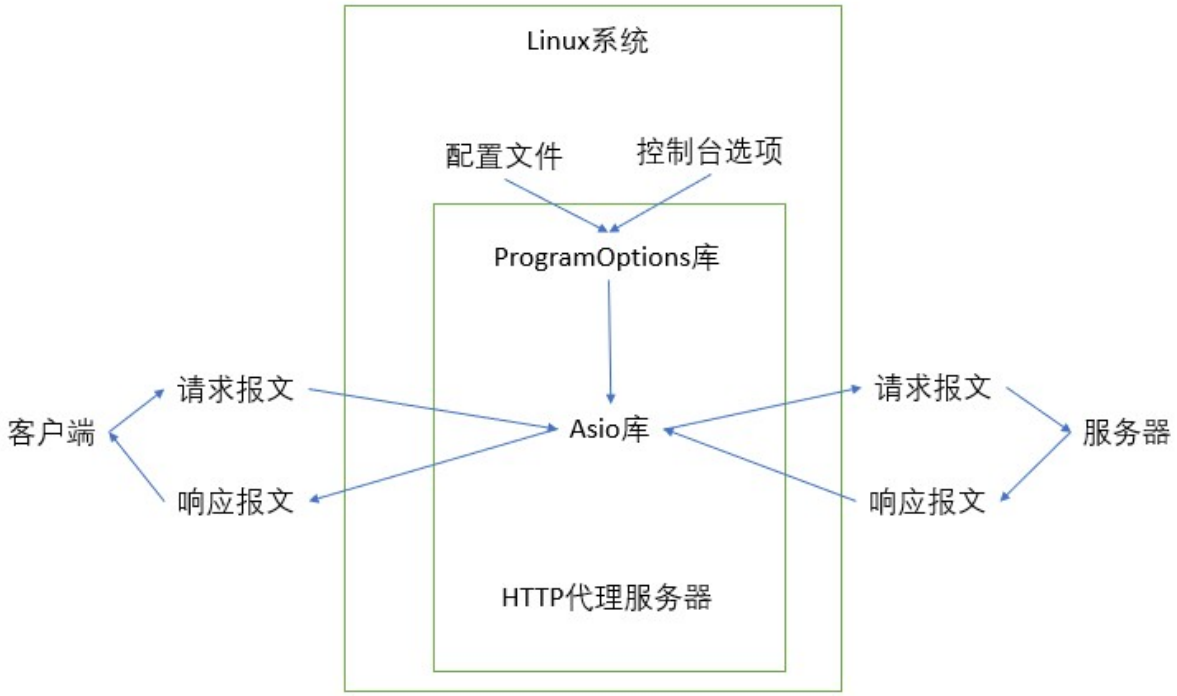

图 3-3 HTTP 代理服务器输入输出框架

3.4 程序启动流程图

图 3-4 展示的 HTTP 代理服务器从启动到开始处理客户端请求的完整流程图：

图 3-4 HTTP 代理服务器程序流程图

HTTP 代理服务器是运行在后台的服务类型程序，启动运行时需要使用控制台进行操作，也可以使用 Linux 操作系统自带的服务管理程序启动。代理服务器启动前需要安装标准对配置文件的内容进行编辑，不然程序就会因为错误读取配置文件而崩溃。使用控制台启动程序时也可以在程序后面输入控制选项来覆盖配置文件的部分内容。代理服务器开始运行后首先就会去读取控制选项的内容，然后读取配置文件的内容，根据这些输入的内容完成全局参数类的初始化。下一步程序就会按照配置内容提前向系统申请一定个数的线程，以完成线程池的初始化。最后根据配置内容启动 TCP 服务，开始监听来自客户端的请求。

3.5 客户请求处理流程图

图 3-5 描述的是一次来自客户端 HTTP 代理请求的完整处理流程：

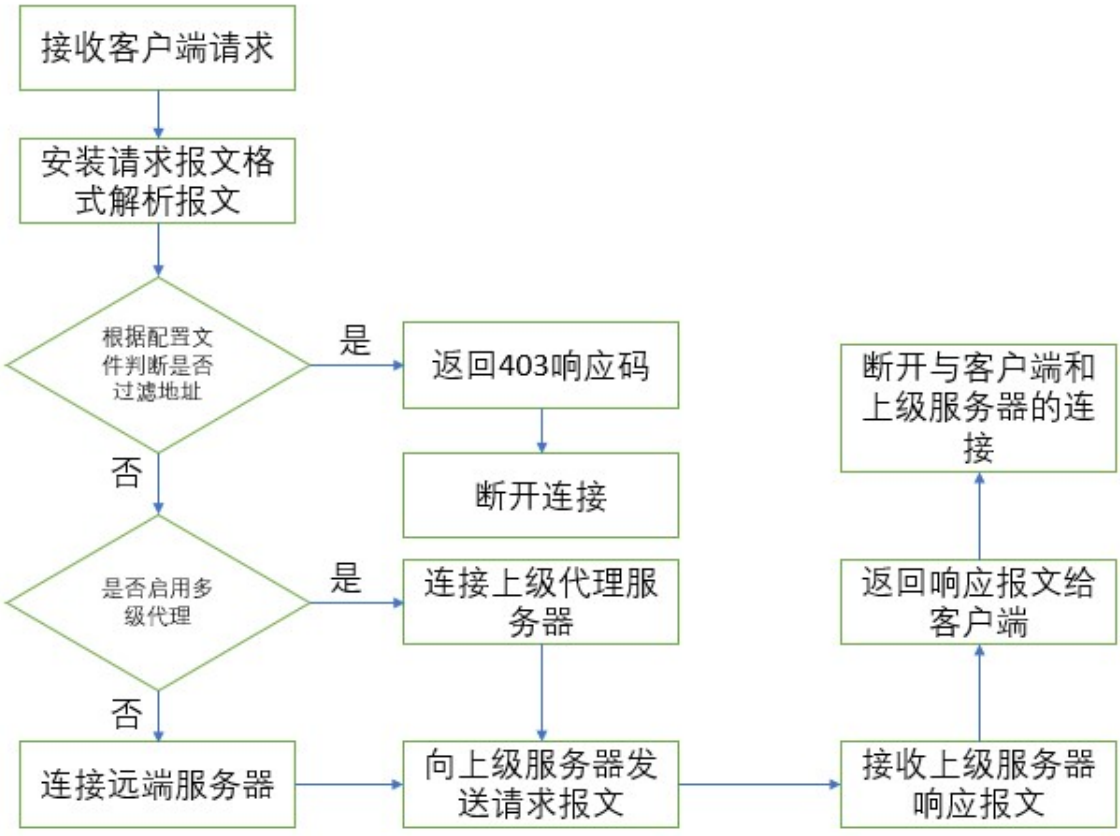

图 3-5 HTTP 代理请求处理流程图

代理服务器程序在接收到来自客户端请求报文后，首先是按照 HTTP 协议请求报文的标准格式对报文进行解析。然后读取配置文件中需要过滤的地址和 IP 与解析出来的内容进行对比得出客户端的请求是否被接收，如果请求的地址在过滤配置表中则立即返回客户端 403 响应码并退出连接，如果没有在配置表中就进入到下一步。下一步开始判

断代理服务器是否启用多级代理，即是否设置了上游代理服务器，如果启用了多级代理就连接配置中的上级代理服务器，如果没有就连接请求报文中的远端服务器。然后向处于连接中的上级代理服务器发送请求报文。在收到来自上级代理服务器的响应报文后将报文发送给客户端的浏览器。

3.6 程序设计代码详解

3.6.1 配置文件部分

配置文件包含三个部分：基本配置文件、过滤配置表、反向代理地址表。

1、基本配置文件

基本配置文件在代码中的缺省位置是“./config”，可以通过命令行输入选项的方式修改这个默认地址，后面的命令行输入选项将详细介绍修改方法。

基本配置文件包含以下配置项：

（1）port

用于保存 HTTP 代理服务器的监听端口。

（2）listen

用于保存 HTTP 代理服务器的监听地址。

（3）timeout

定义程序网络通讯连接无响应的最长等待时间，防止多长等待而消耗系统资源，单位为秒。

（4）bindsame

这是一个布尔值，包含 true 或 false 两种选项。这个配置项用于决定代理服务器在收到客户端请求报文后连接的地址，如果是 true 代理服务器就会把上级代理服务器的连接地址定义为客户端的地址，如果为 false 则不执行 bindsame 这个选项的那段程序。

