武汉邮电科学研究院硕士学位论文

基于线程池的轻量级 Web 服务器设计与实现

Design and Implementation of a Lightweight Web Server Based on Thread Pool

专　　业：通信与信息系统

研究方向：互联网技术

导　　师：马卫东

研 究 生：陈沛　　学号：20140003

二〇一七年三月

摘 要

随着互联网技术的不断普及与快速发展，Web 应用以其优秀的通用性与交互性，成为互联网接入平台的首选；然而，市面上流行的几款 Web 服务器在处理请求时占用物理资源相对较多，制约了网络应用在资源相对有限的情况下对请求的响应速度；因此，开发轻量级高性能 Web 服务器，保证网站迅捷稳定的工作，对互联网的进一步发展普及具有重要意义。

本论文选取轻量级的异步非阻塞多线程 Web 服务器作为研究课题，研究了一种可根据负载动态调节大小的线程池以及线程池任务调度算法并应用在Web服务器中，主要内容可归纳如下：

首先，通过对 HTTP、TCP 等重要计算机网络协议的详细介绍，以及对 Socket、进程线程、服务器 IO 模型等网络程序设计基础的学习，结合对 Nginx 服务器的深入分析，设计了一种轻量级的异步非阻塞 Web 服务器，服务器基于 Reactor 模式异步处理海量请求，利用线程池处理计算任务，占用资源少、运行速度快、部署方便，可以有效的应用在资源相对有限的网络环境中；

接着，介绍了线程池的基础技术以及设计思想，设计了一种可动态调节大小的线程池：当请求数量增加时，线程数量随之增加以满足系统性能的需求；当请求数量较少时，线程池销毁空闲线程以有效的利用系统资源。另外还研究了一种基于线程池的任务调度方法，可以将任务分配给最合适的线程进行处理；

最后，在 Linux 平台下对服务器的基础功能、系统容量、吞吐率以及延迟表现进行了测试，结合测试结果优化设计。测试结果显示本文设计的服务器支持基本的 Web 功能，运行稳定可靠，性能表现良好，满足设计要求。

论文创新点在于：研究了一款轻量级的异步非阻塞多线程 Web 服务器。基于 IO 多路复用与线程池技术实现，运行稳定可靠，性能表现良好；研究了一种可根据负载动态调节大小的线程池，以及一种基于线程池的任务调度方法。

关键词： Web 服务器；轻量级；IO 多路复用；线程池

Abstract

Web application has excellent versatility and great interactivity, thus becoming the first choice of the Internet access platform as the growing popularity and rapid development of Internet technology. However, most of the popularity Web server takes up too much physical resources when processing a request and restricts the site response speed of massive requests. Therefore, the development of lightweight high performance Web server to ensure fast and stable work at the time of limited resources is important to promote the further development of the popularity of Internet.

In this thesis, a lightweight asynchronous non-blocking Web server is selected as the research topic. A thread pool which can be resized dynamically based on the load had been designed and used in Web server. The main contents can be summarized as follows:

This thesis designed a lightweight asynchronous non-blocking Web server based on the detailed analysis of HTTP, TCP and other important Internet network protocols, as well as the analysis of network programming based on Socket, process thread and server IO model, combined with the depth study of Nginx server. The server can process requests asynchronously based on the Reactor pattern, and compute data tasks with thread pool. It can be effectively applied to the network environment with limited resources;

This thesis introduces the basic technology and design idea of thread pool, then design a thread pool with dynamically size: when the request increases, the number of threads increases to meet the demand of system load; when the request decreases, The thread pool recovers the excess threads to make efficient use of system resources. In addition, a task scheduling method based on thread pool is designed, In this method, the task can be submitted to the most suitable thread for processing;

In this thesis, the basic functions, system capacity, throughput and delay performance of the server are tested. The web server was optimization combined with the test results. Test result shows that the server can support the basic Web service functions, performance

stable and reliable, success to meet the design requirements.

The innovations for this work can be summarized as follows:

A lightweight asynchronous non-blocking Web server is implemented. The server perform stable and reliable based on IO multiplexing and thread pool technology;

Design a thread pool with dynamically size according to the load, and study a task scheduling method based on thread pool.

Key Words： Web server; lightweight; IO multiplexer; thread pool

目　录

第 1 章 绪论

1.1 课题背景及研究意义

随着互联网的迅猛发展与不断普及，网络应用的种类与功能也随之飞速增长。在线购物、实时聊天、网络视频等工具让人们的生活更加便利、娱乐方式更加丰富，用户只需通过打开浏览器访问一个简单的网站，就可以足不出户的知晓天下大事。如图 1.1，2016 年中国互联网络信息中心(CNNIC)的调查报告显示中国的网民人数已经超过了 7 亿[1]。而且根据摩尔定律：单位价格能买到的集成电路元器件数目每隔 18-20 个月便会翻倍，同时性能也随之翻倍，可知今后计算机硬件设备的性能将会持续增长、价格则继续降低，互联网的普及率也将随之不断提高。

图 1.1 近年中国网民规模与互联网普及率

作为 Web 应用的核心，Web 服务器的作用至关重要，传统的 Web 服务器对每一个请求都分配一个进程进行处理，用户点击网页将请求通过 Internet 发送至服务器，服务器接收请求进行处理后，接着控制数据库选取相关数据进行增删改查，再将处理结果通过 Internet 返回客户端，客户端收到数据后在浏览器上显示结果。这在互联网初期是非常有效的方法，但是在互联网高速发展的今天，Web 系统的日访

问量已经从百万次达到千万次，甚至亿次，Web 服务器承受的压力不断增加，这同时也带来了很多的问题：例如网页内容加载时间过长、网页无响应甚至 Web 服务器崩溃等等问题。2007 年 Google 公司的调查报告显示：Web 页面响应时间每增加 1 秒都会导致 4%的用户流失[2]。而在各个互联网公司都在抢占流量入口的今天，每减少 1%的用户都代表着上亿元的经济损失，这对任何一家企业都是不能承受的。所以，研究更高性能更快响应的 Web 服务器的任务变得尤其重要。

提升 Web 服务器性能最直观的方法就是增添硬件设施，这可以在一定程度上改善 Web 应用容纳用户的能力，然而，如果在主机资源相对有限的情况下，海量用户的访问会给 Web 服务器带来一系列性能问题。所以急需从软件架构的层面上设计一种具有良好可靠性的轻量级 Web 服务器，从而满足 Web 应用不断发展而导致的高性能需求。

1.2 国内外研究现状

自 20 世纪 80 年代互联网出现以来，安全可靠的 Web 服务器就保证了互联网企业的稳定工作与快速发展，而互联网行业的整体发展同时也促进着 Web 服务器技术的不断进步。

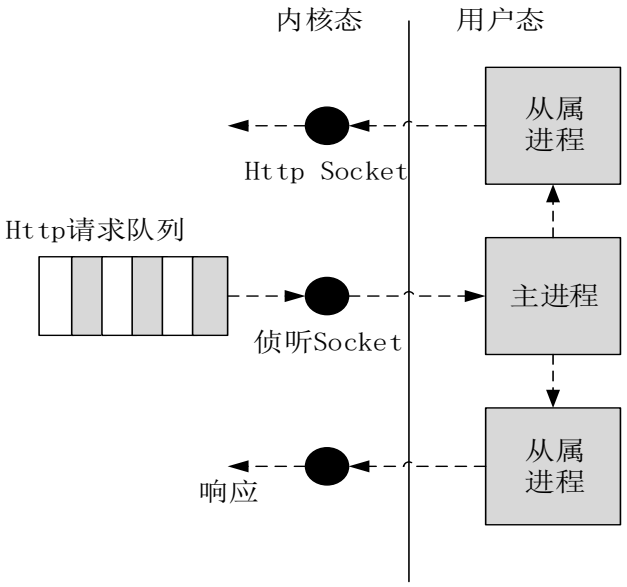

图 1.2 带主进程的多进程结构

如图 1.2，早期的 Web 服务器严格遵循着传统的 Unix 模型，每次收到新的请求都新建一个进程进行处理。整个服务器包含一个主进程与多个从属进程，主进程只负责接收 HTTP 请求，从属进程负责从主线程接收请求并进行具体的事务处理。典型的带主进程的多进程服务器有 NCSA 研发的的 httpd。

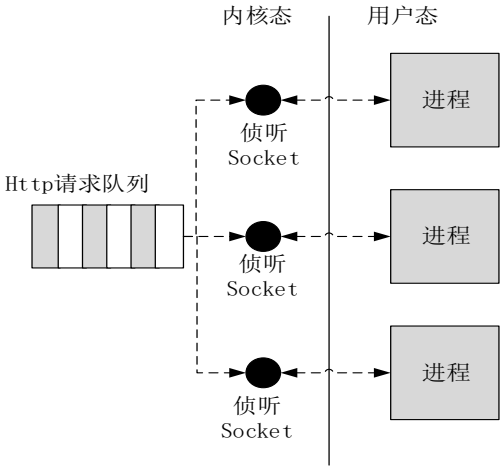

图 1.3 不带主进程的多进程结构

如今主流的多进程 Web 服务器不再区分主进程与从属线程，而是预先派生多个进程，等到 HTTP 请求到达的时候，操作系统将请求分配到一个正在侦听的进程上即可，进程调用 accept()方法接收 HTTP 请求，图 1.3 展示了这种结构。典型的多进程服务器有伊利诺斯州大学国家高级计算程序中心研发的 Apache[3]、Microsoft 公司主营的 IIS 等等[4]，其中 Apache 也是如今市场上应用最广的 Web 服务器。

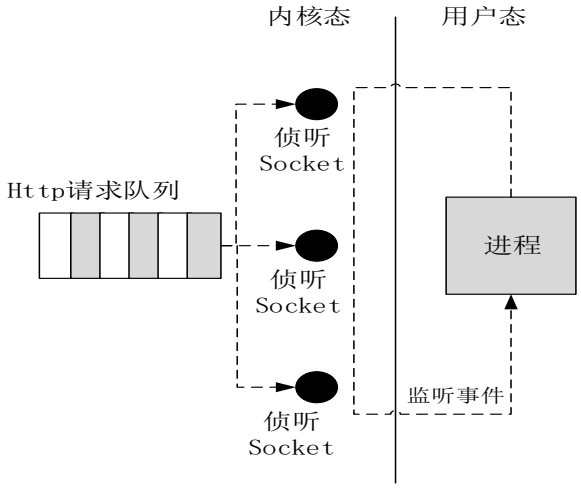

图 1.4 单进程事件驱动结构

由于并发进程的数量受到操作系统限制，而且进程之间切换的代价相对较高，所以多进程服务器在访问量较大的情况下表现并不理想。为了克服这些缺点，单进程事件驱动 SPED 服务器被设计出来。如图 1.4，SPED 服务器利用事件驱动机制，仅使用单个进程监听并处理所有的 HTTP 请求，避免了进程间的切换，在请求量较小的情况下可以得到满意的结果[5]。

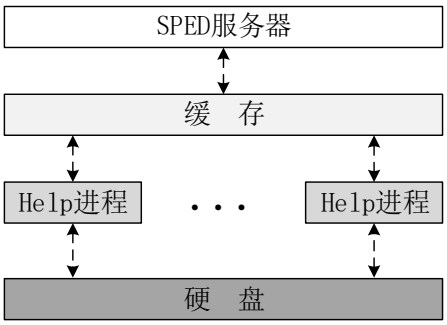

图 1.5 非对称多进程事件驱动结构

但是 SPED 服务器仍然存在着严重的缺陷：在冯洛伊曼计算机体系中， CPU 的指令周期比硬盘的 IO 周期小几个数量级，所以单进程服务器在等待硬盘 IO 的同时会导致其他请求阻塞。因此，Vivek S. Pai 在 SPED 服务器的基础上提出了 AMPED 架构，如图 1.5，AMPED 结构的设计思想是在主进程之外设立多个 Help 进程，Help 进程会在主进程工作的同时把可能命中的文件内容读入缓存[5]，这样主进程就可以直接在缓存中读取数据，从而减少主线程直接进行磁盘 IO 引起的等待时间。

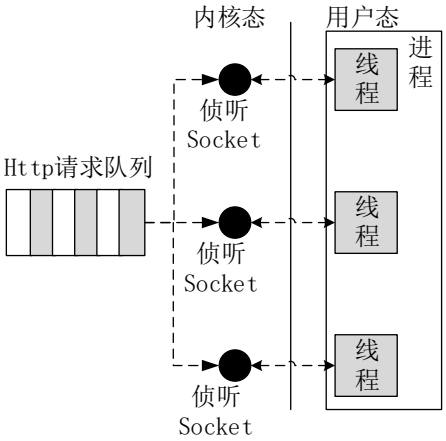

图 1.6 多线程服务器结构

随着互联网用户的继续增多，许多服务器的日访问量已经达到百万次以上，此时使用单进程服务器会导致大量请求发生阻塞。同时，随着多核处理器的出现，为了尽可能地利用服务器的硬件性能，开发人员参考多进程结构提出了多线程结构。如图 1.6，相比于进程，线程共享着同一进程的资源，是 CPU 资源调度的最小单位，线程上下文之间切换的代价更小、速度更快，每一个服务器线程都能处理一个 HTTP 请求。典型的多线程服务器有俄罗斯研究员 Igor Sysoev 开发的 Nginx[6]，德国领导开发的 Lighttpd 等[7]。

通过对多进程服务器与多线程服务器对比可以发现，多进程服务器运行更为稳定健壮，方便操作系统监控管理，能充分利用多核 CPU 实现并行处理。但是存在着内存消耗较大，上下文切换代价高，IO 并发处理能力较低等等缺点；而多线程服务器内存消耗小，上下文切换较快，IO 并发能力强，同样能充分利用多核 CPU 实现并行处理。但是不方便操作系统管理，对共享资源管理要求较高，所以需要根据具体的应用场景进行选择。

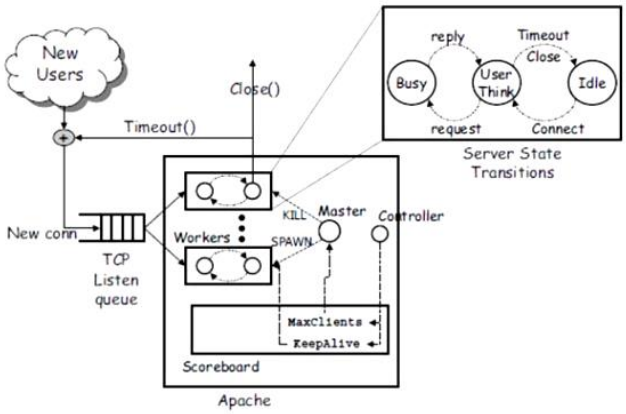

图 1.7 Apache 架构图

此外，Web 服务器还可以根据软件大小、安装方式、硬件开销等因素分为轻量级与重量级两种，一般将耦合度较高，软件安装包较大的服务器称作重量级服务器，而将安装简便、硬件要求较低的服务器称作轻量级服务器。典型的重量级服务器如 Apache 与 Tomcat 使用进程对每个请求进行处理[8]。当用户量增多时，处理进程大量增加会导致服务器 CPU 使用率与内存占用率较高，所以称之为“重量级”，图 1.7

展示了 Apache 的基本架构；与之相对的轻量级服务器如 Nginx 与 Lighty 都是基于事件驱动的服务器，进程不必耗费大量时间等待请求完成，只有当连接状态改变时进程才会继续处理请求，只需一个进程就可以处理大量连接[9]，所以这类服务器被称作“轻量级”。

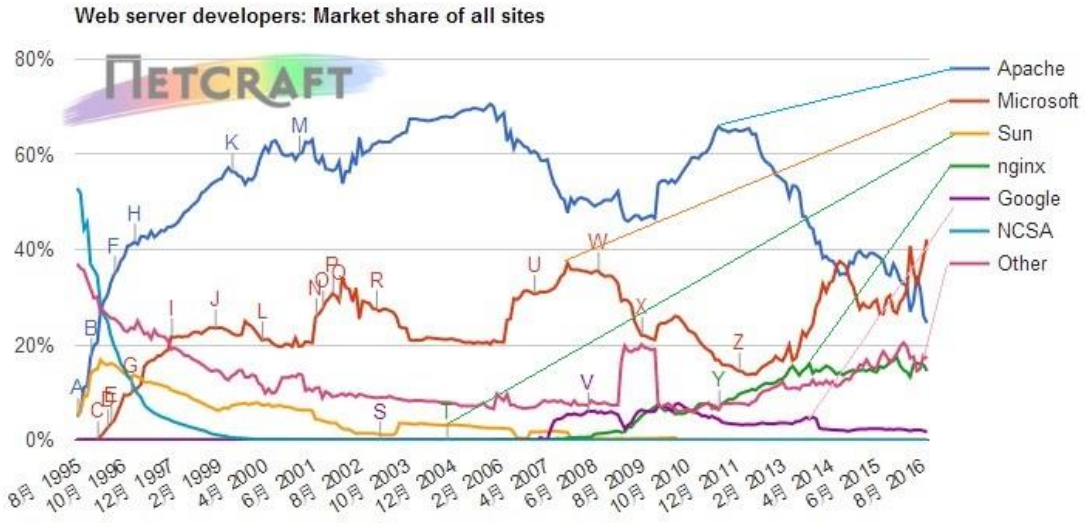

图 1.8 各服务器市场占有率趋势图

如今多线程服务器的市场占有率正在不断攀升，淘宝、网易、优酷、Facebook、GitHub 等国内外知名站点都搭建了基于 Nginx 的服务器框架[10]。NetCraft 提供了关于服务器的权威信息，图 1.8 显示了截止到 2016 年的服务器市场占有率变化，可以看到近年来 Apache 等多进程服务器的份额已经大幅降低，而 Nginx 等多线程服务器份额则持续上升。最新服务器占有率数据如表 1.1 所示。