（5）filter

这个配置项用于指定过滤地址配置表文件的地址，如果没有则不启用过滤。

（6）upstream

用于指定多级代理的上游代理服务器的地址，如果没有则不启用多级代理。

（7）maxthread

这个配置项用于指定 HTTP 代理服务器启动时默认的线程池中的最大可用线程数。

（8）reversepathfile

指定反向代理地址表文件的位置。

（9）reversebaseurl

指定反向代理的基地址。

下面对 proxy.cfg 基本配置文件默认内容进行展示：

#代理服务器监听的端口

port = 7777

#代理服务器绑定的网口地址

listen = 127.0.0.1

#超时最长时间

timeout = 600

#是否绑定相同端口

bindsame = no

#过滤配置表文件位置

filter = ./filter

#是否根据 url 地址过滤

filterURLs = **false**

#是否开启过滤扩展项

filterextended = **false**

#过滤时是否大小写敏感

filtercasesensitive = **false**

#多级代理时，上游代理服务器地址

upstream = 127.0.0.1:1024

#反向代理基地址

reversebaseurl = http://localhost:8888/

#反向代理地址表文件位置

reversepathfile = ./reverse

2、过滤配置表文件

当在基本配置表中定义了过滤配置表文件的地址，即表示启用了 HTTP 代理服务器的过滤功能。过滤功能的作用是用于过滤一些不安全的远端服务器域名以及被屏蔽了的客户端 IP。只需要在过滤配置表中输入需要过滤的地址或 IP，程序在执行时便会不允许被过滤的地址或 IP 访问代理服务器。

3、反向代理地址表文件

反向代理地址表文件是用于查找目标服务器而存在的。用户是使用反向代理功能时需要在地址栏输入反向代理服务器基地址和代理内容的链接，然后代理服务器就可以通过这个链接在反向代理地址表中查找链接对应的目标服务器，最后在目标服务器上面请求用户需要的资源。图 3-6 对反向代理请求时地址的各部分就行了描述：

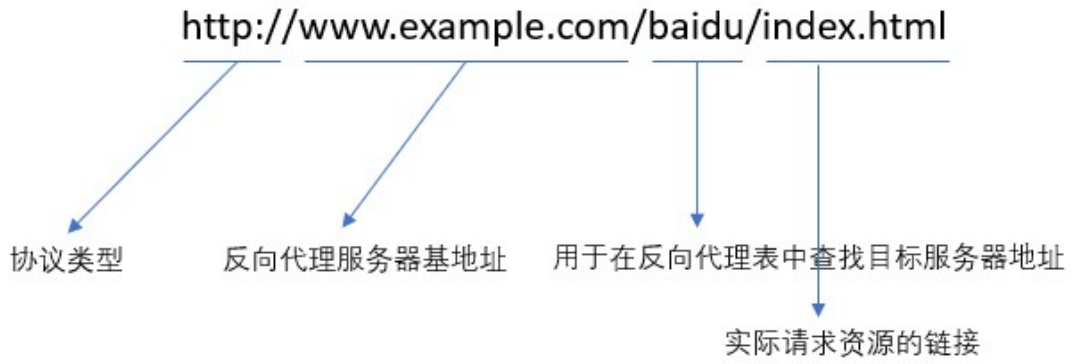

图 3-6 反向代理过程地址组成描述

3.6.2 命令行参数输入选项部分

命令行参数输入是除了配置文件之外另一个用于代理服务器程序配置项输入的部分。命令行参数输入的方法是：在运行程序时，程序地址名后加入需要输入的参数，多个参数之间用空格隔开。可以输入的参数有以下几种：

1、-h 或--help

这是用于打印出程序帮助信息的参数，使用它可以简单查看程序的一些描述信息和可以使用输入参数等，当不知道程序有那些可用的输入参数选项时就可以使用 help 查看。图 3-7 为 HTTP 代理服务器 help 参数输出的信息。

```
[root@radishkill build]# ./HttpProxy --help

Usage: xxx [options]:

Options are::
  -h [ --help ]                 produce help message.
  -c [ --conf ] arg (=proxy.cfg)  use an alternate configuration file.
  -p [ --port ] arg             listen port.
  -v [ --version ]              display version information.

For support, please visit
<https://radishkill.github.io/>:

[root@radishkill build]#
```

图 3-7 help 指令输出的信息

2、-c 或--conf

用于指定一个基本配置文件位置来运行程序。

3、-p 或--port

用于指定 HTTP 代理服务器接收客户端请求时绑定端口，使用这个参数会覆盖基本配置文件中原有的端口配置。

4、-v 或--version

查看程序的版本号。

3.6.3 配置文件与命令行参数输入部分程序代码设计

这部分代码主要用于对配置文件和参数输入的数据解析，解析使用的是 ProgramOptions 库。下面将直接展示包含基本配置文件与命令行参数输入部分的程序代码函数段：

```
int process_cmd(int ac, char **av) {
  msystem::ConfigPool* config_pool = msystem::ConfigPool::GetConfigPool();
  //控制台选项参数初始化
  po::options_description general_options("Options are:");
  general_options.add_options()
     ("help,h", "produce help message.")
     ("conf,c", po::value<std::string>(&config_pool->config_file)->default_value("proxy.cfg"), "use an alternate configuration file.")
     ("port,p", po::value<uint16_t>(), "listen port.")
     ("version,v", "display version information.")
  ;

  po::options_description usage("Usage: xxx [options]");

  po::options_description support("For support, please visit\n<https://radishkill.github.io/>");

  po::options_description visible("");
  visible.add(usage).add(general_options).add(support);
```

```
        //配置文件参数初始化
        po::options_description config("Configuration");
        config.add_options()
            ("port", po::value<uint16_t>(), "listen port")
            ("listen", po::value<std::string>(), "listen address")
            ("upstream", po::value<std::string>(&config_pool->upstream), "")
            ("timeout", po::value<uint32_t>()->default_value(60), "")
            ("bindsame", po::bool_switch()->default_value(false), "")
            ("filter", po::value<std::string>(&config_pool->filter), "")
            ("filterURLs", po::bool_switch()->default_value(false), "")
            ("filterextended", po::bool_switch()->default_value(false), "")
            ("filtercasesensitive", po::bool_switch()->default_value(false), "")
            ("maxthread", po::value<uint32_t>(&config_pool->maxthread)->default_value(10),
"maxthread")
            ("reversebaseurl", po::value<std::string>(&config_pool->reverse_base_url), "reverse
baseurl")
            ("reversepathfile", po::value<std::string>(&config_pool->reversepath_file), "reverse
pathfile")
            ;
        po::variables_map vm;
        po::store(po::parse_command_line(ac, av, general_options), vm);
        po::notify(vm);

        //处理 help 参数 显示并退出
        if (vm.count("help") || vm.empty()) {
          std::cout << visible << std::endl;
          std::exit(0);
        }
        //处理 version 参数 显示并退出
        if (vm.count("version")) {
```

```
  cout << "develop version" << endl;
  std::exit(0);
}
//处理端口与地址参数，并保存
if (vm.count("listen")) {
  config_pool->listen_addrs.push_back(vm["listen"].as<std::string>());
}
if (vm.count("port")) {
  config_pool->port = vm["port"].as<uint16_t>();
}
//读取配置文件参数
std::ifstream ifs(config_pool->config_file.c_str());
if (!ifs) {
  throw  std::string("can not open config file: ") + config_pool->config_file;
} else {
  po::store(po::parse_config_file(ifs, config), vm);
  po::notify(vm);
  ifs.close();
}

if (config_pool->listen_addrs.size() == 0 && vm.count("listen")) {
  config_pool->listen_addrs.push_back(vm["listen"].as<std::string>());
} else {
  throw "bind_address in config file is necessary";
}

if (config_pool->port == 0 && vm.count("port")) {
  config_pool->port = vm["port"].as<uint16_t>();
} else {
  throw "port in config file is necessary";
```

```
    }
    if (vm.count("timeout")) {
      config_pool->idletimeout = vm["timeout"].as<uint32_t>();
    }
    if (vm.count("bindsame")) {
      config_pool->bindsame = vm["bindsame"].as<bool>();
    }
    if (vm.count("filterURLs")) {
      config_pool->filter_url = vm["filterURLs"].as<bool>();
    }

    if (vm.count("filterextended")) {
      config_pool->filter_extended = vm["filterextended"].as<bool>();
    }

    if (vm.count("filtercasesensitive")) {
      config_pool->filter_casesensitive = vm["filtercasesensitive"].as<bool>();
    }
    return 1;
  }
```

通过上述函数解析后，配置文件与控制台选项参数就会被保存到 config_pool 这个变量内，其中配置文件与控制台选项相同参数部分会以控制台选项的参数为准覆盖配置文件的参数。

除了基本配置文件之外，程序刚启动时还需要解析的配置文件有过滤配置表和反向代理地址表两个文件。因为两个文件使用的存储规范不符合 ProgramOptions 库，所以使用的是程序内的处理代码。代码使用逐行读取的方式将表中的内容读取出来，然后用键值对的方式存储到了 map 变量内。下面给出反向代理地址表文件解析函数段的内容与注释：

```
  void ParseReverseFile() {
    msystem::ConfigPool* config_pool = msystem::ConfigPool::GetConfigPool();
```

```
if (config_pool->reversepath_file.empty()) {
    return;
}
//打开反向代理地址表文件
std::ifstream ifs(config_pool->reversepath_file);
if (!ifs.is_open())
  throw std::string("can not open reverse file: ") + config_pool->reversepath_file;
std::array<char, 8129> buff;
std::string v1;
std::string v2;
uint8_t skip = false;
uint8_t v2_flag = false;
while (!ifs.eof()) {
  //读取一定字符
  std::streamsize size = ifs.readsome(buff.data(), buff.max_size());
  if (size == 0) {
    break;
  }
  //对读取的字符进行处理
  for (auto iter = buff.begin(); iter != buff.begin()+size; ++iter) {
    if (v1.empty() && *iter == '#') {
      skip = true;
    } else if (v1.empty() && *iter == ' ') {

    } else if (!v1.empty() && *iter == ' ') {
      v2_flag = true;
    } else if (*iter == '\n') {
      if (!skip)
        config_pool->reversepath_list.insert(std::make_pair(v1, v2));
      v1.clear();
```

```
            v2.clear();
            skip = false;
            v2_flag = false;
          } else {
            if (!v2_flag)
              v1.push_back(*iter);
            else
              v2.push_back(*iter);
          }
        }
      }
      //关闭文件
      ifs.close();
    }
```