表 1.1 各服务器市场占有率数据统计

Developer	August 2016	Percent	September 2016	Percent	Change
Microsoft	445,105,755	38.58%	542,498,796	42.19%	3.61
Apache	300,028,832	26.01%	316,042,289	24.58%	-1.43
Nginx	181,606,297	14.74%	186,529,038	15.51%	1.23
Google	22,111,431	1.92%	21,467,729	1.67%	-0.25

1.3 论文主要内容及创新点

本论文主要研究了一种轻量级高性能的多线程 Web 服务器的原理，设计以及实现。课题主要内容包括以下三点：

（1）研究了 HTTP，TCP 等重要 Web 通信协议，学习了网络程序编程基础，对如今流行的轻量级服务器 Nginx 的架构进行了深入剖析。最后研究了一种高性能的多线程 Web 服务器，基于 Reactor 模式可以满足在有限资源条件下的高性能表现，支持半同步半异步模式，其中 IO 请求由 Epoll 调用异步进行处理，线程池同步处理计算请求，配置简单，功能完备。

（2）学习了线程池的基础技术以及设计思路，设计了一种可动态调节大小的线程池：当系统负载增加时，线程数量随之增加以满足性能需求；当负载持续减少时，线程池销毁过剩的线程，有效的利用系统资源。另外研究了一种基于线程池的任务调度方法，可以将任务递交给最合适的线程进行处理。

（3）完成了服务器代码的最终实现并对其基础功能、系统容量、吞吐率以及延迟等指标进行了测试，结合测试结果优化设计。测试结果显示服务器能够支持基本的 Web 服务功能，运行稳定可靠，性能表现良好，满足设计要求。

课题创新点在于：研究了一种异步非阻塞的多线程服务器，支持异步处理网络 IO 请求，同步处理计算任务，在资源相对有限的情况下可以保证良好的性能表现；研究了一种动态线程池模型，可以根据系统当前负载动态调节线程池大小，有效的利用系统资源；提出了一种基于线程池的任务调度方法，方案可行，表现良好。

第 2 章 Web 服务器理论与研究

Web 服务器通过网页的方式为用户接入互联网，当用户向 Web 服务器建立连接并发出请求时，Web 服务器会接收数据并执行相应的业务处理，接着遵循既定的网络协议响应用户请求，最后以文件、图像、表格和脚本的形式构成网页内容返回客户端浏览器显示。如今几乎所有的企业或机构都拥有自己独立的网站，公司利用互联网的便捷创造了大量价值，其中 Web 服务器是业务稳定运行的核心保证。本章旨在介绍 Web 服务器的基本原理、典型架构及其遵循的网络协议。

2.1 计算机网络协议分析

网络协议是计算机网络中进行数据交换而约定的标准，它规定了通信时数据遵循的格式和数据的意义。只要遵循相同的网络协议，用户与服务器之间就可以进行信息交互，实现网络通信。

2.1.1 HTTP 协议

作为 Web 技术的核心，HTTP 超文本传输协议是无连接的，每次传输数据都需要建立连接、发送请求、接收请求、最后断开连接，即每次完成请求后服务器就不再耗费资源保留该请求的内容；同时 HTTP 协议也是无状态的，不同连接之间互不相关，所以 HTTP 响应速度快，通信效率高。但是一些需要在前后请求中共享的数据则必须利用其他方法进行保存，这势必会导致额外的数据传输[11]。

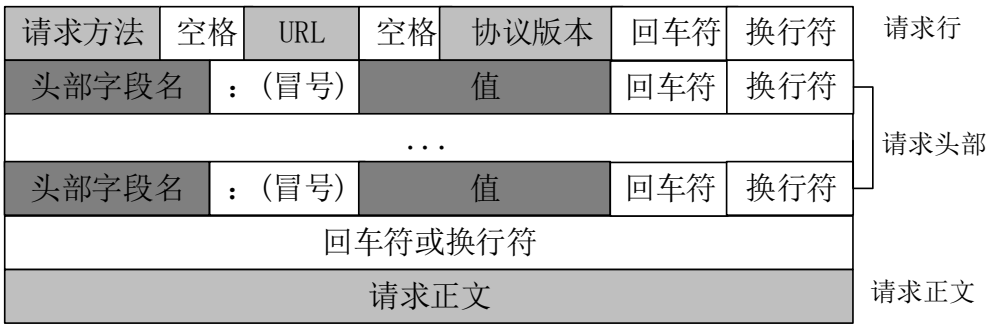

图 2.1 HTTP 请求报文格式

HTTP协议的主要工作包括发送请求与接收响应，如图2.1展示了HTTP请求报文的一般格式，包括请求行、请求头部、空行和正文4个部分[12]。

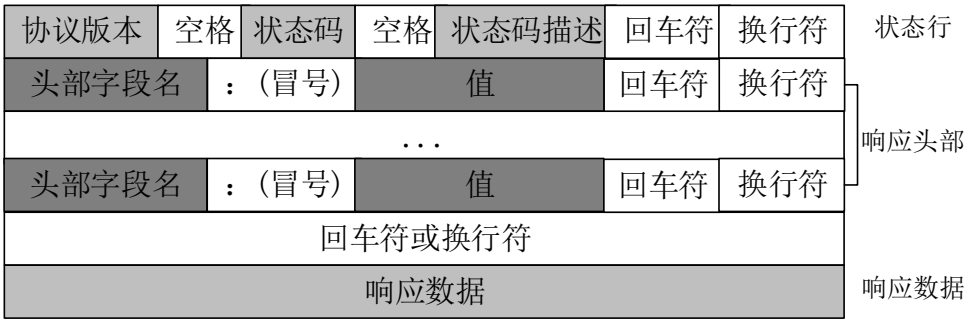

图 2.2 HTTP 响应报文格式

如图2.2，HTTP响应报文同样是由状态行、消息报头、空行与正文四部分组成。其中状态行包括HTTP协议版本，响应状态代码，状态代码的文本描述三部分。状态代码只有三位，百位表示响应的类别，表2.1展示了HTTP响应状态码的5种取值[13]。

表2.1 HTTP响应状态码

1xx	指示信息：表示请求已接收，继续处理。
2xx	成功：表示请求已被成功处理
3xx	重定向：请求需要进一步的操作
4xx	客户端错误：请求错误或无法实现
5xx	服务器端错误：服务器未能响应请求

2.1.2 TCP 协议

TCP传输控制协议是一种面向连接的、可靠的传输层通信协议[14]。它工作在计算机网络模型中的传输层，主要负责主机之间的可靠连接与拥塞控制等功能。当传输层接收到了应用层发送的HTTP报文后，TCP协议将数据流分割成合适大小的报文段，之后再将报文传给网络层进行网络寻址传输。

在 TCP 建立连接的过程中，服务器进程被动打开并处于监听状态，准备接收客户进程的请求，如果接收到客户进程的请求则做出响应；客户进程则是主动开启，

绑定本地 IP 地址和端口尝试连接服务器。在成功建立连接之后服务器便可以与客户进行可靠数据传输，等待数据传输完毕后释放连接。TCP 连接的建立与销毁的过程如图 2.3。

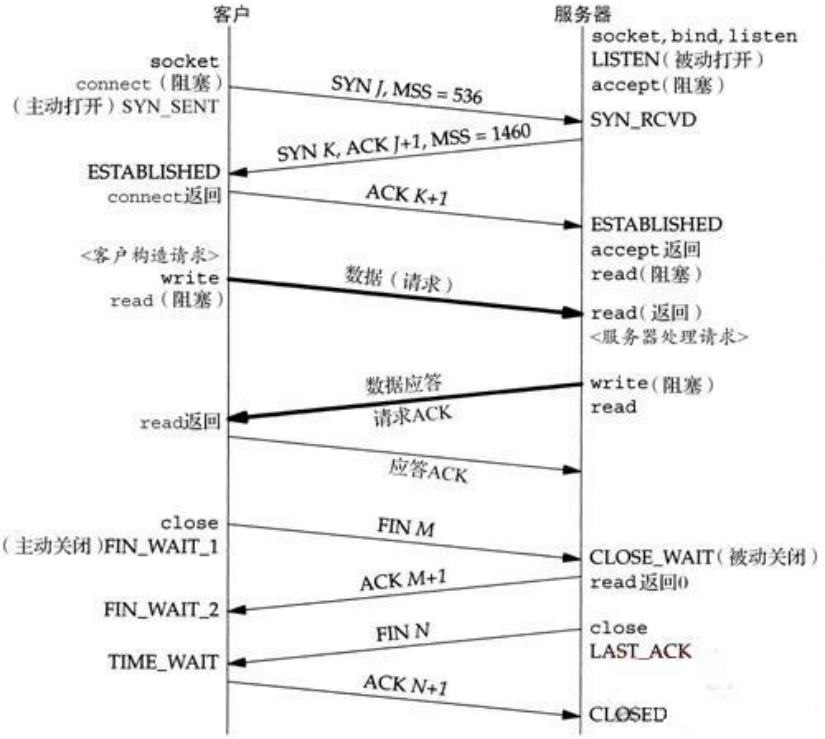

图 2.3 TCP 连接的建立与断开

在计算机网络中，如果发送方数据发送速率超过接收方的接收速率，接收方就会丢失部分数据，同时过多的报文会导致网络拥塞。流量控制就是利用滑动窗口机制，让发送报文的序号与响应报文的序号差值不超过窗口大小，以保持发送速率接收速率相当。

拥塞控制则是保证注入网络的数据不超过网络负载上限，相比于流量控制是端到端的通信流量控制，只需控制发送端发送数据的速率不超过接收端的接受速率即可，拥塞控制则需要考虑整个网络中所有的主机以及其他影响因素[14]，以保证网络每个部分正常负载工作，所以拥塞控制是针对整个网络的全局性的控制。

2.2 网络程序设计基础

2.2.1 Socket 编程

Socket 的概念起源于 Unix，而在 Unix/Linux 操作系统中“一切皆文件”，即所有事物都可以用打开、读写、关闭等方法来进行操作。Socket 就是该模式的一种实现方式，Socket 可以看作一种特殊的文件[15]，其中一些 Socket 函数可以对其进行操作（读/写 IO、打开、关闭）。

Socket 地址主要由地址协议族 sa_family_t，端口号 in_port_t 和网际地址 in_addr 几部分组成。

```
struct sockaddr_in {
    sa_family_t    sin_family;    /* address family: AF_INET */
    in_port_t      sin_port;      /* port in network byte order */
    struct in_addr sin_addr;      /* internet address */
};
/* Internet address. */
struct in_addr {
    uint32_t       s_addr;        /* address in network byte order */
};
```

Socket 的本质是网络编程接口 API，是对 TCP/IP 可靠网络传输的封装，客户端和服务器都可以从 Socket 套接字发送或接收 HTTP 消息，图 2.4 展示了客户端与服务器通过 Socket 建立连接的过程。首先新建一个未定义的 Socket 套接字，接着通过 bind()函数将 Socket 返回的文件描述符 fd 与某个具体的网络地址和端口绑定，并返回函数调用的结果，这样就完成了套接字的初始化；然后再使用 listen()函数监听 Socket 套接字，当监听到客户端有请求发出时，通过 accept()函数可以接收客户端的请求并进行处理。

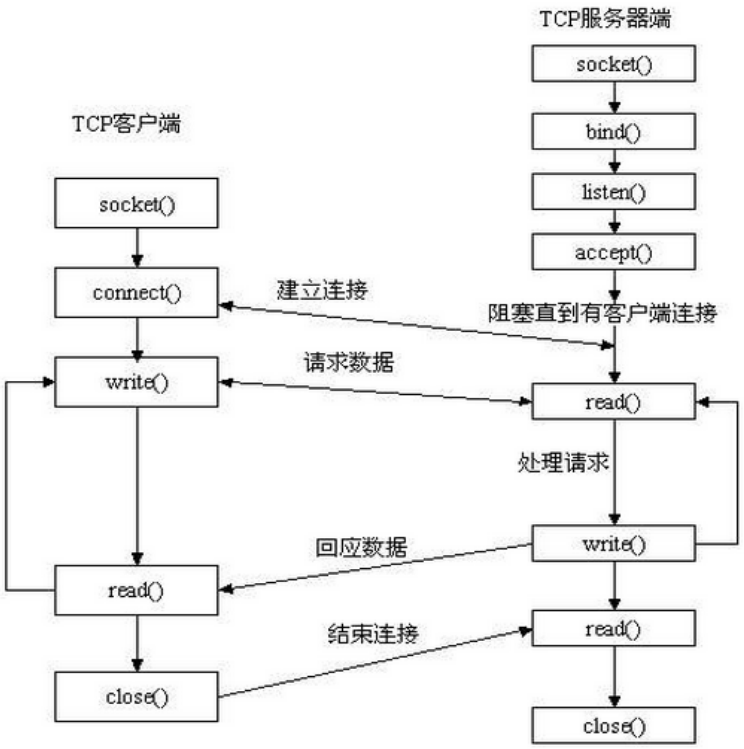

图 2.4 Socket 连接过程

2.2.2 进程与线程

进程是程序在数据集合上的运行过程，是系统资源分配的最小单位[16]。进程可以看作程序的执行实体，也可以看作线程的容器。程序可以看作是数据、指令与逻辑的描述，而进程则可以看作程序的一次执行过程。

线程则是程序执行的最小粒度，是 CPU 调度和分派的最小单位。一个进程中有多个独立的线程，线程之间可以并发执行，如果进程只有一个线程，则线程就是进程本身。线程自身并不拥有系统资源，而是与其他线程共享着进程占用的系统资源。

相比于进程，线程的创建与销毁时间都大幅缩短，但是如果服务器频繁执行时间较短的任务的话，需要不断地创建并销毁线程，这同样会带来大量的时间与资源浪费。线程池则是预先分配线程资源的技术，在线程池初始化的时候会创建一定的线程等待，这些线程处于睡眠状态，只占用较少的内存而不消耗 CPU 资源。当请求到来的时候，如果有空闲线程，则线程池给请求分配一个空闲的线程进行处理；若

线程池中所有线程都处于运行状态，则线程池需要新建一定数量的线程。当系统负载较低的时候，可以通过移除一直处在睡眠状态的线程来减少内存占用。线程池技术不仅减少了新建与销毁线程的开销，也方便了服务器对线程进行管理。

值得注意的是，线程池的实现可以分为线程数量固定与动态变化两种。线程数量固定的线程池在系统初始化的时候就会新建一定量的线程并在之后的工作中不再变化。当请求数超过线程数目时，剩余的请求只能等待，当请求数少于线程数目时，多余的线程也不能销毁,所以固定数量的线程池在负载变化较大的情况下表现不佳；而线程数动态变化的线程池则会根据当前系统的负载来动态调节线程数目，在系统负载较高时，线程池会增加一定的线程来处理请求，当负载较低时，线程池又会销毁多余的线程以避免资源浪费。其中，线程数目的动态计算成为了线程池设计的重点与难点。

2.2.3 服务器 IO 模型

轻量级高性能的服务器设计需要构造高性能的 IO 模型，常见的 IO 模型可以分为同步与异步，阻塞与非阻塞几种。

同步和异步的区别在于用户线程与内核的信息交互方式：如果用户线程发起请求后被阻塞，在内核 IO 操作完成后才能继续执行，这种 IO 方式称为同步 IO；如果用户线程发起 IO 请求后不阻塞，转而执行其他请求，当内核 IO 操作完成后再切换到原请求，这种 IO 方式称为异步 IO。

阻塞和非阻塞的区别则在于用户线程调用内核 IO 的方式：需要等待 IO 请求彻底完成才能返回的 IO 方式称为阻塞 IO；调用 IO 请求后立即返回用户空间，无需等到 IO 请求完成的方式称为非阻塞 IO。

1. 同步阻塞 IO

同步阻塞 IO 模型中用户线程进行 IO 操作时被阻塞等待，是最基础的 IO 模型。如图 2.5 所示，用户线程调用 read()函数发起 IO 读操作并进入内核空间。此时用户线程是被阻塞的。内核将数据准备完毕之后拷贝到用户空间，完成 read 操作，之后用户线程才能恢复运行状态，所以同步阻塞 IO 模型对 CPU 的资源利用率不高。

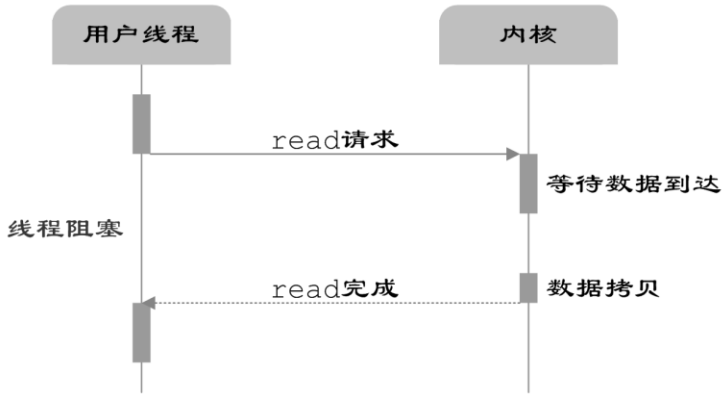

图 2.5 同步阻塞 IO 模型

2. 同步非阻塞 IO

如图 2.6，在同步阻塞 IO 模型的基础上设置 Socket 为 NONBLOCK 就是同步非阻塞 IO 模式，用户线程发起 IO 请求后立即返回，但没有获取到完整数据，所以需要用户线程不断重复发起 IO 请求，消耗大量资源，直到数据结果返回后，才继续执行下一步操作，一般较少使用。

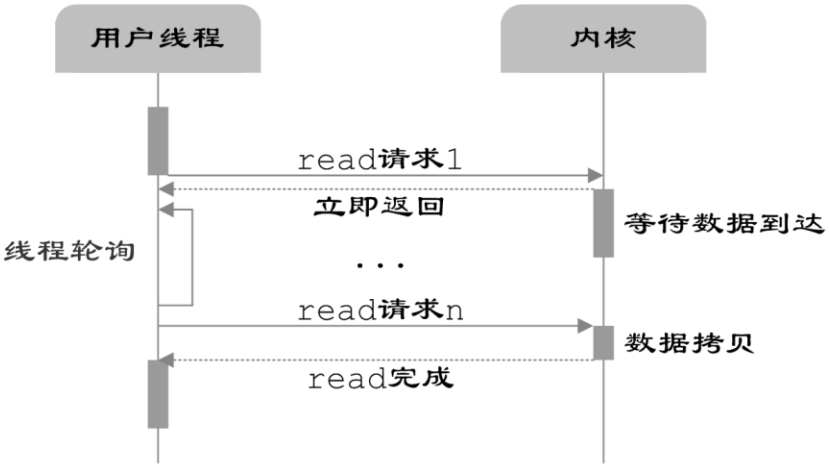

图 2.6 同步非阻塞 IO 模型

3. IO 多路复用

IO 多路复用模型的基础是多路复用函数 select()，select 函数是由操作系统内核提供的，可以避免同步非阻塞 IO 模型中不断重复请求的问题[17]。如图 2.7 所示，使用 IO 多路复用模型的服务器中，一个线程内可以同时处理多个 IO 请求。用户向 Reactor 中注册多个 Socket 连接之后便可以立即返回处理其他请求，Reactor 不断地

调用 select()函数查询内核，有结果时即通知用户线程处理。Linux 中的 Epoll 就是属于 IO 多路复用。

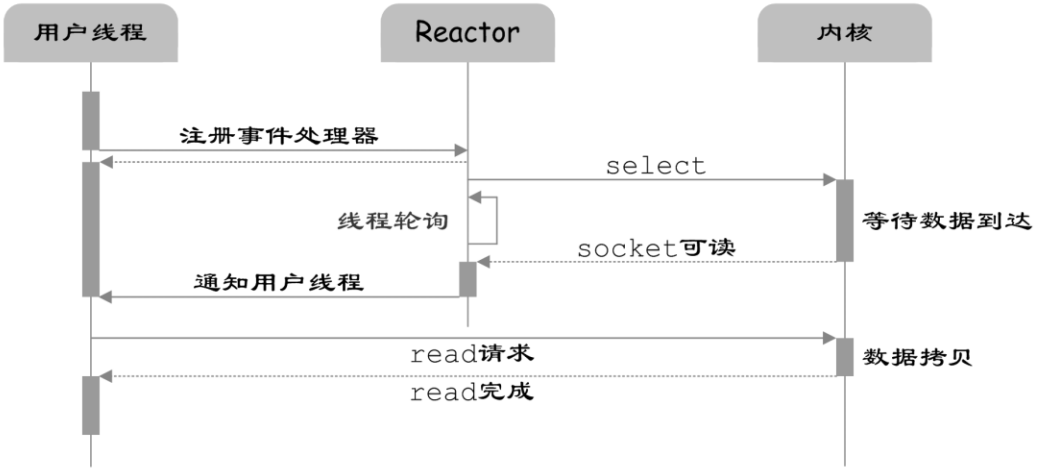

图 2.7 IO 多路复用模型

4. 异步 IO

异步 IO 模型的实现同样需要操作系统的支持。在 IO 多路复用模型中，当内核通知用户线程处理事件时，用户线程需要自行读取、处理数据。如图 2.8，在异步 IO 模型中使用了 Proactor 设计模式，用户线程直接向内核发送异步 read 请求，之后便立刻返回，内核会完成读取数据等工作并拷贝到用户线程的缓冲区，当用户线程收到通知时直接使用即可。

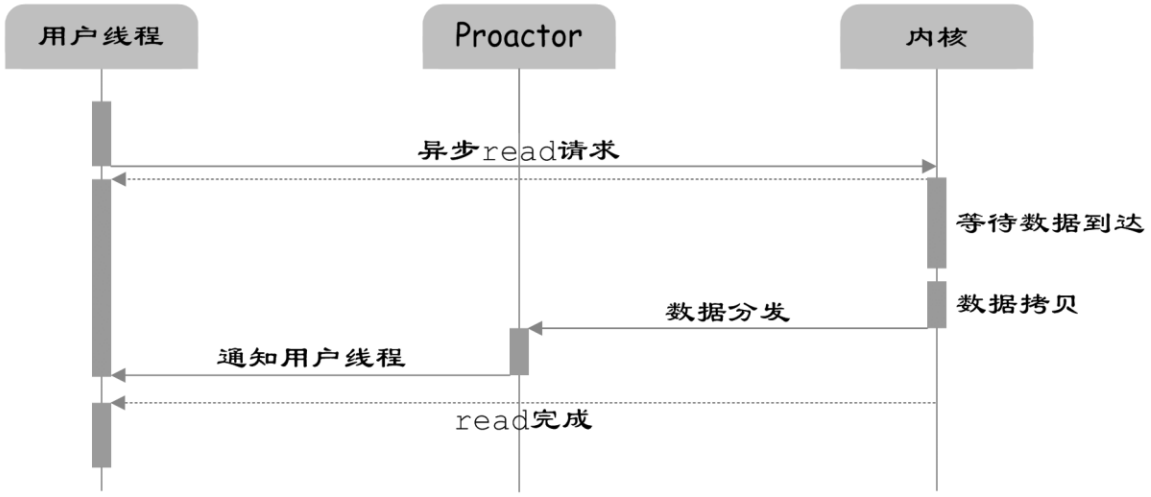

图 2.8 异步 IO 模型

2.3 Nginx 架构研究

Nginx 是 2004 年发布的一款免费开源 Web 服务器软件，是由俄罗斯软件工程师 Igor Sysoev 开发并维护的[18]，具有高性能，高并发和低内存消耗等一系列优点，适用于负载均衡、缓存、带宽控制等各种应用中。正是因为 Nginx 的这些优点，现在已经广泛应用在互联网中，使用率仅次于 Apache，所以对 Nginx 的研究有助于学习轻量级高性能 Web 服务器的设计。

在 Unix 系统中，Nginx 启动后会以守护进程的方式在后台运行，如图 2.9 是 Nginx 服务器的架构图，Nginx 支持多进程与多线程工作方式，默认以多进程方式工作，守护进程会 fork 出一个 Master 进程和多个 Worker 进程[19]，Master 进程的工作主要包括：接收信号并发送信号至 worker 进程，监控并管理各个 Worker 进程，异常情况重新启动 Worker 进程等等工作，并不负责对信号的处理。各个 Worker 进程之间相互独立对等，用户只需要与 Master 进程通信，之后请求会分配到一个 Worker 进程进行处理。

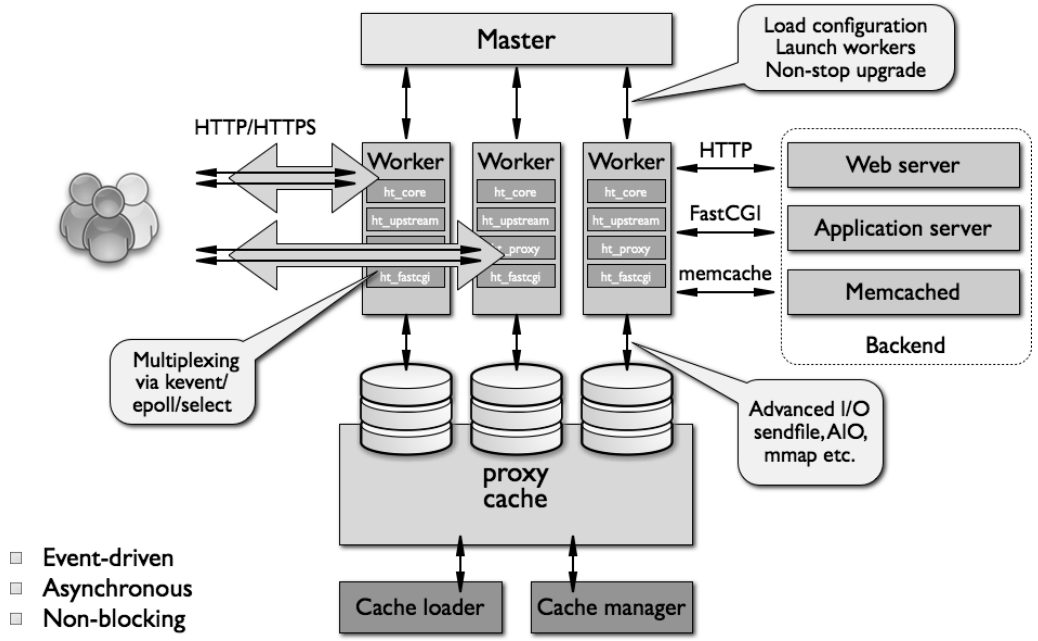

图 2.9 Nginx 架构模型

Nginx 默认采用 pre-fork 方式工作，最开始只有 Master 进程运行，Master 进程初始化之后建立 Socket 套接字并绑定监听，然后再 fork 出多个 Worker 进程，每个 Worker 进程都同时监听同一个 Socket 套接字。为了保证同一时刻只会有一个进程建立连接，Nginx 提供了一个 accept_mutex 共享锁，各个 Worker 进程不断竞争 accept_mutex 共享锁，只有当一个 Worker 进程获取到共享锁之后才可以开始对请求进行处理，并将结果返回给客户端，最后断开连接，这样 Nginx 就实现了一个完整的网络请求。

Nginx 采用了多路 IO 复用的模式来处理请求，基于 Linux 系统支持的 Epoll 调用同时监控多个事件。Epoll 可以设置阻塞的超时时间，如果有事件在超时时间之内准备好就直接返回结果给进程；若事件需要等待则注册到 Epoll 中，等事件完成之后 Epoll 再通知进程进行处理，进程可以去处理其他工作而不必阻塞在当前事件上，这样就可以实现处理大量的并发。需要注意的是，一个服务器进程只能同时处理一个请求，当异步事件未准备好而主动让出进程时，由于 CPU 速度太快，进程在请求间进行不断地切换可以看作同时处理多个请求[19]。

Nginx 默认是以多进程单线程的方式进行工作，与简单的多线程服务器相比，Nginx 不用创建或管理线程，也不存在上下文切换，可以避免线程抖动和锁的开销。并发数增加只是会占用更多的系统资源而已，在如今硬件过剩的环境下完全可以接受。同时，Nginx 中每个 worker 进程绑定一个核心，也可以很好的利用多核 CPU。

用伪码来表示 Nginx 的事件处理模型就是：

```
while (true) {
    for t in run_tasks:
        t.handler();
    update_time(&now);
    timeout = ETERNITY;
    for t in wait_tasks: /* sorted already */
        if (t.time <= now) {
            t.timeout_handler();    //处理超时任务
```

```
            } else {
                timeout = t.time - now;       //更新事件时间
                break;
            }
        nevents = poll_function(events, timeout);
        for i in nevents:
            task t;
            if (events[i].type == READ) {
                t.handler = read_handler;
            } else (events[i].type == WRITE) {
                t.handler = write_handler;
            }
            run_tasks_add(t);
    }
```

2.4 本章小结

本章首先对应用在服务器中的关键网络协议进行了介绍，主要包括HTTP和TCP两种协议；接着对网络程序设计基础进行了分析，如Socket编程通信、进程、线程与线程池的概念、各种服务器 IO 模式等；最后参考了当今最流行的轻量级服务器Nginx的架构设计，学习了Nginx实现 IO 多路复用的方法。总之，本章介绍了 Web 服务器的关键原理与技术，是进一步设计轻量级高性能 Web 服务器的基础。

第 3 章 Web 服务器的需求分析与设计

一般情况下研发人员可以从响应时间、可靠性、可伸缩性、安全性、灵活性、平台兼容性和易管理性等指标来评价一个服务器的性能[20]，一个优秀的 Web 服务器应该尽可能在这些指标上做得更好。而对于轻量级 Web 服务器来说，则更侧重于响应速度、资源占用、灵活性等方面。本章遵循软件工程的思想，首先对高性能轻量级 Web 服务器进行了需求分析，然后基于需求分析进行了总体架构设计，最终成功完成了一种轻量级的 Web 服务器的模块设计。

3.1 高性能 Web 服务器需求分析

1.参数动态配置

在大部分 UNIX 服务器上，用户的指令都是通过命令行来交互实现的[21]，服务器一般需要在启动之前完成具体的配置，所以需要设计服务器在启动时，通过命令行输入对一个或多个参数实现服务器的动态配置，否则以默认方式进行启动。

命令行参数配置的格式如下：该命令表示服务器启动 50 个线程监听 1024-65555 的端口，管理线程为 1023。

Server –start 1024 65555 50 1023

当然，服务器也应该支持 conf 配置文件进行动态配置，具体格式与 JSON 兼容，即每个键值对表示一项配置，中间以冒号分隔。

startPort：1024　　#起始监听端口

2.支持并发连接

多用户同时访问是 Web 服务器的基本要求之一，也一直是服务器设计的重点和难点。2001 年著名的 C10K 问题被提出[22], 16 年后的今天, 既得益于软件的进步, 也得益于硬件性能的提高，现有的语言和库可以轻松地写出 C10K 的服务器. 现在, 该是考虑 C1000K(百万连接)甚至 C10M（千万连接）的问题的时候了。但是高并发访问会带来一系列的问题，如硬件资源不足，资源竞争导致网页响应时间过长甚至崩溃等问题，这都是设计高性能服务器所需要考虑的。

3.HTTP 协议支持

在 RFC2145 协议中描述了有关 HTTP 版本的信息。HTTP/0.9 只接受 GET 方法，现在已经很少使用，现在主流的 HTTP 协议版本有 HTTP/1.0 与 HTTP/1.1 版本，HTTP/1.0 一般使用在代理服务器中，HTTP/1.1 版本是目前最流行的版本，默认采用持久连接，本文设计的服务器只需要支持最流行的 HTTP/1.1 即可，当请求协议版本不兼容时应该通过状态码给用户返回对应的错误信息。

4.请求方法支持

HTTP/1.1 共定义了八种请求方法，包括 GET、PUT、POST、DELETE、HEAD、TRACE、OPTIONS、CONNECT，每种请求方法对应着不同的功能。对于本文设计的轻量级 Web 服务器来说，主要任务是提供静态或动态 Web 页面服务，所以 GET 和 POST 请求方法是必须支持的，其他方法则为可选。

5.支持线程池

使用线程池可以减少在创建和销毁线程上所花的时间以及系统资源的开销，而且更便于系统对线程的管理，本文旨在设计一种多线程轻量级服务器，如不使用线程池，有可能造成系统创建大量线程而导致耗尽系统内存。