3.6.4 任务处理与线程池部分

在介绍任务处理之前必须对 Asio 库中的 io_context 类进行介绍。

io_context 是网络模型中处理网络信息的最小任务管理单元，Asio 库在处理网络连接时必须首先创建一个 io_context 对象，这样网络连接的读写任务都可以在 io_context 对象的环境中运行。

图 3-8 对 io_context 的工作结构进行了描述。io_context 类的实例被创建后就可以通过向里面插入任务代码段的方式运行任务，io_context 正在运行任务时也可以插入新任务，这样它就会在执行了上一个任务后开始执行下一个任务，直到没有新的任务插入为止。

HTTP 代理服务器的线程池模型使用 Boost 里面的 thread_group 类进行管理，其中建立线程池模型的过程为：实例化一定个数的 io_context 类实例处理，将这些实例的指针用循环队列的方式保存起来，方便之后调用插入新任务，然后用 thread_group 类申请同样个数线程空间，在线程里面执行 io_context 实例的 run 方法。向线程池插入新任务时，需要先从队列的前面取出 io_context 实例，然后插入任务函数，再把这个实例放到队列的尾部。当因为插入过多任务导致队列里的 io_context 都处于执行状态时，新的任务就会被插入到正在执行的任务后面，等待前一个任务执行完成后，执行新的任务。在

实际运行环境中，大部分任务实际都处于休眠阶段，正在等待 I/O 口的响应，如果新插入的任务提前被唤醒，io_context 立即运行被先唤醒的任务函数。图 3-9 对实际线程池模型运行结构进行了描述。

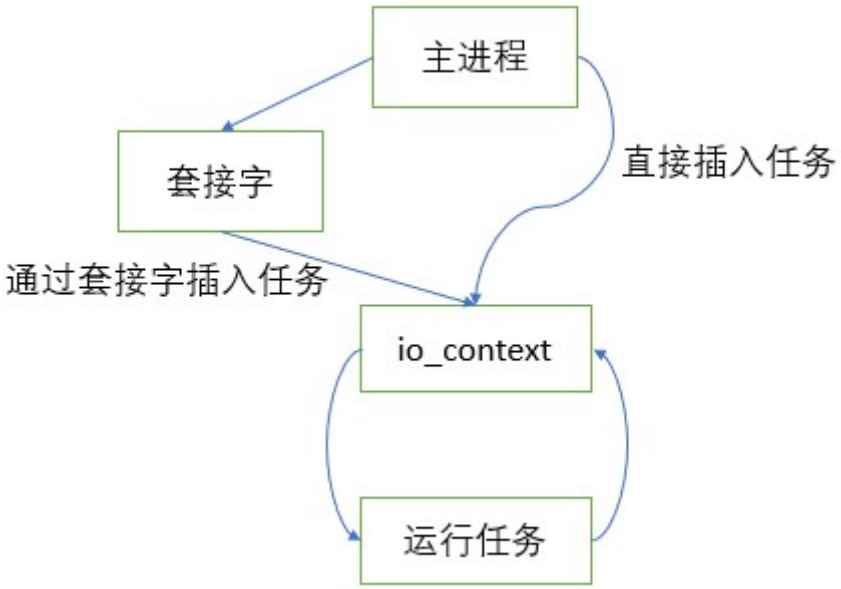

图 3-8 io_context 工作结构图

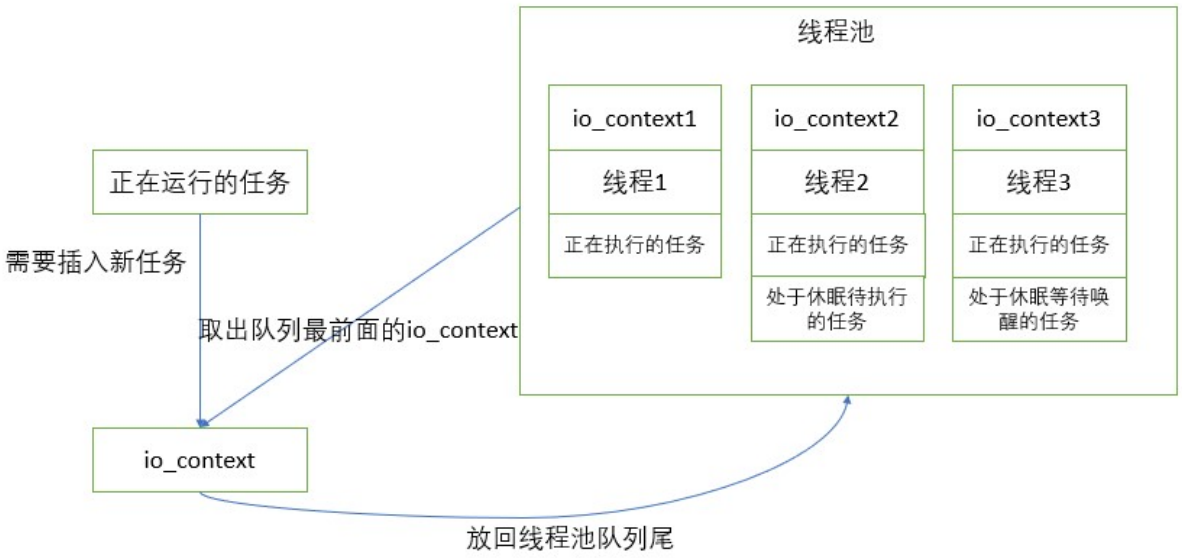

图 3-9 线程池模型任务插入与处理过程

下面给出线程池建立部分的详细代码段与注释:

```
//建立线程池模型
msystem::ioctx_deque io_contexts;
//初始化任务队列
std::deque<msystem::ba::io_service::work> io_context_work;
boost::thread_group thr_grp;
for (int i = 0; i < 10; ++i) {
    msystem::io_context_ptr ios(new msystem::ba::io_context);
    io_contexts.push_back(ios);
    io_context_work.push_back(msystem::ba::io_context::work(*ios));
    //申请线程并绑定任务处理的 io_context
    thr_grp.create_thread(boost::bind(&msystem::ba::io_context::run, ios));
}
```

3.6.5 请求监听部分

请求监听部分的作用是监听来自客户端的连接请求，HTTP 代理服务器需要先开启 TCP 端口监听服务功能，这样只要有来自 TCP 客户端是连接，经过 TCP 三次握手确认后，请求监听部分代码就会把确认过的连接转交给请求报文接收任务处理线程进行处理，然后开启下一次监听。

下面将展示代理服务器请求监听部分初始化的代码段与注释：

```
//解析地址
asio::ip::tcp::resolver resolver(*io_contexts_.front());
asio::ip::tcp::endpoint listen_endpoint = *resolver.resolve(address, port).begin();
//初始化监听端口
acceptor_.open(listen_endpoint.protocol());
acceptor_.set_option(asio::ip::tcp::socket::reuse_address());
acceptor_.bind(listen_endpoint);
acceptor_.listen();
//对多个网口监听进行处理
std::vector<std::string>::iterator iter = config_pool_->listen_addrs.begin();
for (; iter != config_pool_->listen_addrs.end(); ++iter) {
 if (iter == config_pool_->listen_addrs.begin())
```

```
        continue;
      boost::asio::ip::address addr = boost::asio::ip::make_address((*iter).c_str());
      acceptor_.set_option(boost::asio::ip::multicast::join_group(addr));
    }
```

这部分代码使用的是 Asio 库中 acceptor 类，其原理与 C 语言 TCP 套接字监听相同。这段代码首先从配置文件中解析出需要监听地址与端口，然后打开 acceptor 实例并绑定地址与端口，最后开始监听来自客户端的请求。当有客户端连接服务器时，系统就会接受连接请求，下面展示代理服务器接收到来自客户端的连接请求后运行的处理函数，并等待下一个请求这一过程的代码段与注释:

```
    int ProxyServer::StartAccept() {
      //从线程池中取出线程用于执行监听
      io_contexts_.push_back(io_contexts_.front());
      io_contexts_.pop_front();
      std::shared_ptr<Connection> new_connection = std::make_shared<Connection>(*io_c
ontexts_.front());
      acceptor_.async_accept(new_connection->Socket(),
                             boost::bind(&ProxyServer::HandleClient, this, new_connection, ba::plac
eholders::error));
      return 0;
    }


    void ProxyServer::HandleClient(std::shared_ptr<Connection> new_connection, const bo
ost::system::error_code &error) {
      if (error) {
        std::cout << error.message() << std::endl;
        return;
      }
      //处理请求并再次开始监听
      new_connection->Run();
      StartAccept();
```

```
}
```

3.6.6 请求报文处理部分

请求报文处理部分代码包含两个部分，第一部分为报文接收阶段，第二部分为报文处理阶段。接收部分使用的是 Asio 库函数 async_read 进行读取，async_read 能异步接收来自客户端的请求报文，当有数据需要接收时，async_read 的处理函数就被唤醒，没有数据时就保持休眠状态，这样就可以将系统资源交给急需处理的任务函数。async_read 每次读取一定数量的报文内容，然后将这部分内容交给请求报文解析函数进行解析，如果解析函数返回 Good，即表示请求报文已经处理完全了，如果返回 Bad，则表示请求报文内容有误，并放弃了本次解析。下面展示了请求报文接收函数代码段的内容：

```
void Connection::ReadRequest(const bs::error_code& ec, size_t len)
{
 if (ec)
 {
   std::cout << ec.message() << std::endl;
   Shutdown();
   return;
 }
 HttpParser::ResultType result;
 //使用解析函数解析请求报文
 std::tie(result, std::ignore) = http_parser_.ParseRequest(cli_recv_buffer_.data(), cli_rec
v_buffer_.data() + len);
 if (result == HttpParser::kGood)
 {
   std::cout << "url: " << http_.raw_url << "\n";
   std::cout << "method: " << http_.method << "\n";
   std::cout << "version: " << http_.http_version << "\n";
   for (const auto& h : http_.headers) {
     std::cout << h.first << ":" << h.second << "\n";
   }
   //使用处理函数处理请求报文
```

```
    if (ProcessRequest() < 0) {
      Shutdown();
      return;
    }
    //连接上级服务器
    ConnectToServer();
  } else if (result == HttpParser::kBad) {
    Shutdown();
    return;
  } else {
    //异步读取请求报文内容
    ba::async_read(client_socket_, ba::buffer(cli_recv_buffer_), ba::transfer_at_least(1),
        boost::bind(&Connection::ReadRequest,
            shared_from_this(),
            ba::placeholders::error,
            ba::placeholders::bytes_transferred));
  }
}
```

请求报文解析部分使用的一个单独的 HTTP 协议报文解析类进行的管理。解析函数会逐个字符的去读取报文内容，然后按照状态机的方式对报文的处理状态进行严格管理，一旦发现一个不符合报文规范的字符，解析函数就会返回 Bad，表示请求报文内容有误，本次请求被拒绝。其中请求报文读取到的内容会被存放到 HTTP 协议内容结构体内，方便后续调用。下面展示 HTTP 协议内容结构体定义部分的代码段：

```
struct HttpProtocol
{
  std::string method;
  std::string raw_url;
  std::string status_code;
  std::string reason_phrase;
  std::string url;
```

```
    std::string http_version;
    std::string host;
    int port;
    uint8_t protocol_major;
    uint8_t protocol_minor;
    uint8_t connect_method;
    HeadersMap headers;
    std::size_t data_length;
    std::string data;
};
```

HTTP 协议结构体的主要内容包括以下几个：

1.method 用于保存 HTTP 协议请求报文的请求方法。

2.raw_url 用于保存请求报文的原始 url 地址。

3.status_code 用于保存响应报文的请求状态码。

4.reason_phrase 用于保存响应报文的响应信息。

5.http_version 用于保存 HTTP 协议的版本号。

3.6.7 连接服务器与上传请求报文部分

当来自客户端的请求报文解析成功后，就可以从请求报文中得到请求方法和目标服务器地址两个关键的数据。请求方法用于选择客户端请求服务器时使用的连接方法，包括 GET、POST、CONNECT 等，其中 GET 和 POST 方法连接处理流程一样，而 CONNECT 方法则需要建立 HTTPS 安全链接。目标服务器地址用于保存客户端想要连接的服务器的实际地址。

连接上位机部分代码，首先使用 Asio 库中的 query 解析类对远端服务器的地址和端口进行解析，得到服务器的实际 IP，然后使用 async_connect 异步连接方法来连接远端服务器。当连接成功的任务处理函数被执行后，处理函数就会判断连接是否使用了 CONNECT 方法，如果不是就发送组装好的请求报文给远端服务器，如果是则发送安全链接响应报文给客户端。图 3-10 为连接建立过程的流程图。

3.6.8 HTTP 代理响应部分

HTTP 代理的响应属于 HTTP 普通代理过程的最后一步，代理服务器需要等待远端服务器返回响应报文，然后代理服务器再将响应报文返回给客户端。来自远端服务器的

响应报文包含 5 部分内容，它们分别是：响应码、HTTP 协议版本号、响应码内容、响应头、响应消息体。响应码与响应码内容反映了本次请求返回结果的状态，下面列出几个常见的响应码：

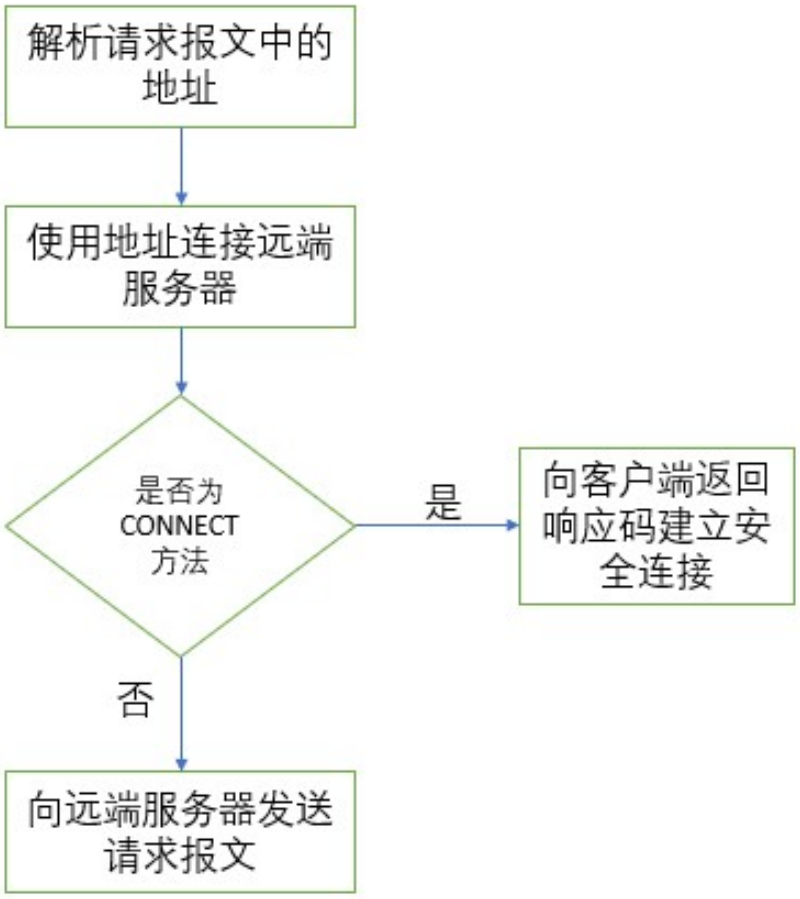

图 3-10 连接建立流程图

1、100 Continue

服务器只接受到了部分请求，等待客户端继续发生其余的内容。

2、101 Switching Protocols

服务器跟随客户端的请求，将转换到另一种协议。

3、200 OK

请求成功。

4、301 Moved Permanently

所请求的页面已经转移到了新的地址。

5、400 Bad Request

服务器未能正确解析请求报文。

6、404 Not Found

服务器找不到需要请求的页面。

7、500 Internal Server Error

请求未完成，服务器遇到未知的错误。

所以只有当来自服务器的响应码为 200 时才表示本次请求被正确处理，并返回了相应的内容给客户端。

3.6.9 HTTPS 安全链接建立部分

当客户端发出的请求报文方法为 CONNECT 时，表示客户端需要本次连接的建立为 HTTPS 的形式。这种方法的处理方式与 GET 和 POST 类型的方法将有所不同。对于 CONNECT 方法的请求，代理服务器会首先去连接远端服务器，当远端服务器发回连接成功的信号后，代理服务器会向客户端发送建立安全链接的响应报文。其中响应报文的内容为：“HTTP 协议版本 200 Connection established”，后面还可以添加与代理服务器版本信息相关的内容。

客户端在收到来自代理服务器的响应报文后，就会发送其支持的加密算法给服务器，代理服务器收到内容后不做处理直接转发给远端服务器，然后远端服务器会发回其选定的加密算法，代理服务器也是直接转发内容即可，这之后的内容代理服务器只是充当客户端与代理服务器之间的内容中转站，不需要做任何处理。

图 3-11 展示了客户端与服务器之间建立安全链接的过程。因为现在互联网上大部分网站都已经替换为 HTTPS 站点了，所以 HTTP 代理服务器支持 HTTPS 安全连接的建立也是非常重要的。