6.日志功能

日志功能可以给开发者和用户提供管理服务器和监视服务器等手段，当服务器有一定活动时，日志系统可以把该活动记录下来并写入日志中，等到服务器发生异常事件时就能够从日志中找出并解决问题。服务器需将日志保存在以当日期命名的日志文件当中，记录着服务器接收、处理请求以及运行错误等各种信息。

7.安全性保证

OpenSSL 是一个安全套接字层密码库，封装了密码算法、常用的密钥和证书封装管理功能及 SSL 协议[23]，对 OpenSSL 的支持可以保证应用间的保密性和可靠性，能使用户与服务器之间的通信不被攻击者窃听。

8.平台兼容性支持

虽然绝大部分的服务器是运行在 Linux 系统上的，但是仍有一定数量的服务器工作在 Windows 与 Mac OS X 系统上，优秀的服务器应该保证对平台兼容性的良好

支持，保证在不同操作系统平台下都能正常运行。

3.2 Web 服务器总体设计

本文的主要任务是设计一个高性能的轻量级 Web 服务器并编写代码实现，在此基础之上研究一种动态大小的线程池，以及用于线程池的任务调度算法，以满足在硬件资源有限的前提下尽量满足更多的用户请求。如图 3.1，根据 3.1 节对高性能轻量级服务器的需求分析，本文设计的的 Web 服务器将划分为 6 大模块：

（1）服务器初始化模块：启动服务器并进行初始化，包括服务器参数的配置与 Socket 套接字的初始化等等工作；

（2）TCP 连接模块：处理服务器与客户端之间的网络连接与断开事宜，包括 Socket 连接断开与接收发送 TCP 报文数据等工作；

（3）HTTP 协议模块：根据客户端的请求，处理并返回相应的 HTTP 响应报文；

（4）IO 多路复用模块：以异步非阻塞 IO 模式保证多用户对服务器的并发访问；

（5）线程池模块：使用线程池调度管理线程，避免过多的新建与销毁线程带来的额外开销[24]；

（6）辅助模块：包括日志模块，文件上传下载模块，字符串处理模块，网络字节序处理模块等等；

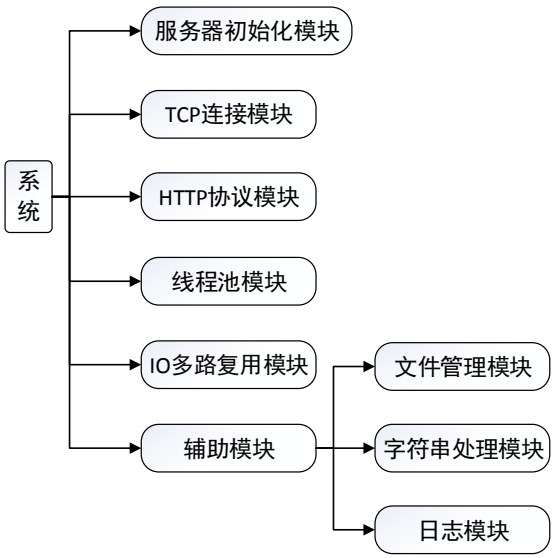

图 3.1 服务器总体设计图

3.3 Web 服务器主要模块设计

3.3.1 TCP 连接模块

著名的 Nginx 服务器只能作为一个 HTTP 服务器对 HTTP 报文进行处理，如果要处理 TCP/IP 协议，则需要安装第三方的 TCP-Proxy-Module 模块[25]。本文设计的服务器直接对传输层进行处理，在 TCP 连接的基础上添加对 HTTP 请求响应报文的处理就形成了一个 HTTP 服务器。

TCP 连接模块主要负责处理客户端与服务器之间 TCP 连接的建立与断开过程，包括接收 TCP 请求，为 TCP 连接分配资源、接收 TCP 报文数据，发送 TCP 报文数据、断开 TCP 连接，撤销连接占用的资源等工作。TCP 协议模块处理流程图如 3.2。

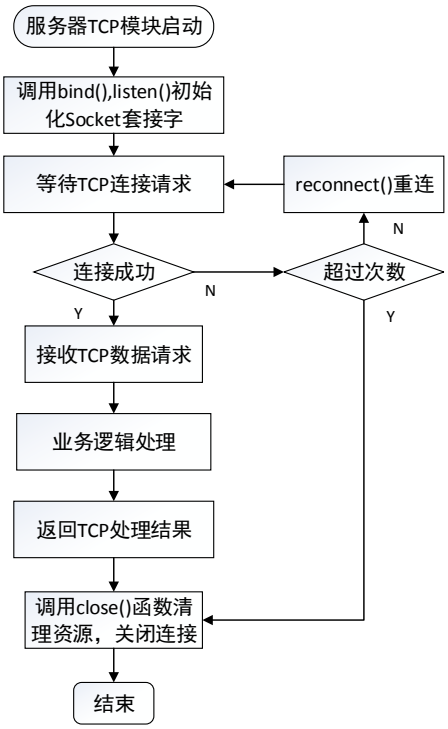

图 3.2 TCP 连接流程图

3.3.2 HTTP 协议模块

HTTP 协议模块是 Web 服务器的核心，在本文设计的服务器中，HTTP 连接本

质上是在 TCP 连接的基础上加入了对 HTTP 请求与响应的处理。在 TCP 连接成功建立并开始传递数据之后，TCP 协议模块会将接收到的数据放入缓冲区 buffer 中，HTTP 模块则负责读取数据、调用字符串处理模块按照 HTTP 协议解析数据，接着将有效信息传递给业务逻辑单元进行处理，待处理完成之后模块再遵循协议将响应信息封装成一个完整的 HTTP 报文，最后提交给 TCP 连接模块返回客户端。图 3.3 展示了 HTTP 协议模块处理的一般流程。

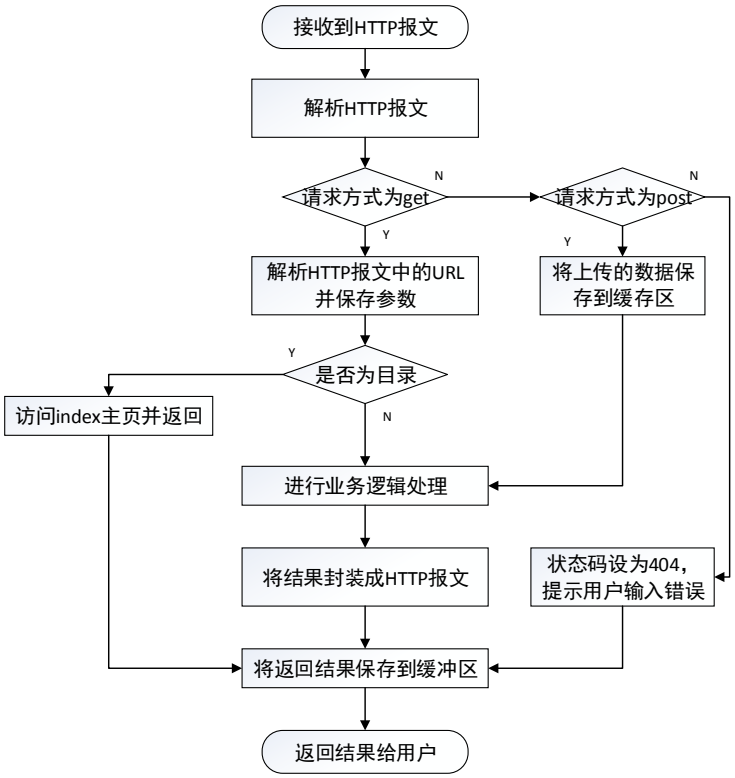

图 3.3 HTTP 请求处理流程图

3.3.3 高性能 IO 模块

对于多线程服务器来说，常见的 IO 处理模型有 3 种：

（1）对每个到来的请求都新建一个线程进行处理，每个线程都使用阻塞式 IO 操作。但是阻塞式 IO 方式中线程大部分时间都在等待内核 IO 完成，占用的 CPU 资源并没有工作，所以如果线程一直阻塞在事件上等待返回结果的话会导致服务器资源的浪费。

（2）在阻塞式 IO 操作基础上使用线程池。线程池能够很好的管理线程，减少

线程频繁新建销毁的开销，性能相比第一种模型有所提高，但碍于阻塞式 IO 的特性，在多线程服务器中依然表现不理想。

（3）非阻塞 IO 结合 IO 多路复用技术工作。Java 中的 NIO、Linux 下的 Epoll 机制都是这种方式[26]，在 Epoll 中线程可以在等待 IO 的时候转而处理其他任务，当请求被内核 IO 事件异步唤醒就会通知线程进行处理，这样就能满足服务器异步 IO 请求的条件。另外 Epoll 的 IO 效率不会随着监听事件数目的增长而降低，特别适合使用在高并发访问环境下。但是对于计算任务密集而 IO 操作较少的请求来说，本方法较为浪费资源。

本文基于 Reactor 思想研究了一种新的服务器处理模型，以保证服务器在海量并发访问下的性能表现：使用 IO 多路复用结合非阻塞 IO 方式处理请求中的 IO 部分内容，使用线程池结合任务队列处理请求中的业务逻辑操作。具体示意图如 3.4：服务器开启一定的 IO 线程与一个线程池，IO 线程只负责处理网络连接与网络 IO，线程池负责具体的业务逻辑操作。建立连接后 IO 线程将读写事件注册到 Epoll 中转而处理其他请求，这样当某个连接的 Socket 上有可读事件的时候，Epoll 会唤醒该 IO 线程进行读取，读取到的数据被添加到任务队列中，随之由线程池负责从任务队列中取出数据进行具体的业务逻辑处理；待处理完毕之后 Socket 上会出现可写事件，同样的，Epoll 会唤醒线程发送数据。

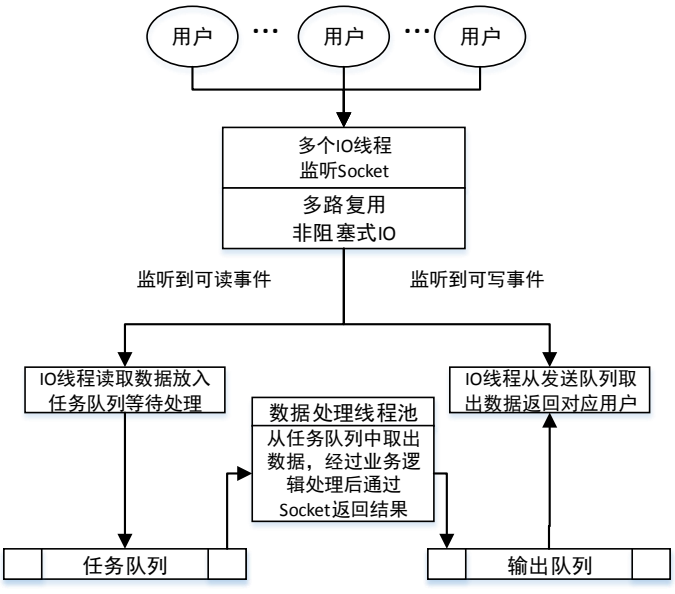

图 3.4 服务器编程模型示意图

这种方式中每个线程监听多个端口，负责 IO 的线程数目基本固定，不必频繁的创建和销毁；业务逻辑处理线程则通过线程池进行管理，活跃线程数根据当前计算负载进行变化，CPU 和 IO 都能保持较好的效率。

3.3.4 线程池模块

线程池技术能够显著的提高服务器的效率，在高并发服务器中，单个任务处理的时间比较短，待处理任务的数量比较大，若让用户自己管理线程则过于繁琐低效，而线程池技术能够代替用户对线程进行管理，并减少在创建和销毁线程上所花的时间以及系统资源的开销[27]。

本文研究了一种可高效运行的线程池模型，主要应用在处理服务器的计算任务中，由调度线程、线程池、任务队列和管理线程组成。由于服务器实时请求动态变化，所以需要线程池中的活跃线程数保持同步变化，其中管理线程能很好的根据任务队列大小动态调整线程池的大小，如图 3.5 所示，管理线程循环访问任务队列的状态。当任务队列为空的时候，管理线程销毁线程池中的空闲线程；当任务队列不为空的时候，如果线程池有空闲线程则等待调度线程分配任务，否则管理线程会新建线程进行处理。此外，如果线程池中的线程数量过多会导致切换代价过高，线程数量过少又会影响系统性能，所以线程池应保证一个最大值与最小值。

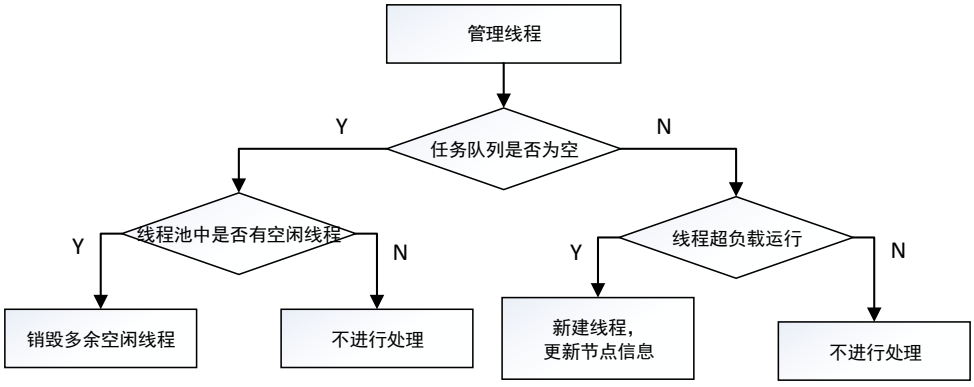

图 3.5 动态管理流程图

另外，如果调度线程随机选择线程池中的线程分配任务的话，由于任务内容不同，很可能出现某些线程空闲而某些线程阻塞的情况，严重影响系统性能。本文中

调度线程以守护线程模式运行，利用负载均衡的思想，将任务队列中的任务合理均衡的分配到线程池中合适的线程上去，可以提高系统的处理能力，减少用户等待响应的时间。对于任务队列中的每一个任务，调度线程都会按照公式（3.1）为其分配处理线程。

$$M = CV_n \tag{3.1}$$

式（3.1）中，M 表示线程当前的负载，C 表示线程当前处理的任务数，n 为线程已经处理的任务数，V_n 为线程在处理 n 个任务后的效率。算法通过当前任务数与效率的乘积表示线程的当前负载，调度线程首先判断是否存在 n=0 或者 C=0 的线程，若有则随机分配任务到其中一个，如果没有则将请求分配到当前负载 M 最低的线程上，然后对 V_n 进行更新。V_n 可以通过公式（3.2）进行计算：

$$V_n = \begin{cases} \dfrac{(n-1)V_{n-1} + 1/t_{n-1}}{n} & n > 0 \\ 0 & n = 0 \end{cases} \tag{3.2}$$

式（3.2）中，t_{n-1} 表示线程处理第 n-1 个任务所耗费的时间，算法通过对上个请求的处理时间进行算术平均来表示当前线程的工作效率。本方法计算简单、执行时间短、基本不消耗硬件资源，对系统性能影响可以忽略，具有一定的实用价值。调度线程的具体流程如图 3.6 所示。

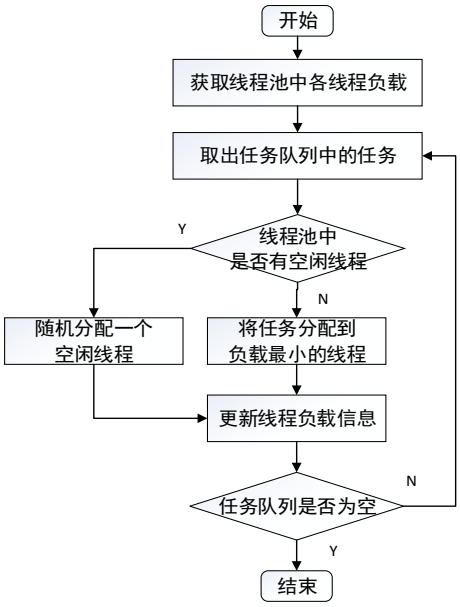

图 3.6 任务调度流程图

3.3.5 日志模块

日志模块是服务器系统中不可或缺的组成部分，日志模块会在 Web 服务器接收、处理、响应请求以及运行时记录通知、警告、错误各种信息。在试验后对日志进行统计、分析、综合，日志模块可以让开发者更好的查找排除错误原因、掌握服务器的运行状况、维护和管理服务器系统[28]。