3.6.10 反向代理功能

HTTP 代理服务器对反向代理功能提供了简单的支持，开启反向代理功能需要在基本配置文件中设置反向代理基地址与反向代理地址表文件位置两个配置项，并对反向代理地址表文件的内容进行补充。反向代理流程是从处理客户端请求报文处开始的，当读取到配置文件中设置了反向代理基地址时，并且请求服务器地址与基地址相同，客户端发送请求地址首先会被去除掉基地址部分，然后读取反向代理地址表的地址，按照地址表的内容替换请求地址，并向相应的服务器请求客户端需要的文件，最后返回从服务器请求到的文件给客户端。

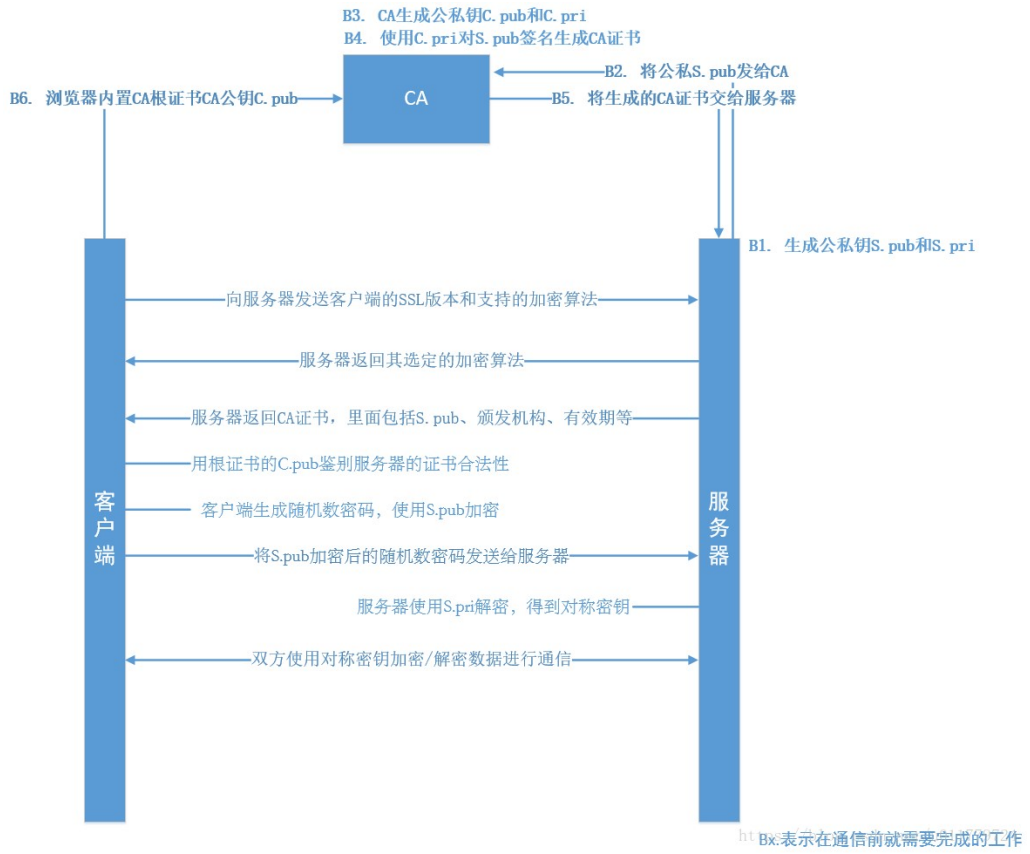

图 3-11 客户端与服务器建立 HTTPS 安全链接的过程

下面展示反向代理地址替换部分的代码段与注释：

```
//用反向代理地址表内容确认是否反向代理
if (!config_pool->reversepath_list.empty()) {
  std::string rewrite_path;
  for (const auto& v : config_pool->reversepath_list) {
    p = http_.raw_url.find(v.first);
    if (p == 0) {
      //找到地址相应地址表 并替换
      rewrite_path = v.second;
      rewrite_path += http_.raw_url.substr(v.first.length());
      break;
    }
  }
  if (!rewrite_path.empty()) {
```

```
    http_.raw_url = rewrite_path;
  }
}
```

第 4 章　系统测试过程及结果

4.1 测试环境搭建

为了对 HTTP 代理服务器的性能进行测试，需要使用到以下几种工具：

1、Linux 平台

用于编译与运行 HTTP 代理服务器程序，编译时使用的是运行在本地的 ArchLinux 系统，运行时使用的是运行在远端的阿里云服务器 CentOSLinux 系统。

2、C++编译器

ArchLinux 平台的代码使用的是 GCC 编译器与 CMake 工具进行编译。云服务器平台使用相同版本的 C++库运行，其中 C++库来自 GCC 编译器，GCC 编译器的版本为 9.2.0。

3、浏览器

浏览器主要运行于客户端，用来连接云服务器的 HTTP 代理服务器程序。浏览器使用的是 Google Chrome 浏览器，版本为 83.0.4103.61。因为浏览器不自带代理连接工具，所以测试还使用了 Google Chrome 浏览器官方商店中的 Proxy SwitchyOmega 代理连接工具用于连接代理服务器。Google Chrome 浏览器包含开发者工具可以对访问网络时的资源拉取时间进行测试。

4、网络数据包分析工具

这个工具使用的是免费并且开源的 WireShark 软件。WireShark 可以抓取电脑网络连接中的各类数据包，并用于查看和分析，是网络程序开发过程中使用的重要测试工具之一。

5、云服务器连接工具

这个工具使用的 Git 管理工具中附带的 Linux 虚拟平台，使用虚拟平台中自带的 SSH 软件连接云服务器。

6、网络环境

网络环境包含局域网和公网两部分，局域网由测试电脑和路由器直接组成，公网部分由阿里云服务器提供，配合一个 slayer.top 的域名，就可以在测试环境还原真实 HTTP 代理服务器的工作环境。

4.2 测试过程

4.2.1 数据内容正确性

这部分测试主要用于确认HTTP代理服务器是否正确的代理了浏览器所发出的请求报文。包含非 CONNECT 方法与 CONNECT 方法两个部分，下面对测试过程与结果进行介绍：

1、非 CONNECT 方法

非 CONNECT 方法指的是测试时浏览器发送的请求报文的方法体是 GET、POST 等类型。下面以访问 http://slayer.top:12345/网站为准，这是搭建在阿里云服务器上面的 Web 服务器。因为网页内容较少，所以用于对数据内容部分进行测试。

图 4-1 为无代理情况下访问 http://slayer.top:12345 时数据流在 WireShark 软件上的情况。

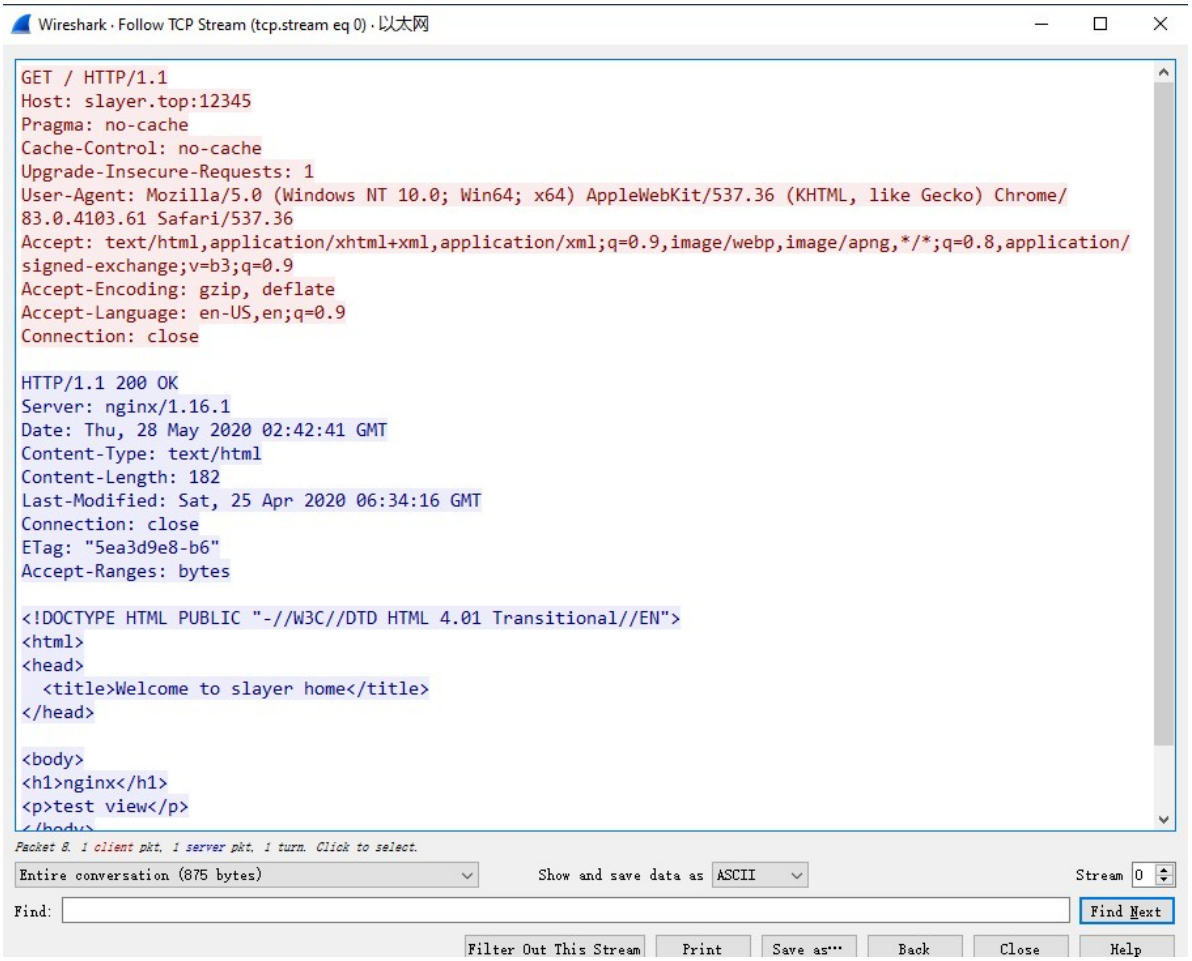

图 4-1 无代理访问 http://slayer.top:12345 的数据流

图 4-2 为使用 HTTP 代理服务器访问相同站点时的数据量情况。

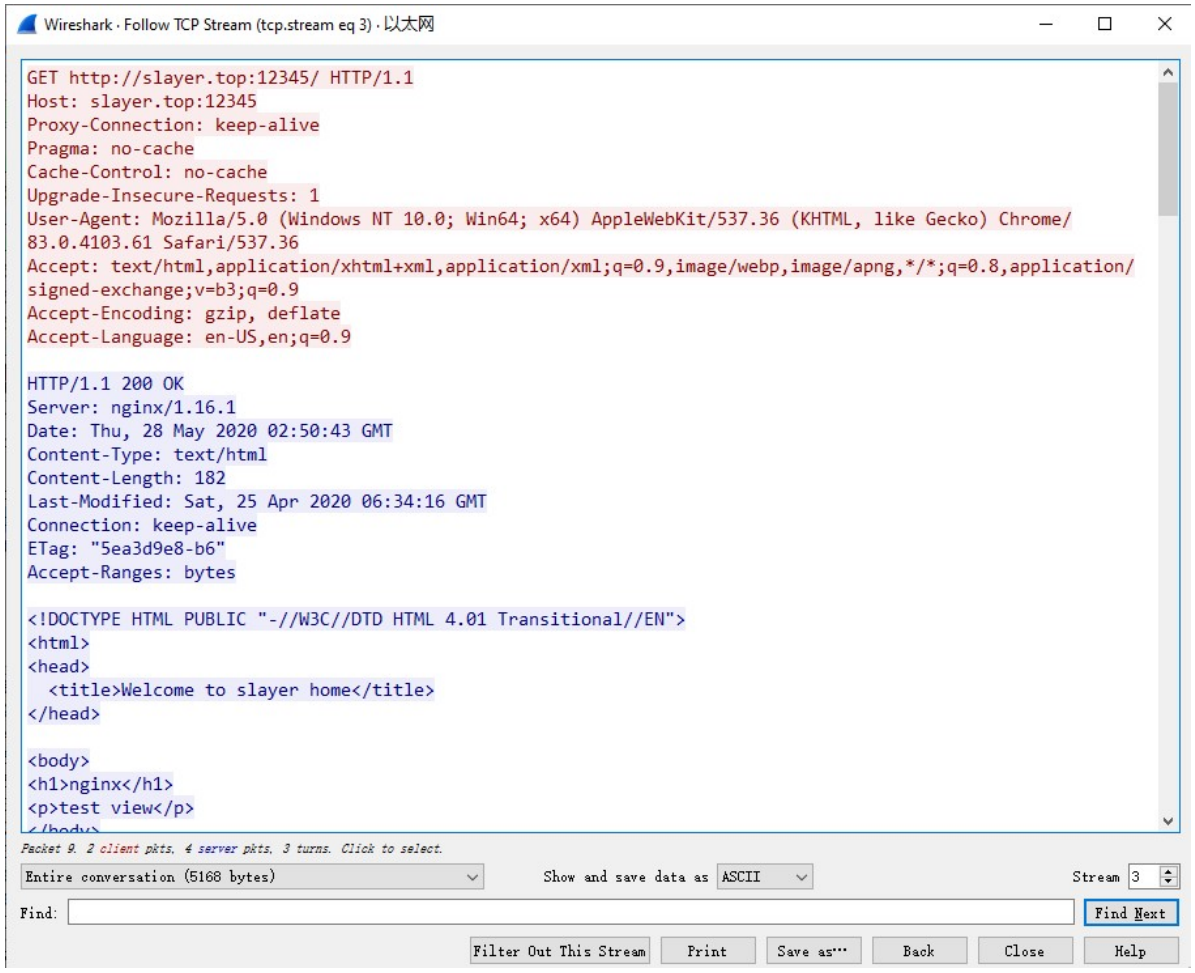

图 4-2 使用代理访问 http://slayer.top:12345 的数据流

对比分析两图中的数据流内容可以得出使用代理与不使用代理访问 http://slayer.top:12345 得到的数据一致，由此可见 HTTP 代理服务器在非 CONNECT 方法下可以正确代理。

2、CONNECT 方法

当请求报文为 CONNECT 方法时，其实质为请求网页为 HTTPS 类型。由于 HTTPS 类型网页的内容在传输时已经被加密处理，所以测试这类请求的数据正确性只能通过浏览器是否正确显示目标网页内容为准。下面通过访问 https://www.baidu.com 站点为例，这个站点是百度搜索的首页地址。

图 4-3 为无代理情况访问 https://www.baidu.com 时浏览器响应内容。图 4-4 为使用 HTTP 代理服务器访问百度首页响应的内容。

图 4-3 与图 4-4 访问百度首页得到的内容基本一致，可以得出 HTTP 代理服务器可以正确代理 CONNECT 类型连接。

图 4-3 无代理访问百度首页得到的内容

图 4-4 使用代理访问百度首页得到的内容

4.2.2 响应速度测试

这部分测试主要用于确定 HTTP 代理服务器在代理过程中是否对连接的速度存在较大的影响。测试时使用的工具是 Google Chrome 浏览器的开发者工具，访问的站点为百

度首页。

图 4-5 为无代理情况下访问百度首页时，在开发者工具中显示的响应速度。

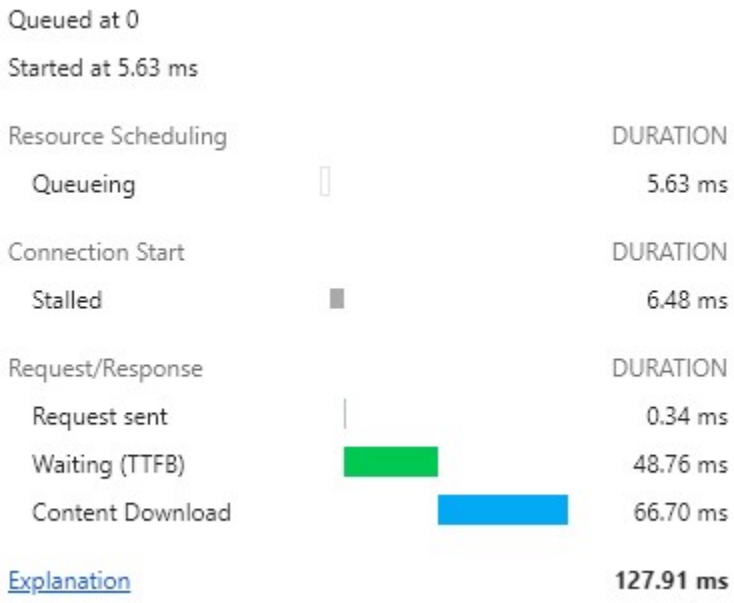

图 4-5 无代理情况访问百度首页速度显示

图 4-6 为使用 HTTP 代理服务器访问百度首页的响应速度显示情况。

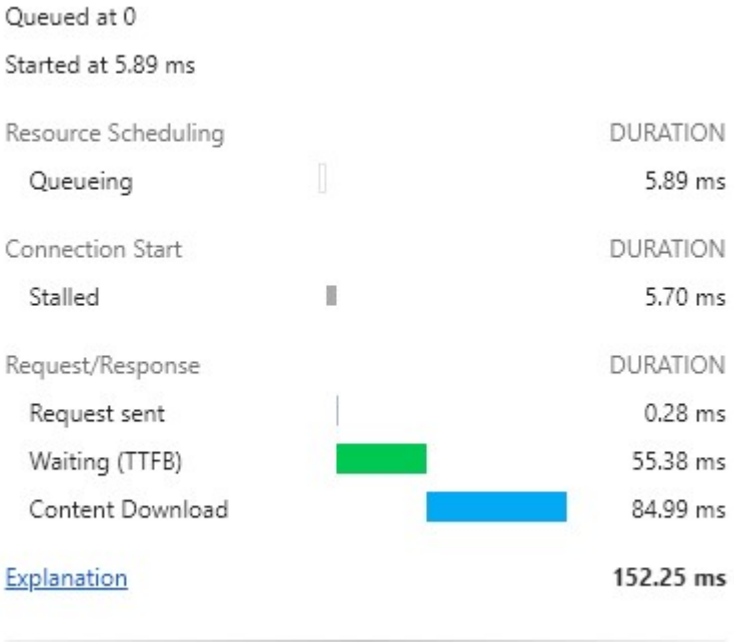

图 4-6 使用代理访问百度首页速度显示

从图中对比分析可以得到，无代理与使用代理拉取百度首页 61.8KB 内容使用的总

时长分别为 127.91ms 与 152.25ms。由于网络连接存在波动等种种原因，这个时间为多次测试下较稳定的数据。可以得出使用 HTTP 代理服务器代理情况下由于连接多经历了一级连接，所以响应速度会比无代理情况稍微慢一些，不过这个速度在可以接受的范围之内。