参考 log4j/logback 通用日志库，本文设计的日志模块实现以下基础功能：

（1）设立了七种日志级别，包括"FATAL"，"ERROR"，"UERR"，"WARN"，"INFO"，"DEBUG"，"TRACE"，"ALL"；

（2）日志消息可以输出到多个位置，如屏幕、文件等；

（3）日志消息的格式可配置；

（4）日志模块不阻塞正常的执行流程；

（5）在记录大量日志的过程中不应该影响系统性能。

3.4 本章小结

本章首先对高性能轻量级 Web 服务器进行了需求分析，接着基于需求分析进行了服务器的总体设计，最后对服务器分模块进行具体设计。本章研究了一种新的服务器编程模型，将线程池与非阻塞 IO、IO 多路复用方式协同工作，能保证数据计算和网络 IO 都有良好的效率；另外设计了一种大小动态变化的线程池，并结合负载均衡算法的思想，可以将任务分配到合适的线程上，提高了服务器性能。下一步需要对所设计 Web 服务器进行具体的代码实现。

第 4 章 轻量级 Web 服务器的具体实现

通过对轻量级高性能服务器的需求分析与概要设计可知：TCP 连接模块完成了可靠数据传输的任务；HTTP 协议模块则是用户与 Web 服务器实现交互的基础；线程池模块用于多线程服务器中，有助于资源管理与提升性能；IO 模块则是保证服务器快速响应的关键；日志模块记录服务器的活动及异常，对于高性能服务器有着重要的作用。本章在此基础上对轻量级高性能 Web 服务器进行了进一步的详细设计，并在 Linux 操作系统上使用 C++实现了服务器的具体模块代码编写[29]，最终成功完成了一种轻量级的 Web 服务器的设计与实现。

4.1 TCP 连接模块实现

如图 4.1，在 TCP 模块中，每个 TCP 连接都抽象成一个 TcpConnection 结构体，包含着 TCP 连接所需要的所有信息，如 TCP 连接两端的 IP 地址 host 与端口 port，以及连接的超时时间 timeout、已连接时间 connectedTime、重连间隔 reconnctedTime 等等，另外可以看到一个 TcpConnection 对象包含着一个 Channel 通道对象，Channel 结构是对 Socket 读写操作的封装，在 4.3 节有详细介绍。TcpServer 结构体是 Web 服务器的基础，一个 Web 服务器通常只有一个 TcpServer 对象，TcpServer 对象负责服务器的初始化，分配资源以及管理 TCP 连接等工作。

在 TcpServer 初始化之后会启动一定 IO 线程监听指定端口，当获取到 Socket 文件描述符 fd 之后，服务器就会新建一个 TcpConnection 对象并关联到线程上完成连接，之后便由该线程负责这个 TcpConnection 的读写事件，由于本文设计的是一种异步非阻塞服务器，线程不应该阻塞在等待数据上，所以服务器会将这个 Socket 文件描述符注册到 Epoll 中，当没有数据到来时线程转而执行其他任务，待到有读写事件发生时，Epoll 会唤醒该线程进行处理。TcpConnection 对象拥有一个输入缓冲区 input 与一个输出缓冲区 output，Socket 读取到的数据保存在输入缓冲区 input 中，输出缓冲区 output 则负责保存服务器处理之后的数据，等待下一步的发送。Input 与 output 缓冲区在容量不足的时候会动态扩展[30]。

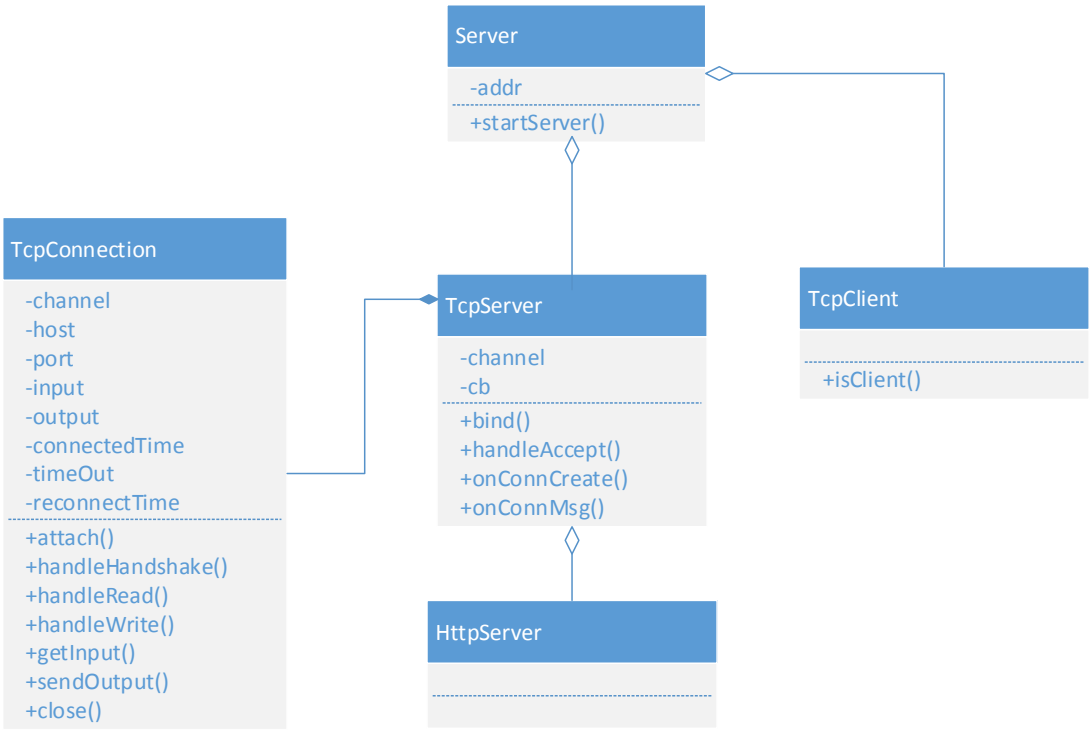

图 4.1 TCP 连接模块类图

以下对 TCP 模块中的主要函数及其作用作简单介绍：

（1）void handleAccept()；

在调用 Socket 套接字的 bind()与 listen()函数进行初始化之后，TcpServer 就开始监听客户端发来的请求，当收到用户发送的连接请求之后，TcpServer 会调用 attach()函数开始三次握手工作，attach()函数会依次调用 getPeerName()函数获取用户的 IP 地址与端口，调用 getSockName()函数获取本地 Socket 套接字的文件描述符，然后给 TCP 连接分配资源，将服务器状态改为 SYN_REV，最后新建一个 TcpConnection 对象进行进一步工作。

（2）int handleHandshake(const TcpConnPtr& con)；

在 TcpServer 对象调用 handleAccept()函数之后会新建一个 TcpConnection 对象，并由它来完成三次握手的剩余步骤。当 Socket 返回第二次握手的请求时，TcpConnection 对象用 handleHandshake()函数进行处理，函数首先会把连接状态改为 connected 并开始计时，接着返回用户对应的确认报文，此时服务器可认为 TCP 连接已经成功建立，之后便可以接收 TCP 数据报文开始工作。

（3）bool connectEstablished(const TcpConnPtr& con)；

connectEstablished()函数是一个回调函数，在 TCP 成功建立连接之后会被触发，它的主要任务是将该 Socket 的文件描述符通过 epoll_ctl()函数注册到 Epoll 中[30]，注册类型是读写，这样当有读写事件发生时 Epoll 就会异步唤醒指定线程进行处理。

（4）void handleRead(const TcpConnPtr& con)；

当客户通过 TCP 连接发送数据到服务器时，Epoll 异步唤醒 IO 线程执行 handleRead()回调函数，函数负责将数据读取到输入缓存区 input 中，进而触发 getInput()函数，它的作用是将缓存区 input 中的数据添加到任务队列并通知线程池模块进行处理。

（5）void handleWrite(const TcpConnPtr& con)；

同 handleRead()函数一样，当应用层对请求进行业务逻辑处理之后，对应的 Socket 套接字会产生写请求，Epoll 异步唤醒 IO 线程执行 handleWrite()函数，将处理后的数据写到输出缓存区 output 中，同样会触发 sendOutput()函数将缓存区中数据返回给用户，需要注意的是，返回结果的时候并不是一次性返回所有数据，也有可能分段返回[31]。

（6）void reconnect()；

当 TCP 连接失败或者因为某些原因断开后，服务器会调用 reconnect()函数重新连接，reconnect()函数的实质就是在一定的时间间隔之后，服务器主动向对应的用户发送 TCP 连接。为了避免网络资源浪费，服务器会析构未连接成功的 TcpConnection 对象并将分配的资源进行回收。

（7）void close(const TcpConnPtr& con)；

当连接失败或者任务处理完之后，服务器与客户端断开连接，此时需要调用 cleanUp()函数清理服务器预留的资源，回收输入输出缓存区，否则服务器很快就会提示内存不足。

4.2 HTTP 协议模块实现

HTTP 是基于 TCP 可靠传输的应用层协议，而且在本文设计的服务器中，HTTP

连接被看作是在 TCP 连接的基础上加入了对 HTTP 请求与响应的处理，所以 HTTP 协议模块中同样有 HttpServer 和 HttpConnection 两个结构体，HttpServer 表示一个支持 HTTP 协议的 Web 服务器，而 HttpConnection 继承自 TcpConnection，包含一次 HTTP 连接的所有内容。如图 4.2，此外还有 HttpRequest 和 HttpResponse 两个结构体，他们是对 HTTP 请求报文和响应报文的描述。

TCP 模块接收到报文段之后会解析成一个 HTTP 报文并传递到应用层， HTTP 模块一般通过 getResquest()函数读取 HTTP 报文，HttpRequest 结构体中的 tryDecode()函数可以提取 HTTP 请求中的信息，原理是遵循 HTTP 报文格式利用字符串操作函数对数据进行逐行解析，解析完之后根据请求方法 method 的值触发不同的回调函数 onGet()/onPost()，回调函数会调用 HttpRequest 中的一些 API 如 getArgs()获得请求参数，getBody()获得请求正文并提交到业务逻辑部分进行处理。

业务逻辑部分处理完成之后返回的结果并不是标准的 HTTP 报文，所以 HttpResponse 会调用 tryEncode()函数将返回数据构造成标准的 HTTP 响应报文，包括设置状态码，构建 HTML 页面等等工作[32]。最后通过 sendResponse()函数将数据传递给 TCP 模块传输，至此便完成了一个完整的 HTTP 连接。不同于 TCP 连接，HTTP 是非持久的，所以模块也没有分配资源保留本次连接的信息。

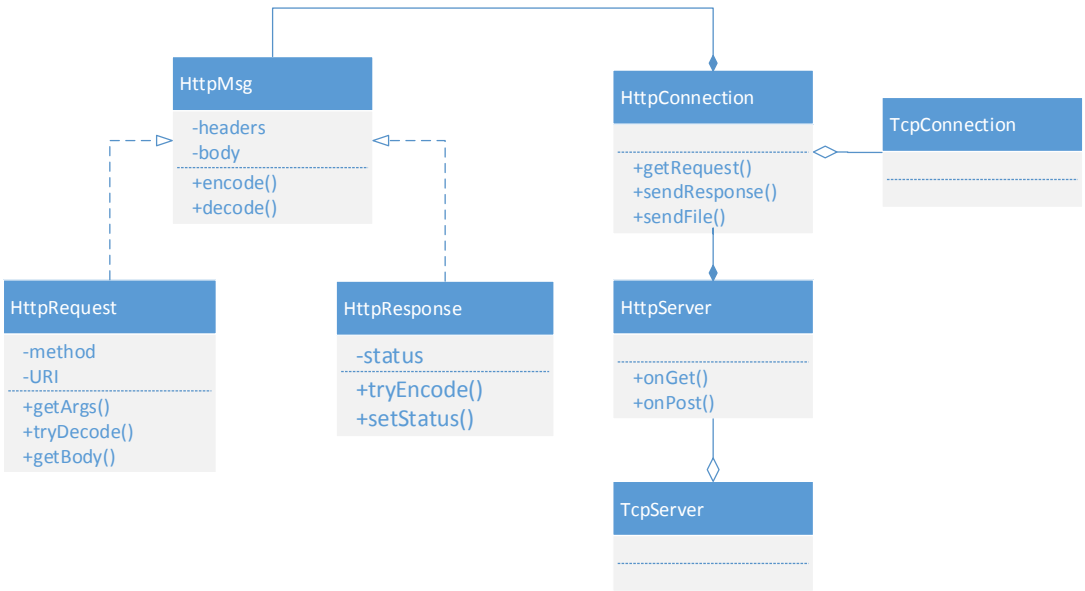

图 4.2 HTTP 协议模块类图

下面对 HTTP 协议模块中的主要函数及作用作简单介绍：

（1）Result tryDecode(Slice buf, bool copyBody, Slice* line1)；

当 HTTP 模块从 TCP 报文中提取出一个 HTTP 报文的时候，需要调用 decode()函数对报文进行解析以提取有用的信息，例如在 GET 请求方法到来的时候，服务器会调用 map_get()将报文中的信息整理成键值对数组 headers，然后服务器可以在数组中查找 URL，Accept 等头部名称取得对应的值进行下一步操作。

（2）int tryEncode(Buffer& buf)；

在服务器处理完请求的时候需要向客户端发送 HTTP 响应报文，这样浏览器才能正确识别服务器返回的内容，所以就需要调用 encode()函数与 C++中的字符串操作函数，将返回结果与 HTTP 属性联接在一起封装成一个标准的 HTTP 报文。

（3）void sendFile(const string& filename)；

在大部分的网页中，HTTP 协议不但可以用来访问网站，还可以用来传输文件，sendFile()函数就可以实现这一功能。服务器首先将通过 HTTP 协议接收到的二进制数据缓存，只要用户请求报文中传递的字符编码格式正确，数据就能正确的把二进制数据还原成文件进行传输[14]。

（4）void onGet/onPost(const HttpCallBack& cb)；

从第三章的需求分析可知，本服务器至少需要支持 GET 与 POST 请求方法，针对不同方法的 HTTP 请求，服务器需要设置不同的回调函数 onGet()/onPost()，这样当收到 HTTP 报文并通过 decode()解析出客户的请求方法之后，就会触发对应的函数进行业务处理。比如当请求方式是 GET 的时候，服务器会对照着找到对应 URI 的内容并返回给用户；当请求方式是 POST 的时候，服务器会将请求内容与数据一并递交给对应的 Web 后端程序，由网站开发者负责业务流程。

4.3 IO 模块实现

IO 模块是保证服务器高性能与快速响应的关键，如图 4.3，为了保证高性能的 IO 响应，本文设计了 Channel，Poller，EventBase 三个类来保证 Reactor 模式的实现。其中 Channel 对象是对 Socket 的包装，封装了可以进行 Epoll 的一个文件描述符 fd，

包括 listenfd、connectfd、eventfd 和 timerfd 几种[33]；Poller 对象是对 Epoll 的包装，封装了操作系统对 IO 多路复用的支持；而 EventBase 事件分发器是实现 Reactor 模式工作的关键，每个 IO 线程有且仅有一个 EventBase 对象，一个 EventBase 对象包含一个 Poller 对象，每个 Poller 对象上都注册了该 IO 线程监听的多个 Channel，EventBase 对象循环检查并驱动 Poller 对象发现事件，然后执行已经准备就绪的 Channel 中对应的回调函数。

服务器初始化后会开启一定的 IO 线程以 SO_REUSEPORT 方式监听指定端口，并给每个线程新建一个 EventBase 对象，SO_REUSEPORT 方式允许多个线程监听同一个端口且保证各线程均衡[34]。当 IO 线程收到连接请求之后会新建一个 TcpConnection，并将 TcpConnection 对象对应的 Channel 对象注册到 Poller 对象中，EventBase 对象循环执行 loop()函数，loop()函数实际是调用 Poller 对象的 poll()函数以获得就绪的 Channel 集合，再遍历该集合执行每个 Channel 的 handleEvent()回调函数。在这种工作模式下，监听请求的 IO 线程数目不会频繁的创建与销毁，减少了资源浪费；对于计算密集的任务请求来说，服务器会将请求中的逻辑业务内容分派到线程池中进行处理，线程池模块在 4.4 节有详细介绍。

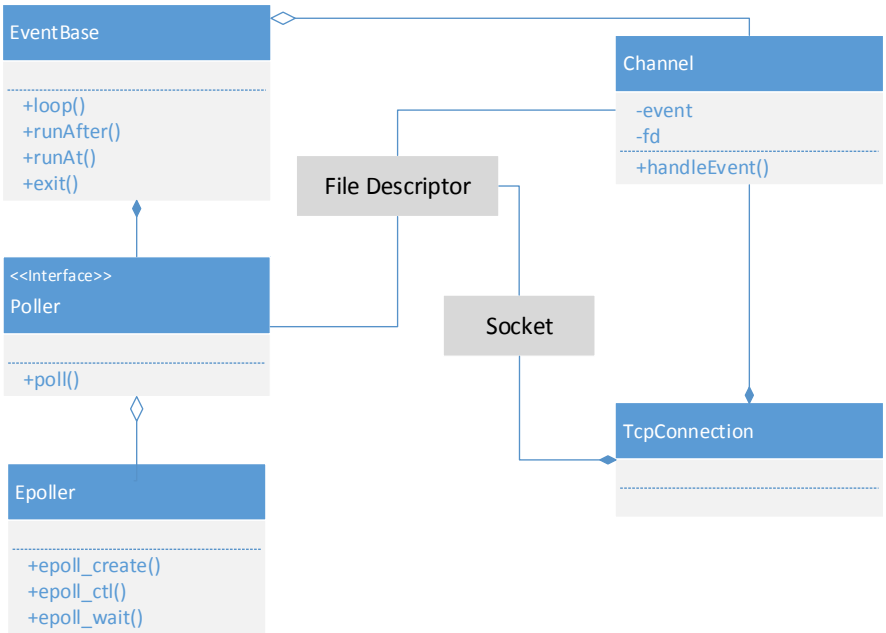

图 4.3 高性能 IO 模块类图

本文设计的高性能 IO 模块基于 Epoll 实现，主要调用了三个函数：

（1） int epfd = epoll_create(int size)；

调用 epoll_create()函数创建并初始化 Epoll 的句柄，size 是系统内核监听事件的数目大小。

（2） int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)；

epoll_ctl()函数可以用来控制 Epoll 监听的事件，如注册、修改、删除事件等等工作。当线程处理完某个请求后等待系统响应结果的时候就将它注册在 Epoll 中。

（3） int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout)；

调用 epoll_wait()函数可以在事件异步响应的时候通知服务器，参数 events 表示内核监听的事件集合，maxevents 表示监听事件的个数[35]，当超过 timeout 还没有事件触发时，返回超时。

Reactor 模式的实现核心是 EventBase 对象中的 loop()函数，它循环调用 Poller 对象的 poll()函数，直至服务器停止工作。poll()函数的核心代码如下：

```
lastActive= epoll_wait(fd, activeEvents, kMaxEvents, waitMs);
while (--lastActive>= 0) {
    int i = lastActive;
    Channel* ch = (Channel*)activeEvents[i].data.ptr;
    int events = activeEvents[i].events;
    if (ch) {
        if (events & (kReadEvent | POLLERR)) {
            ch->handleRead();
        } else if (events & kWriteEvent) {
            ch->handleWrite();    }}}
```

从代码中可以看到 poll()函数会调用 epoll_wait()函数获得已经就绪的 Channel 对象集合，再依次遍历该就绪集合执行对应 Channel 中的回调函数 handleRead()/handleWrite()，至此便完成了一个完整的 Reactor 模式。

4.4 线程池模块实现

相比于进程，线程的的上下文切换代价较低，IO 并发能力强，创建与销毁时间都大幅缩短，但是如果服务器频繁执行时间较短的任务的话，同样会引起大量线程的创建和销毁，这也会带来大量资源浪费。线程池技术就是在服务器中始终保持一定量的线程处于工作状态，减少在创建和销毁线程上的系统资源开销。另外多线程服务器对共享资源管理要求较高，不方便操作系统管理[36]，线程池技术也能够代替用户对线程进行管理。

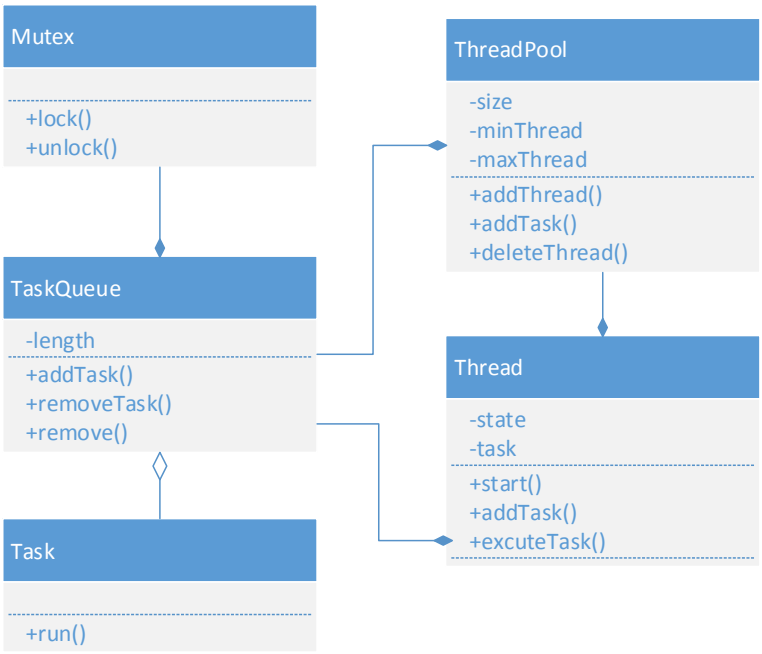

图 4.4 线程池模块类图

根据第三章的需求分析可知，本文需要实现一种高性能的线程池模型用于密集数据处理，模型由调度线程、计算线程池、任务队列和管理线程 4 部分组成，管理线程负责保证线程池中的线程数随任务队列负载动态变化。如图 4.4，线程池 ThreadPool 在服务器初始化的时候会设立一个最小线程数 minThread 与最大线程数 maxThread。线程池至少保证 minThread 数量的线程，当线程池中没有空闲线程而且有请求在等待的时候，线程池会新建空闲线程进行处理，线程增长的数量上限不能超过 maxthread；当任务队列为空且线程数大于 minThread 时，线程池会销毁多余线

程，但不能少于 minThread 个。根据经验公式可知[31]，线程池的最佳工作线程数取决于处理器数目 N 以及工作队列中任务的性质。由于本文中线程池主要负责处理计算性质的任务，在线程数为 N+1 的时候可以获得最佳效率，所以 minThread 设为处理器数目 N；而 linux 默认一个进程支持的线程数目为 1024，则最大线程数 maxThread 设为 1024N。

管理线程自服务器启动便轮询任务队列 TaskQueue 与线程池 ThreadPool 的状态，如果有空闲线程并且任务队列为空的话，则表示线程相对当前连接过多，通过 deleteThread()把超过 minThread 数的空闲线程销毁。如果没有空闲线程并且任务队列不为空的时候，管理线程会调用 addThread()增加任务队列大小 2 倍的线程并放入线程池中，然后启动线程去执行任务。管理线程执行的的核心代码如下：

```
while(1){
    if(TaskQueue.front == TaskQueue.rear)        //任务队列为空
        //遍历线程池销毁超过阈值的空闲线程
        deleteThread (ThreadPool, minThread);
    else{
        if(ThreadPool.size != 0)){              //是否有空闲线程
            DeQueue(TaskQueue, fd);      //删除队头元素
            chooseThread (ThreadPool, fd);   //选择一个空闲线程处理任务
        }
        else{
            //新建任务队列中任务数量 2 倍的线程
            addThread(ThreadPool, maxThread , TaskQueue.length*2);          }}}
```

线程池的使用有两种策略：一种是整个线程池共享一个任务队列，有新的任务到来即添加到任务队列尾部，每个空闲线程循环从任务队列头部取出任务执行[37]。另一种是每个线程带有一个任务队列，在线程的动态存储区存放计算负载的信息，负载的计算方式如公式（3.1），调度线程接收到任务之后轮询线程池并分配到负载最低的线程，线程把该任务放入自身的任务队列并排队等待执行。

以下简单介绍下线程池模块中的主要函数：

（1）void addThread()；

在线程池处理能力不足的时候，管理线程需要调用 addThread ()函数新建线程进行操作，本质是通过调用 pthread_create()库函数实现。此时新创建的线程负载为 0，之后再由调度线程进行选择。

（2）nodeId chooseThread(int node, task& t, CallBack& cb)；

本函数是调度线程的核心实现，当接收到新的任务时，调度线程会遍历线程池寻找负载最低的线程并将任务分派到其对应的任务队列，如果有负载为 0 的线程则停止遍历直接分配。

（3）void updateLoad(const nodeId& id)；

当接受到新的请求或者处理完任务之后，线程的负载值都会发生变化，需要注意的是 updateLoad()函数是处理线程主动调用，updateLoad()函数从对应线程的动态存储区取出保存的数据重新计算并更新权值。。

（4）void deleteThread(const nodeId& id)；

在线程池处理能力过剩的时候，为了避免多余线程继续占用资源，管理线程会调用 deleteThread()删除冗余的空闲线程，相较于 addThread()，线程在退出之前会将对应的记录删除。

由于线程共享进程中的资源，所以在多线程服务器中，线程的同步互斥问题不可避免。例如在使用线程池时，一个线程不断从向缓冲区中写入数据，而另一个线程又不断的从缓冲区中读取数据，这样难以保证共享资源的安全，本文主要利用互斥锁 mutex 保证线程安全，所有需要对共享资源进行操作的线程竞争同一个互斥锁，在同一时间只有一个线程能拥有互斥锁，这个线程允许访问临界资源，其他线程会被挂起直到互斥锁被释放[38]。

4.5 日志模块实现

日志模块的实现较为简单，本文可以通过 setLogLevel()函数可以记录不同事件的日志等级，包括"FATAL"，"ERROR"，"UERR"，"WARN"，"INFO"，"DEBUG"，

"TRACE"，"ALL"七种；setLogAppender()函数可以设置日志的输出位置，在服务器运行的同时，日志默认会被记录并保存在以时间命名的日志文件中，服务器规定了每个日志文件的大小与日志文件的数目，过期的日志文件会被覆盖或者存储在别处，避免占用太多的磁盘空间。setLogLayout()函数可以设置日志的输出格式，以便开发或运维人员查阅。

需要注意的是，在多线程服务器中，多个线程并发记录日志不能保证线程安全，所以一般情况下都是用一个简单的全局 mutex 保护日志 IO，但是在高性能 Web 服务器中，这种竞争全局锁的方法效率难以接受。所以本文使用一个独立的日志线程收集日志信息，需要记录日志的线程只需将日志传输到该线程即可，这种方法也称为异步日志[39]。

4.6 本章小结

本章主要内容是在前文的需求分析与设计的基础上，对高性能轻量级服务器的主要模块进行了具体实现与详细介绍，包括 TCP 连接模块、HTTP 协议模块、高性能 IO 模块、线程池模块、日志模块等等。至此完成了本文对于轻量级服务器的设计及实现，接下来的章节是对于所设计 Web 服务器的性能测试以及优化。

第 5 章 性能测试与结果分析

一般情况下 Web 服务器的性能表现与硬件配置、操作系统、网络速度等因素有关[40]，通过对所设计的服务器进行测试能有效的确定影响 Web 服务器性能的关键因素，有助于进一步采取有效的方法进行优化。本章通过对 Web 服务器的基本功能、响应时间、系统容量、资源利用率、吞吐能力等指标进行测试，可靠的获得了本文所设计的服务器的性能评价。

5.1 测试环境

由于本章需要对所设计的服务器进行系统容量测试，而对于百万级的并发网络访问，个人电脑几乎无法模拟测试，实际的物理机实验也很难实现；另外在对服务器进行压力测试的时候，同样希望服务器配置能动态设定以进行对比测试，所以本文选择采用云主机作为测试环境。

如图 5.1，首先创建两台 uCloud 云主机，一台作为服务器，一台作为客户端。CPU 设置为 16 核，内存 64G，系统选择 ubuntu14.04 64bit，由于需要测试服务器的并发连接能力，属于网络 IO 密集型任务，所以需要打开网络增强选项，申请弹性 IP 并绑定到主机，然后建立连接并逐渐增加，观察服务器负载情况。

图 5.1 服务器硬件配置

所以，对本文设计的 Web 服务器进行测试的硬件环境如表 5.1。

表 5.1 服务器测试环境

CPU	16 Core 3000MHz
内存	64G
硬盘	20G SSD
操作系统	Ubuntu 14.04

接下来在操作系统上安装好 C++编译器和一些必备的库[41]：

```
apt-get update
apt-get install -y screen git make g++
```

接着安装 Linux 上对 Web 服务器的一些监测工具：

```
apt-get install -y nload iptraf httping
```

从 Git 代码托管网址拷贝代码并编译，之后所有的测试环境便安装完备了。

```
git clone https://github.com/chenper/myServer
cd server
make
```

5.2 Web 功能测试

soapUI 是一个对 Web 服务进行测试的开源工具，它可以让开发人员通过图形化界面测试基于 SOAP 的 Web 服务，随着新版本的推出，soapUI 也增加了对 REST 的支持[42]。本文利用 soapUI 对服务器进行基础的 HTTP 请求响应测试。

首先在一台主机上启动 HTTP 服务器 testServer，绑定 8080 端口开始测试，设置 GET 方法回调函数 onGet()与响应内容，等待 HTTP 请求到来，核心代码如下，测试结果如 5.2 所示。

```
int threads = 1;                          //线程数为 1
setloglevel("TRACE");                     //设置日志级别
ThreadPool base(threads);                 //开启线程池
```

```
HttpServer testServer(&base);                          //新建一个 HTTP Server
int r = sample.bind("", 8080);                         //绑定 8080 端口
exitif(r, "bind failed %d %s", errno, strerror(errno));
sample.onGet("/get", [](HttpConnPtr& con) {            //设置 GET 方法回调函数
    string v = con.getRequest().version;
    HttpResponse resp;
    resp.body = Slice("GET method works");             //设置返回报文内容
    con.sendResponse(resp);                            //发送 Response 报文
    if (v != "HTTP/1.1") {                             //仅支持 HTTP/1.1 版本
        con->close();      }
    Signal::signal(SIGINT, [&]{base.exit();});
testServer.loop();                                     //循环等待 HTTP 请求
```

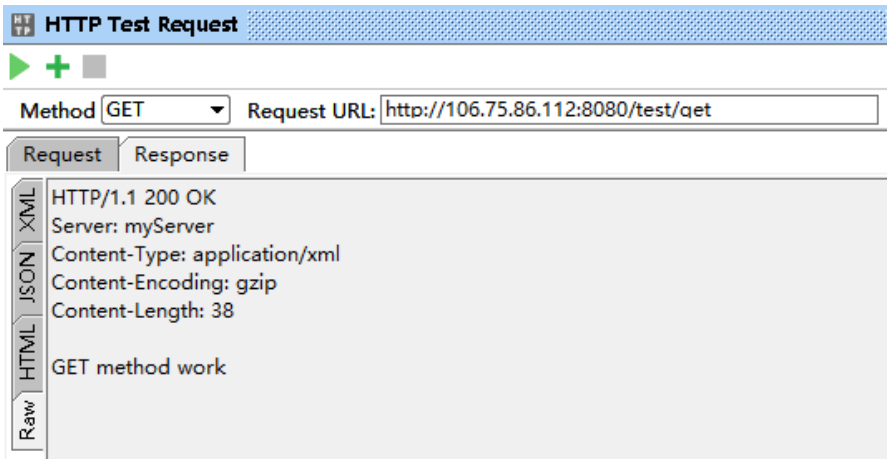

图 5.2 HTTP GET 方法测试结果

从图 5.2 中可以看到 HTTP 请求报文与响应报文的详细内容：请求方法为 GET，URL 为 http://106.75.86.112:8080/test/get，HTTP 版本是 1.1；响应报文中状态码为 200 表示正常工作，Server 主机类型为本文设计的服务器 myServer，返回报文格式为 XML，长度为 38，内容是回调函数设定的“GET method work”。可知服务器能正确响应 HTTP GET 请求报文。

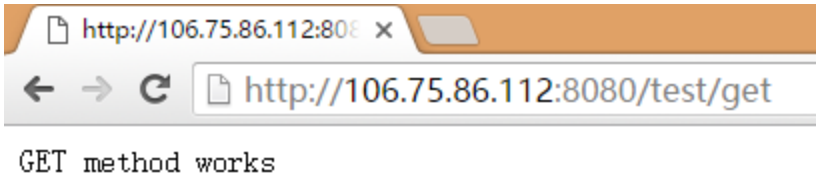

图 5.3 通过浏览器访问服务器

继续通过 Chrome 浏览器访问同一 URL，结果如图 5.3，可知在实际使用中用户也可以通过浏览器访问服务器获取服务，满足 Web 服务器的基本要求。接着进一步测试 POST 方法，首先将服务器的回调函数改成 onPost()，onPost()设置如果请求成功则显示“POST method works”，并且 onPost()函数会将接收到的正文字符串改成大写，在 POST 请求的正文中传输“this is data to post”继续测试，结果如图 5.4 所示。