4.3 对比分析

使用网络上能比较常用的轻量级 HTTP 代理服务器 TinyProxy 与本设计所开发的 HTTP 代理服务器进行对比分析。对比的主要内容为响应速度，即两款软件在拉取同一页面时，所花费的时间。

图 4-7 为 TinyProxy 代理服务器拉取百度首页网络较稳定情况下所花费的时间图。

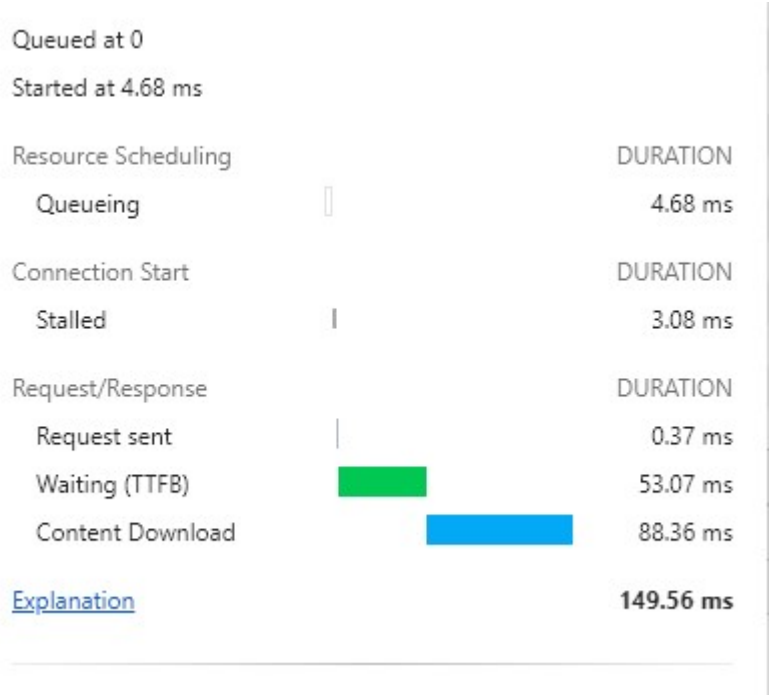

图 4-7 TinyProxy 拉取百度首页内容的速度显示

通过图 4-7 与图 4-6 对比分析可以发现两款代理服务器软件在拉取百度首页 61.8KB 内容时所花费的时间基本一样。本设计所制作的 HTTP 代理服务器软件在性能方面与网络上的流行软件大致相同，符合一款代理服务器对响应速度的要求。

结论

本文所设计 HTTP 代理服务器采用了基于 Linux 的 C++编程语言进行编写，使用了 Boost 中的 Asio 先进 C++网络编程库进行基本网络传输部分的模块在制作。任务控制方面采用了线程池模型相关的技术以提高 HTTP 代理服务器在实际运行中的响应速度与效率。同时代理服务器具有对非法连接客户端与危险网站域名的过滤功能，很好的保证了用户在使用代理服务器过程中的网络安全性。经过测试，代理服务器系统在 HTTP 代理过程中的表现极佳，与现如今网络上常用代理服务器相比，访问与传输速度方面基本不相上下。

代理服务器开始诞生时的主要目的是为了提高互联网访问速度，也就是使用缓存的方式提高客户端访问同一数据时的速度，但是随着现在网络质量与网络速度的逐步提高，代理服务器缓存的作用也被逐渐淡化，因此本次设计的 HTTP 代理服务器并不包含缓存相关技术的研究。

通过本次设计，发现现在代理服务器设计更加注重的是其在分布式方面的应用，分布式技术也就是将服务器资源存放到多个服务器节点上，使得多服务器计算速度更快，连接更加可靠。给代理服务器实现反向代理技术，用户访问反向代理服务器就是分布式系统接入，访问各个节点的重要手段之一。反向代理服务器还能对资源服务器进行访问速度评价，得出用户访问某个资源最快最稳定的路径，当存在大量访问是反向代理服务器还能将连接分发至不同的资源服务器，降低服务器的负载。目前本次所设计的 HTTP 代理服务器包含简单的反向代理功能，可以反向代理一些来自客户端的简单 HTTP 请求。

本次系统设计还包含一些设计难点。其中最难的就是 Linux 系统与 C++编程语言的熟练问题。Linux 系统与 Windows 系统相比缺少了很多可视化的设置，很多配置都必须通过控制台来完成，这极大的提高了 Linux 系统的学习难度。C++编程语言虽然上手简单，但实际上自从 C++11 新特性发布以后，C++就包含许多好用但是学习起来非常困难的语法。但想要设计出高性能、稳定的 HTTP 代理服务器选择这一套方案也是最佳的。可见在做设计的过程中不能因为使用工具和制作工程的复杂，而选择简单的方法。

虽然本次设计的 HTTP 代理服务器在功能使用上并不是很完善，对于内容过滤和反向代理方面都只是一个测试阶段功能，在实际使用中的表现也不尽人意，同时还有一些

原本计划设计的功能，却因为种种原因而没有实现的，但是我一定会在将来继续学习新的知识，使用新知识来进一步完善和提高本次设计中很多不足的地方。

致谢

本设计与论文是在我的导师詹曦老师的亲切关怀与悉心指导下一步一步完成的。詹曦老师不仅在设计内容上给我认真指导，同时对于我学习、生活上也给予了无微不至的关怀。在此谨向詹曦老师致以诚挚的谢意和崇高的敬意。

同时我还要感谢那些在我遇到困难时无言给我提供帮助的可敬师长、同学和朋友们。是你们帮助我完成了我大学的最后一课，在这里请接收我最真诚的谢意！最后我还要感谢我的父母，谢谢你们！

参考文献

[1] 常智.高性能 HTTP 反向代理研究与实现[D].哈尔滨工程大学，2013.

[2] 狄刚.HTTP 实现代理服务器及缓存替换算法的研究[D].吉林大学，2010.

[3] 周伟.高性能 HTTP 代理服务器关键技术研究与实现[D].哈尔滨工程大学，2012.

[4] 赵玉伟.WWW 中缓存机制的应用研究[D].武汉理工大学，2006.

[5] 彭妮.阶段化异步事件驱动高性能 Web 服务器[D].长沙理工大学，2006.

[6] 杨杰，舒辉.代理服务器特性探测技术研究[J].计算机应用，2009，29(03):826-829.

[7] 李家欣，倪亮，等.具备高速缓存的 HTTP 代理防火墙的设计与实现[J].计算机工程，2003(3): 84-85.

[8] zvrop.HTTP 代理服务器也 DIY[J].黑客防线，2004(02S):44-48.

[9] 孙永辉，姜昱明.HTTP 代理服务器的设计与实现[J].计算机工程与设计，2003，24(7):56-58.

[10] 倪卫东.Http Proxy 的运行机制及实现方法[J].航船电子对抗，2006(1): 35-38.

[11] 廖勇，邓欣茹.用代理服务器联通网络[J].网络安全和信息化，2017(06):61-64.

[12] 车葵，刑书涛.网络代理服务器的设计与实现[J].学术.技术，2007(12):38-40.

[13] 吴益清，谢培泰.代理服务器的原理与实现[J].信息工程大学学报，2000，1(4):40-43.

[14] 陈忠菊.HTTP 服务器的研究和实现[J].电脑编程技巧与维护,2018(08):133-135.

[15] 赵泽华.反向代理服务器缓存动态配置系统的设计与实现[D].北京邮电大学,2019.

[16] 曹伟,邹庆华.Linux 的 http 网络服务器[J].电子技术与软件工程,2016(14):21.

[17] 郭大伟,张伟,姜晓艳.一种基于 Nginx 的 UDP 反向代理服务器数据转发策略[J].北京信息科技大学学报(自然科学版),2019,34(06):87-91.

[18] 江存.基于 Linux 的 2 种 HTTP 服务器实现与对比分析[J].现代计算机(专业版),2017(24):58-61+76.

[19] 戴伟,马明栋,王得玉.基于 Nginx 的负载均衡技术研究与优化[J].计算机技术与发展,2019,29(03):77-80.