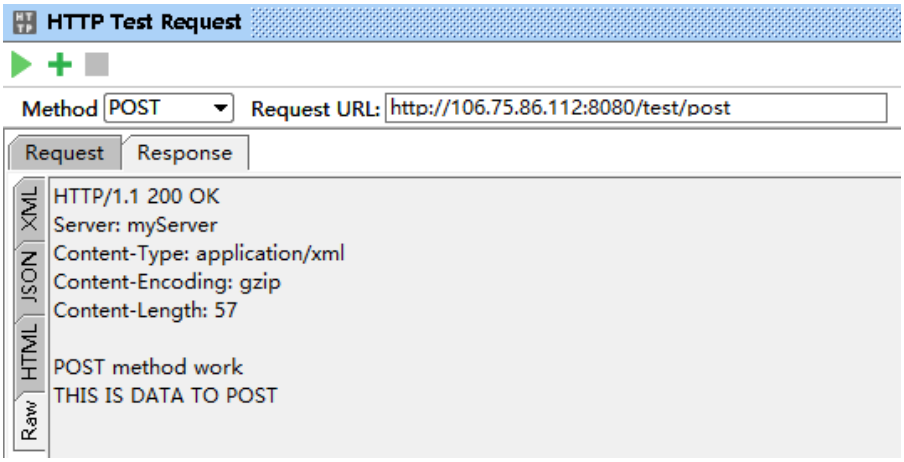

图 5.4 HTTP POST 方法测试结果

从图 5.4 中可以看到请求方法为 POST，协议版本为 HTTP/1.1，URL 为 http://106.75.86.112:8080/test/post，状态码为 200 表示正常响应请求，Server 主机类型为本文设计的服务器 myServer，返回报文格式为 XML，Content-Length 表示响应报文中返回的数据长度为 57，内容是回调函数设定的“Post method work”，而且把 POST 到服务器的数据都改成了大写“THIS IS DATA TO POST”。可知服务器能正确响应 HTTP POST 请求报文。

通过本节测试可以得到结论：本文的 Web 服务器实现了通过 HTTP 协议进行访

问的功能，并且支持 GET 和 POST 两种请求方法，用户可以通过浏览器对 Web 服务器进行访问，满足了 Web 服务器的基本设计要求。

5.3 服务器系统容量测试

2001 年著名的 C10K 问题被提出，C10K 是研究服务器同时支持上万个并发连接的问题，如果服务器支持服务的用户数越多，也就意味为单个用户提供 Web 服务的成本越低，服务器的性能越好。本节对设计的服务器进行了系统容量方面的测试，包括对海量连接的处理能力，以及高并发访问下的网络情况。

5.3.1 系统参数设置

Linux 操作系统默认对文件描述符的数量以及缓冲区大小作了限制，为了支持海量并发连接，首先需要对系统相关参数进行调整[43]。

```
sysctl -w fs.file-max=10485760          #系统允许的文件描述符数量 10m
sysctl -w net.ipv4.tcp_wmem=1024        #每个 tcp 连接的写入缓冲区 1k
sysctl -w net.ipv4.tcp_tw_recycle=1     #快速回收 time_wait 的连接
sysctl -w net.ipv4.tcp_tw_reuse=1
sysctl -w net.ipv4.tcp_timestamps=1
sysctl -w net.ipv4.tcp_rmem=1024        # tcp 连接缓冲区大小
#修改默认的本地端口范围
sysctl -w net.ipv4.ip_local_port_range='1024  65535'
#设置用户进程支持打开的最大文件数
echo '* soft nofile  1048576' >> /etc/security/limits.conf
echo '* hard nofile  1048576' >> /etc/security/limits.conf
#单个进程支持的最大文件数
ulimit  -n 1000000
```

5.3.2 并发连接仿真

接下来开始对服务器的并发连接性能进行测试，首先编写代码启动一个 Web 服务器 HttpServer，服务器以多进程多线程方式运行，进程数目设置与 CPU 核心数相同，进程中的线程在指定端口上监听请求，如果有连接成功建立则将连接状态显示在命令行上，核心代码如下：

```
for (int i = 0; i < processes; i ++) {
    pid = fork();                          //循环调用 fork()函数新建多个子进程
    if (pid == 0) {
        long connected = 0, closed = 0, recved = 0;          //设置连接状态
        for (int i = 0; i < end_port-begin_port; i ++) {     //监听多个端口
            TcpServerPtr p = TcpServer::startServer(&base, "", begin_port + i, true);
            p->onConnCreate([&]{ TcpConnPtr con(new TcpConn);
                con->onState([&](const TcpConnPtr& con) {
                    auto st = con->getState();
                    if (st == TcpConn::Connected) {
                        connected ++;                        //实时监视连接状态
                    } else if (st == TcpConn::Closed || st == TcpConn::Failed) {
                        closed ++; connected --;     }}};
```

选取一台主机作为服务器端，输入命令启动服务器监听指定端口，如图 5.5 所示，服务器正在监听 100-300 的端口号等待请求到来，此时已连接数显示为 0。

```
test/Server 100 300 10 301;
```

```
2016/12/28-09:25:30.369485 3a8 INFO fd 191 listening at 0.0.0.0:215
2016/12/28-09:25:30.369485 3a8 INFO fd 177 listening at 0.0.0.0:241
2016/12/28-09:25:30.369485 3a8 INFO fd 187 listening at 0.0.0.0:193
2016/12/28-09:25:30.369485 3a8 INFO fd 211 listening at 0.0.0.0:288
2016/12/28-09:25:30.369485 3a8 INFO fd 225 listening at 0.0.0.0:201
2016/12/28-09:25:30.369485 3a8 INFO fd 206 listening at 0.0.0.0:244
2016/12/28-09:25:30.369485 3a8 INFO fd 234 listening at 0.0.0.0:267
2016/12/28-09:25:30.369485 3a8 INFO fd 201 listening at 0.0.0.0:288
2016/12/28-09:25:30.369485 3a8 INFO fd 186 listening at 0.0.0.0:234
2016/12/28-09:25:30.369485 3a8 INFO fd 197 listening at 0.0.0.0:251
```

图 5.5 服务器监听端口等待连接

接着编写代码模拟用户的并发访问，同样的将客户端设置为多进程多线程工作，本章测试的是服务器并发容纳用户的能力，所以每个线程轮流向服务器监听端口发送连接请求而不传输其他数据，客户端核心代码如下：

```
for (int i = 0; i < processes; i ++) {
    pid = fork();                          //循环调用 fork()函数新建多个子进程
    if (pid == 0) {
        for (int k = 0; k < create_seconds * 10; k ++) {
            //每秒创建 5000 个连接，提高连接创建的成功率。
            base.runAfter(100*k, [&]{
            int c = conn_count / create_seconds / 10;
            //循环向服务器所有监听端口发送请求
            for (int i = 0; i < c; i++) {
                short port = begin_port + (i % (end_port - begin_port));
                auto con = TcpConn::createConnection(&base, host, port, 20*1000);
                con->setReconnectInterval(5*1000);
                con->onMsg() {}     };   }}
```

选取另一台主机作为客户端，输入命令运行客户端，启动 50 个线程，连接到服务器的内网地址 10.9.195.7 的指定监听端口，每个线程发起 200K 个连接，如图 5.6，客户端已经初始化，开始向服务器发送连接。

```
test/Client 10.9.195.7 100 300 200000 500 50 600 64 301
```

```
2016/12/28-09:25:32.659870 770 INFO poller epoll 3 created
2016/12/28-09:25:32.659770 771 INFO poller epoll 3 created
2016/12/28-09:25:32.659995 770 INFO creating 200000 connections
2016/12/28-09:25:32.659870 771 INFO creating 200000 connections
2016/12/28-09:25:32.659748 773 INFO poller epoll 3 created
2016/12/28-09:25:32.659521 774 INFO poller epoll 3 created
2016/12/28-09:25:32.659448 773 INFO creating 200000 connections
2016/12/28-09:25:32.659080 772 INFO poller epoll 3 created
2016/12/28-09:25:32.659080 772 INFO creating 200000 connections
2016/12/28-09:25:32.659938 774 INFO creating 200000 connections
```

图 5.6 客户端准备发送连接

通过 watch 定时命令可以实时观察服务器并发连接数，输入命令 watch ss –s，命令行界面每隔 2s 刷新更新网络数据，可以看到连接数逐渐增加。

```
Every 2.0s: ss -s                                    Wen Sep 28  14:21:07 2016

Total: 70194 (kernel 0)
TCP:   70283 (estab 70283, closed 0, orphaned 0, synrecv 0, timewait 0/0), ports 0

Transport Total     IP        IPv6
*         0         -         -
RAW       0         0         0
UDP       5         4         1
TCP       70283     70281     2
INET      70234     70230     4
FRAG      0         0         0
```

图 5.7 服务器并发连接数

为了进一步分析服务器的性能，我们希望能记录并发连接数随时间的变化情况，所以编写了一个脚本每隔 2s 将服务器的成功连接数、已使用内存与 CPU 利用率写入文件，进行统计。

```
for((i=1;i<=30;i++));do
    echo netstat –nat | grep ESTABLISHED  | wc –l > connStatus.vi
    echo free -m | sed -n '2p' > connStatus.vi
    echo top –b > connStatus.vi
    sleep 2
    done
```

然后写入 crontab 里每分钟执行一次；

```
* * * * * /bin/bash  /home/chenpei/countConn.sh
```

5.3.3 网络监控

在逐步建立连接的过程中，可以借助 Linux 下的监控工具 Iptraf 查看主机在并发访问环境下的网络状况。iptraf 是一个基于 ncurses 开发的 IP 局域网监控工具，它可以实时地监视网卡流量、生成各种网络统计数据，包括 TCP、UDP、ICMP 和 OSPF 统计信息、节点统计、IP 校验和错误和其它一些信息[44]。

输入 iptraf 命令后，进入操作界面，选择“IP traffic monitor”观察 TCP 连接，如

图 5.8 可以看到每条 TCP 连接的具体信息，包括两端主机的 IP 与端口号，发送的报文数目、大小以及经过的网卡，还可以对端口进行排序统计，可以看到服务器连接端口保持在 100 到 300 之间。

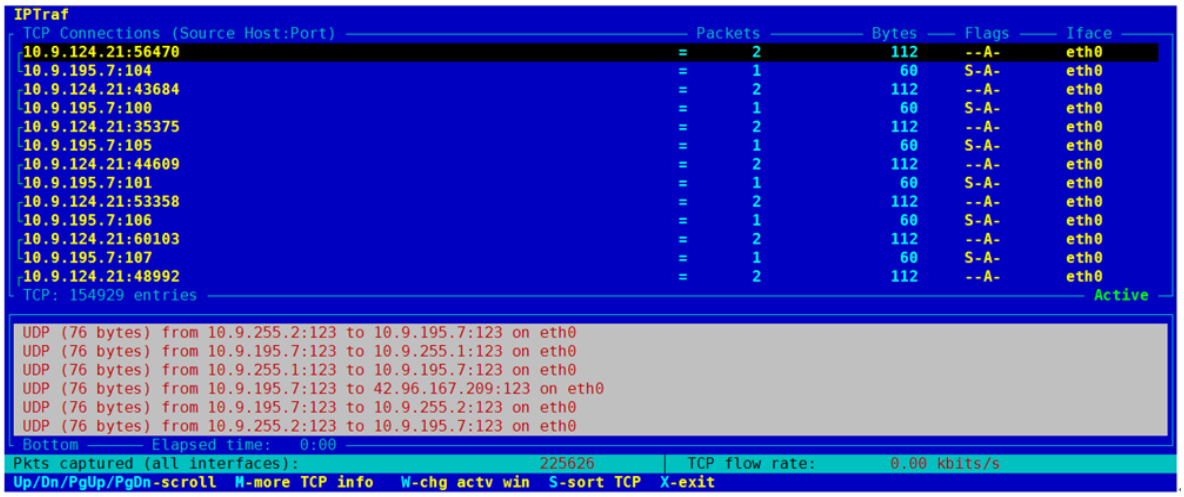

图 5.8 TCP 连接监控

接着选择“Detailed interface statistics”，可以看到各种报文的数量以及实时流量速率。从图 5.9 可以看到，在实验中，服务器建立的连接几乎都是 TCP 连接，而且接收到的报文数要大于返回的报文数；网络处于高负载状态下，流量速度最多可以到 500MBit/s，由于云主机的网络增强选项打开，所以 500Mbit/s 左右的流量速率远未达到峰值，排除了网络速度作为性能瓶颈的可能。

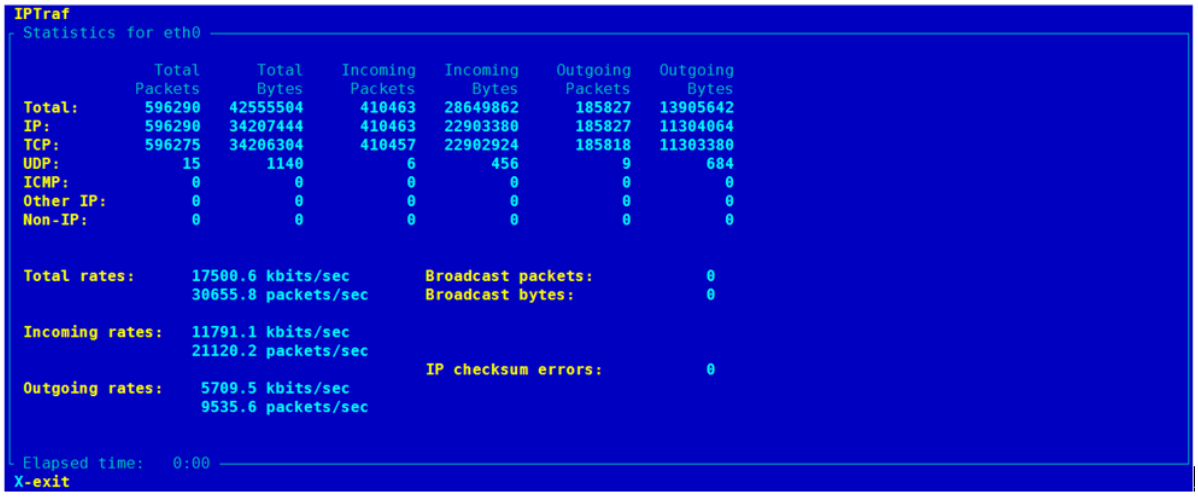

图 5.9 网络流量监控

5.3.4 实验结果分析

与此同时，在命令行可以看到服务器的并发连接数在不断升高，到了单个线程600K 并发连接的时候发生阻塞，并发连接数下降，开始出现 closed 状态的连接，此时服务器 CPU 负载在 85%左右，而内存负载为 100%，此时服务器 CPU 资源与内存都是导致性能下降的瓶颈。

表 5.2 服务器性能参数变化表

时间/ms	并发连接数	CPU 使用率/%	内存使用率/%
10000	151413	1.0	3.2
40000	399875	4.3	7.6
70000	889764	6.8	16.3
100000	2182642	13.2	34.5
130000	3201275	23.5	52.3
160000	4411158	34.9	67.5
190000	5121345	53.3	83.4
220000	5552101	61.7	89.3
250000	5799885	70.2	96.8
280000	6043687	84.3	100
310000	5832536	74.8	98.8
340000	5121134	53.5	83.3
370000	4312347	33.7	66.4
400000	3654843	46.6	53.0

表 5.2 列出了部分 connStatus 文件记录的数据，根据连接数随时间的变化做图 5.10。由 5.3.2 节的设计可知：随着时间增加，客户端发送的请求数目也是线性增加的，从图 5.10 中可以看到在运行初期，服务器的连接数基本是随着时间线性增加，watch 命令也显示没有连接失败的情况发生；随着连接数量不断增加，增长速度逐

渐放缓，达到 6M 左右出现 closed 状态的连接且连接数开始下降。此时内存占用率 100%，阻碍连接进一步增加，而且 CPU 利用率也接近 85%，由于内核要占用一定的 CPU 资源处理 IO，所以 85%的利用率表示 CPU 已经近似于饱和，之后随着客户端发来的请求数继续增长，过多请求竞争服务器资源导致服务器开始发生阻塞，并发连接数量逐渐下降。

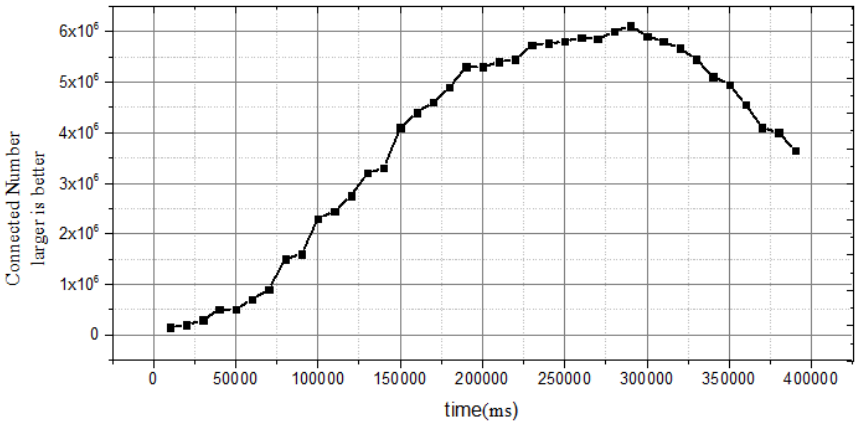

图 5.10 服务器连接数随时间变化

图 5.11 展示了 CPU 利用率与内存占用率随连接数的变化情况，可以看到内存占用率几乎是与连接数呈线性关系，因为服务器只会给成功建立的 TCP 连接分配内存，如果连接失败则服务器会回收分配的内存资源；与之相比，CPU 利用率在最开始也是线性增加的，在连接数接近峰值时 CPU 利用率上涨速度加快，迅速达到 85%，因为当连接数超过服务器负载能力时会导致竞争，一些线程开始阻塞。