[20] R. Fielding and J. Reschke.RFC 7230: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing.https://tools.ietf.org/html/rfc7230.June 2014.

[21] R. Fielding and J. Reschke.RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content.https://tools.ietf.org/html/rfc7231.June 2014.

[22] P. Balakumar.Securing Web Proxy Based Network from Http Attacks with Provision for Detecting Attacker Nodes[J].International Journal of Engineering Science and Innovative Technology，2014，3(2).

[23] Avinash.，V，et al.HTTP Reverse Proxy Authentication[J].International Journal of Advance Engineering and Research Development，2015，2(3).

[24] JEFFREY C.Clarifying the fundamentals of HTTP[J].Software: Practice and experience，2004，2(2).

附录

1、程序主函数

```
int main(int argc, char* argv[]) {
 try {
  //配置管理类 初始化
  msystem::ConfigPool* config_pool = new msystem::ConfigPool();
  msystem::ConfigPool::SetConfigPool(config_pool);
  //读取控制台选项和配置文件
  if (!process_cmd(argc, argv)) {
   std::exit(0);
  }
  //读取过滤配置表
  msystem::Filter* filter = new msystem::Filter(config_pool->filter);
  msystem::Filter::SetFilter(filter);
  //读取反向代理配置表
  ParseReverseFile();
  //建立线程池模型
  msystem::ioctx_deque io_contexts;
  std::deque<msystem::ba::io_service::work> io_context_work;
  boost::thread_group thr_grp;
  for (int i = 0; i < 10; ++i) {
      msystem::io_context_ptr ios(new msystem::ba::io_context);
      io_contexts.push_back(ios);
      io_context_work.push_back(msystem::ba::io_context::work(*ios));
      thr_grp.create_thread(boost::bind(&msystem::ba::io_context::run, ios));
  }
  //启动 HTTP 代理服务器类
```

```
    msystem::ProxyServer proxy_server(io_contexts);
    //开始监听客户端请求
    proxy_server.StartAccept();
    thr_grp.join_all();
  } catch (std::exception& e) {
    std::cerr << "error: " << e.what() << "\n";
    return 1;
  } catch (std::string& e) {
    std::cerr << "error: " << e << "\n";
    return 1;
  } catch (...) {
    std::cerr << "Exception of unknown type!\n";
  }
  return 0;
}
```

2、proxyserver 类的声明与实现

声明部分：

```
#ifndef PROXY_SERVER_H
#define PROXY_SERVER_H
#include <vector>
#include <thread>
#include <boost/asio.hpp>
#include "common.h"
#include "proxyconnection.h"
using boost::asio::io_context;
using boost::asio::ip::tcp;
using boost::asio::ip::address_v4;
namespace asio = boost::asio;
namespace msystem {
```

```
typedef std::deque<io_context_ptr> ioctx_deque;

class ConfigPool;

class ProxyServer {
 public:
  ProxyServer(const ioctx_deque& io_contexts);
  int StartAccept();
  ~ProxyServer();
 private:
  void HandleClient(std::shared_ptr<Connection> new_connection, const boost::system::
error_code &error);
  ioctx_deque io_contexts_;
  tcp::acceptor acceptor_;
  ConfigPool* config_pool_;

};
}
#endif // CONNRECEIVER_H
```

实现部分：

```
#include "proxyserver.h"
#include <iostream>
#include <boost/bind.hpp>
#include "conf.h"
#include "proxyconnection.h"
#include "filter.h"
#include "common.h"
namespace msystem {
ProxyServer::ProxyServer(const ioctx_deque& io_contexts)
    : io_contexts_(io_contexts),
```

```
    acceptor_(*io_contexts_.front()),
    config_pool_(ConfigPool::GetConfigPool()) {

  if (config_pool_->listen_addrs.empty())
    throw "bind_address in config file is necessary";

  std::string address = config_pool_->listen_addrs[0];
  std::string port = std::to_string(config_pool_->port);

  //解析地址
  asio::ip::tcp::resolver resolver(*io_contexts_.front());
  asio::ip::tcp::endpoint listen_endpoint = *resolver.resolve(address, port).begin();
  //初始化监听端口
  acceptor_.open(listen_endpoint.protocol());
  acceptor_.set_option(asio::ip::tcp::socket::reuse_address());
  acceptor_.bind(listen_endpoint);
  acceptor_.listen();
  //对多个网口监听进行处理
  std::vector<std::string>::iterator iter = config_pool_->listen_addrs.begin();
  for (; iter != config_pool_->listen_addrs.end(); ++iter) {
    if (iter == config_pool_->listen_addrs.begin())
      continue;
    boost::asio::ip::address addr = boost::asio::ip::make_address((*iter).c_str());
    acceptor_.set_option(boost::asio::ip::multicast::join_group(addr));
  }
}
int ProxyServer::StartAccept() {
  //从线程池中取出线程用于执行监听
  io_contexts_.push_back(io_contexts_.front());
  io_contexts_.pop_front();
```

```
        std::shared_ptr<Connection> new_connection = std::make_shared<Connection>(*io_c
ontexts_.front());
        acceptor_.async_accept(new_connection->Socket(),
                               boost::bind(&ProxyServer::HandleClient, this, new_connection, ba::plac
eholders::error));
        return 0;
      }

      void ProxyServer::HandleClient(std::shared_ptr<Connection> new_connection, const bo
ost::system::error_code &error) {
        if (error) {
          std::cout << error.message() << std::endl;
          return;
        }
        //处理请求并再次开始监听
        new_connection->Run();
        StartAccept();
      }
      ProxyServer::~ProxyServer() {
      }
      }
```

3、proxyconnection 类的声明部分

```
#ifndef CONNHANDLER_H
#define CONNHANDLER_H
#include <boost/unordered_map.hpp>
#include "httpparser.h"
#include "requesthandler.h"
#include "common.h"
namespace msystem {
```

```
class ConfigPool;
class ConnManager;

class Connection : public std::enable_shared_from_this<Connection>  {
 public:
  Connection(ba::io_context& io_ctx);
  //启动函数
  void Run();
  //长连接时协议交换处理函数
  void HandleServerProxyWrite(const bs::error_code& ec, size_t len);
  void HandleServerProxyRead(const bs::error_code& ec, size_t len);
  void HandleClientProxyWrite(const bs::error_code& ec, size_t len);
  void HandleClientProxyRead(const bs::error_code& ec, size_t len);
  //超时检查函数
  void CheckDeadline();
  //连接远端服务器函数
  void ConnectToServer();
  //解析 IP 地址函数
  void HandleResolve(const bs::error_code& ec, ba::ip::tcp::resolver::iterator endpoint_it
erator);
  //处理与远端服务器之间的连接
  void HandleConnect(const bs::error_code& ec, ba::ip::tcp::resolver::iterator endpoint_i
terator);
  //发送 HTTP 请求报文给远端服务器
  void WriteHTTPRequestToServer();
  //发送原始 HTTP 请求给远端服务器
  void WriteRawRequestToServer();
  //处理服务器写入结果
  void HandleServerWrite(const bs::error_code& ec, size_t len);
```

```
//处理服务器读取结果
void HandleServerRead(const bs::error_code& ec, size_t len);
//处理客户端写入结果
void HandleClientWrite(const bs::error_code& ec, size_t len);
//处理 ssl 客户端写入
void HandleSslClientWrite(const bs::error_code& ec, size_t len);
HttpProtocol& GetHttpProtocol() {
  return http_;
}
ba::ip::tcp::socket& Socket() {
    return client_socket_;
}
//重置连接
void Reset() {
  cli_recv_buffer_.fill('\0');
  ser_recv_buffer_.fill('\0');
  http_.Reset();
  server_response_.Reset();
  http_parser_.Reset(&http_);
  is_persistent_ = false;
  is_proxy_connected_ = false;
}
void Shutdown();
~Connection();
private:
void ReadRequest(const bs::error_code& ec, size_t len);
void ComposeRequestByProtocol(const HttpProtocol& http, std::string& http_str);
void ComposeResponseByProtocol(const HttpProtocol& http, std::string& http_str);
void GetSslResponse(std::string& http_str);
void EstablishHttpConnection(HttpProtocol& http, std::string& http_str);
```

```
  int ProcessRequest();
  int ExtractUrl(const std::string& url, int default_port);
  void StripUserNameAndPassword(std::string& host);
  int StripReturnPort(std::string& host);
  void RemoveConnectionHeaders();
  ba::io_context& io_ctx_;
  ConfigPool* config_pool_;
  std::array<char, 8129> cli_recv_buffer_;
  std::array<char, 8129> ser_recv_buffer_;

  HttpProtocol http_;
  HttpProtocol server_response_;
  HttpParser http_parser_;
  RequestHandler request_handler_;
  ba::ip::tcp::socket client_socket_;
  ba::ip::tcp::socket server_socket_;
  ba::ip::tcp::resolver resolver_;
  ba::steady_timer deadline_;
  uint8_t is_upstream_;
  uint8_t is_server_opened_;
  uint8_t is_proxy_connected_;
  uint8_t is_persistent_;
};
}
#endif
```