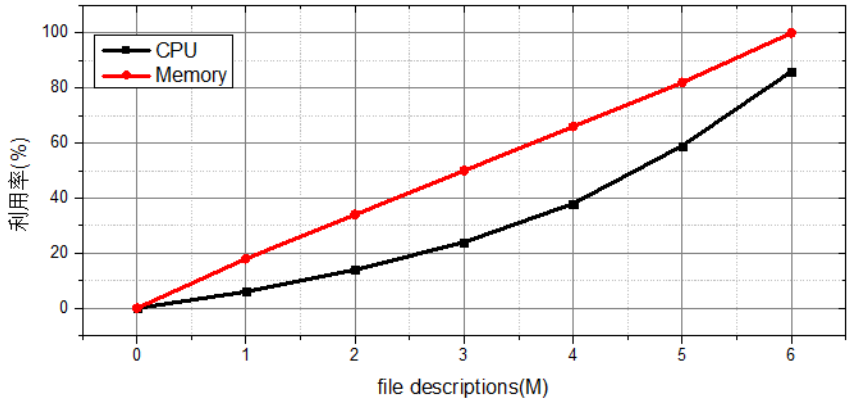

图 5.11 服务器负载随连接数变化

综上可知服务器在 16 核 CPU，64G 内存的条件下并发连接的峰值大概在 6M 左右，平均每个连接占用 10k 内存，所以可以得出结论：本服务器能够容纳百万级以上的并发访问，且保持良好的性能表现，达到了实验目的。

5.4 服务器吞吐率测试

5.4.1 吞吐率对比测试

吞吐率 QPS 指的是服务器每秒钟处理的请求数或数据量，可以有效的表示服务器的性能。为了验证本文所设计服务器的吞吐率，本节选取了时下流行的轻量级 Web 服务器 Nginx 与 Libevent 进行对比测试。Nginx 通常以多进程单线程的方式工作，前文已经详细介绍过了；而 Libevent 是一个用 C 语言编写的轻量级高性能服务器，同样基于事件驱动，支持多线程处理事件，支持多种 IO 多路复用技术，如 epoll、select 和 kqueue 等[45]。与 Nginx 和 Libevent 的吞吐率对比测试能很好的验证本文服务器的性能表现。

本节选用 Webbench 模拟并发连接测试，Webbench 是 Linux 上著名的网站压力测试工具，可以展示服务器每秒响应的请求数 QPS 和每秒钟传输的数据量以及响应失败的请求数[46]。首先启动三台云主机并分配足够的内存，接着分别安装对应的的服务器与 Webbench 工具，输入命令开始测试。

Webbench -c 1000 –t 30 http://hostIP/index.html

如图 5.12 展示了 Webbench 的用法，命令中的 1000 表示模拟的并发连接数，30 表示测试的持续时间，后面跟着要访问的网页 URL，等待 30s 之后就会返回测试每秒钟相应请求数和每秒钟传输数据量。

```
[root@localhost ~]# webbench -c 1000 -t 30 http://192.168.1.113/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Benchmarking: GET http://192.168.1.113/
1000 clients, running 30 sec.

Speed=238592 pages/min, 3360142 bytes/sec.
Requests: 119296 susceed, 0 failed.
```

图 5.12 Webbench 性能测试

为了测试线程数目对吞吐率的影响，服务器分别启动 1 个线程和 4 个线程进行对比测试，由于 Nginx 一个进程只有一个线程工作，所以相当于在 Nginx 中启动了 4 个 Worker 进程工作。逐渐增加连接数，同时观察服务器的负载情况与性能，记录对应的请求响应速率与连接失败个数，部分数据如表 5.3。

表 5.3 服务器 QPS 对比测试

文件描述符个数	Nginx		Libevent		自定义	
	单线程	多线程	单线程	多线程	单线程	多线程
1	19877	33086	20193	30964	20076	31871
10	28962	45836	33146	52675	30997	49715
100	47453	83524	60021	93245	59126	91743
1000	40558	79963	49982	80432	49527	81264
10000	38977	75416	43442	77548	45031	80345
100000		74545		77490		78341

5.4.2 实验结果分析

观察测试数据并绘出趋势图，图 5.13 展示了以单线程工作时，三种服务器的吞吐率与连接数量的关系。实验开始后三种服务器的 QPS 都随着连接数的增加快速上升，连接数为 100 的时候 QPS 达到峰值，本文设计服务器与 Libevent 的 QPS 峰值都在 60000 附近，而 Nginx 的 QPS 峰值只有本文服务器的 78%，原因是 Nginx 的响应内容较多，包括了多个 header，而且使用 chunk 编码[34]。此时三台服务器的 CPU 占用率都在 80%左右，接近满负载状态，当并发连接数进一步上升时，由于 CPU 资源有限，过多的请求相互竞争导致连接成功率降低，QPS 开始下降。

图 5.14 展示了 4 线程工作时的测试结果，可以看到曲线大致的形状不变，并发连接数为 200 的时候 QPS 达到峰值。另外随着线程数量增加，服务器支持的 fd 数量与吞吐率都有相应的增加，Nginx 服务器吞吐率增长明显，达到 97%。整个实验中本文服务器表现始终与 Libevent 的性能相近，略优于 Nginx。

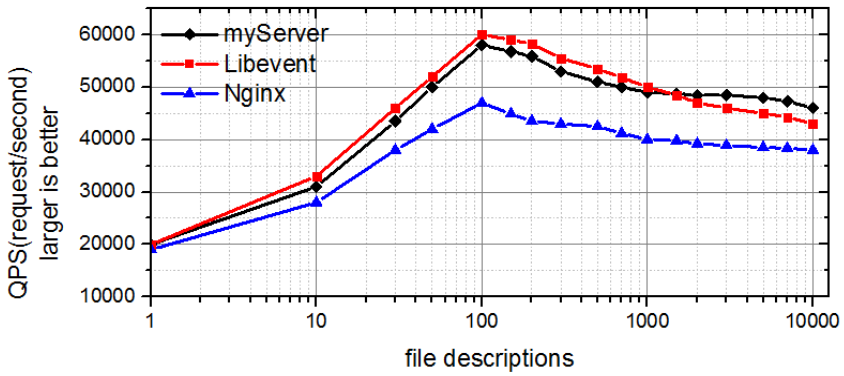

图 5.13 服务器单线程吞吐率

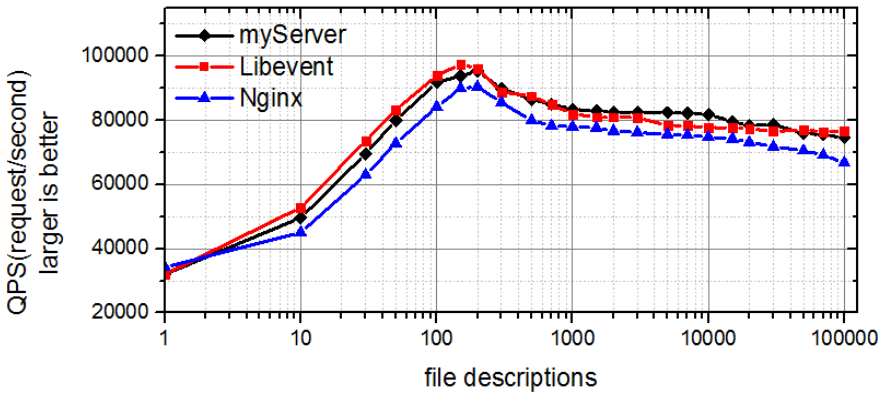

图 5.14 服务器 4 线程吞吐率

需要注意的是，在并发连接数经过峰值继续增加的时候，各个服务器的 QPS 虽然都有所降低，但是降幅较为缓慢，表示三个服务器在高并发情况下都能保证基本 QPS 的访问能力，原因是 Epoll 在高并发的情况下仍然能够保持很好的性能[47]。

实验结果显示：在单线程工作情况下，本文设计服务器吞吐率要优于 Nginx；在 4 线程工作情况下，本文服务器吞吐率有接近 50%的提升，具体表现与 libevent 相当。综上可以得出结论：本文设计的服务器事务处理性能表现良好，在以多线程方式工作时能有效提升吞吐率，达到设计要求。

5.5 访问延迟测试

延时是用户对服务器性能的直观感受，是从客户发送一个 HTTP 请求到接受到 HTTP 响应报文头部的时间间隔，在网络速度相同的情况下，延时由服务器的响应

时间决定。httping 是 Linux 中用于测试 HTTP 服务器访问延时的一款开源工具[48]，本节通过使用 httping 工具对比测量 Libevent 与本文服务器的访问延时，如图 5.15，输入命令行开始测试，其中数字 10 表示尝试连接十次记录延时并取平均值，hostIP 表示服务器的 IP 地址，默认访问服务器主页。

httping -c10 -h serverIP

```
[root@106.75.86.124]# httping -c10 -h 106.75.86.112
PING 106.75.86.112:80 (/):
connected to 106.75.86.112:80 (331 bytes), seq=0 time=12.96 ms
connected to 106.75.86.112:80 (331 bytes), seq=1 time=19.18 ms
connected to 106.75.86.112:80 (331 bytes), seq=2 time=11.60 ms
connected to 106.75.86.112:80 (331 bytes), seq=3 time=8.78 ms
connected to 106.75.86.112:80 (331 bytes), seq=4 time=12.91 ms
connected to 106.75.86.112:80 (331 bytes), seq=5 time=11.05 ms
connected to 106.75.86.112:80 (331 bytes), seq=6 time=16.38 ms
connected to 106.75.86.112:80 (331 bytes), seq=7 time=21.34 ms
connected to 106.75.86.112:80 (331 bytes), seq=8 time=16.97 ms
connected to 106.75.86.112:80 (331 bytes), seq=9 time=19.08 ms
--- http://106.75.86.112/ ping statistics ---
10 connects, 10 ok, 0.00% failed, time 1162ms
round-trip min/avg/max = 8.8/15.0/21.3 ms
```

图 5.15 httping 测试服务器延时

易知 Web 服务器的访问延迟会随着负载的变化而变化，所以需要测试不同负载下两种服务器的访问延时，在 5.3.2 节的 shell 脚本里添加命令：

httping -c5 -h serverIP | tail -n1 | awk '{print $4}' | cut -d/ -f2 > connStatus.vi

每隔两秒向服务器发送 5 个连接并截取延时平均值并保存到文件中，可以记录不同负载下服务器的延时，根据数据作出趋势图 5.16。

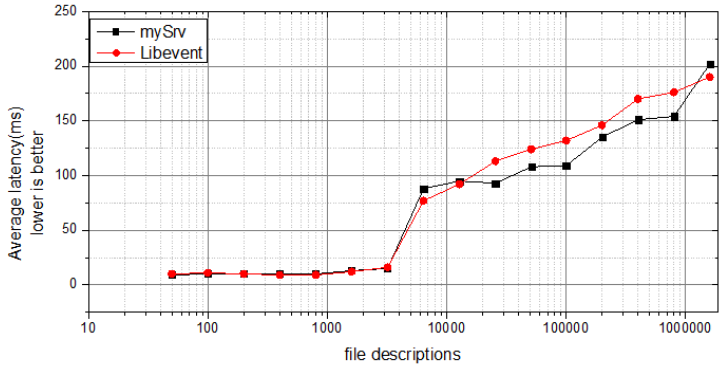

图 5.16 服务器延迟对比图

图 5.16 展示了服务器延时随连接数增长的变化，从图中可以看到当 fd 数目小于

3000 的时候，两台服务器的延时基本保持稳定，时间在 20ms 左右，fd 等于 3000 的时候曲线发生突变，延时提升至 100ms。并且随着 fd 的继续增加，延迟保持缓慢增长，在 1M 连接的时候延时大概在 250ms 左右，依然处于可接受范围之内，不影响使用。从图 5.16 可以看到本文服务器的访问延时总体表现与 libevent 相当，满足日常使用要求。需要注意的是，本节仅仅是测试访问主页的延时，在实际使用中一个 Web 系统的整体延时还与页面内容、网络速度与数据库 IO 速度等因素有关。

5.6 本章小结

本章对设计的 Web 服务器的系统容量、吞吐率等性能指标进行了测试，并对实验数据进行了记录整理与分析，对其中的某些问题进行了阐述与改进，提出了有效的解决方案。Web 服务器整体性能良好，在 16 核 64G 内存的物理环境下能够满足 6M 左右的并发连接请求，通过 Webbench 压力对比测试发现本文设计的服务器吞吐率表现要优于 Nginx，与 Libevent 相当。最后对服务器的访问延迟进行测试，发现服务器主页的访问延时在 20ms 左右，并且在连接数较多的情况下依旧能够保持较低的延时。最后得出结论：本文设计的 Web 服务器性能表现良好，基本达到了设计要求。

第 6 章 总结与展望

本文对广泛应用在互联网中的 Web 服务器进行了研究，通过对比各种 Web 服务器方案，指出了市场上流行的 Web 服务器的优劣。然后对服务器的基本原理以及网络应用设计基础进行了阐释，具体介绍了市面上流行的轻量级服务器 Nginx 的原理以及架构。接着设计了一种基于线程池的新型轻量级 Web 服务器，包括一种动态大小的线程池模型，以及基于线程池的任务调度算法。最后成功实现了本文设计的的高性能多线程 Web 服务器，并进行了实验测试分析，测试数据显示满足设计指标。

具体包括：

（1）通过调研市面上多种 Web 服务器方案，提出了一种基于线程池技术和 IO 多路复用的轻量级异步非阻塞服务器，服务器结合 Epoll 调用与线程池技术实现了 Reactor 工作模式，能够在资源较为有限的情况下保持良好的性能。另外，本文还分析了 Web 服务器的基本原理以及网络程序设计基础，并给出了在这一领域内的进展和成果。

（2）设计了一种大小随系统负载动态变化的线程池：当请求数目增加时，线程数量随之增加以满足性能的需求；当请求数目减少时时，线程池销毁空闲线程以节约系统资源。另外还研究了一种基于线程池的任务调度方法，线程池接收任务之后选择合适的线程进行处理，实现简单，具有一定的实用价值。

（3）遵循软件工程的标准对服务器进行了需求分析、模块设计以及具体的代码实现，满足低耦合高内聚的模块化要求。在 Linux 平台上对本文设计的服务器中进行了具体测试，实验结果显示服务器性能表现良好，支持浏览器访问与 GET、POST 请求方法，在 16 核 CPU，64G 内存的硬件条件下能容纳 6M 的并发连接，吞吐率表现略优于 Nginx，与 Libevent 相当，访问延迟基本稳定，性能表现达到了设计要求。

综上，本文设计的轻量级多线程服务器适用于以下场景：有多个 CPU 可用，未来硬件配置会进一步升级；提供非均质的服务，即服务请求有优先级区别；工作集较大，会出现瞬时流量激增的访问情况。

由于作者水平，开发时间等因素限制，本文实现的轻量级高并发 Web 服务器还存在很多不足之处需要进一步完善和改进。未来将从以下几个方面来展开工作：

(1) HTTPS 的支持

随着互联网的广泛化普及，人们越来越重视网络安全。一般情况下 HTTP 报文通过明文传输，而 HTTPS 通过 SSL 加密传输数据。因此 Web 服务器可以提供对 openSS 的支持来保证数据安全可靠。

(2) 更多功能完善

相比于 Nginx，本文只实现了服务器的基本功能，舍弃了负载均衡、反向代理等功能，换来了更快的响应速度与更低的资源占用，所以还不能作为一个完备的 Web 服务器实际使用，未来可以加入对网页缓存、session、WebSocket 等功能的支持，进一步对服务器功能进行扩展[49]。

(3) 进一步测试优化

文中使用的网络监测程序虽然十分方便有效，但是仍然不能完全反映 Web 服务器的运行情况，例如对服务器延时的测试应该对比不同报文长度进行测试，所以需要更加全面的监测 Web 服务器，并利用监测结果优化服务器设计；另外本文采用 Linux 平台的一些开源工具进行吞吐率、延时等指标的测试，与真实复杂的网站访问情况仍存在一定的差异，需要将服务器部署在实际生产环境中，通过实际工作结果反馈对服务器进行验证完善。

参考文献

[1] 中国互联网络信息中心.中国互联网络发展状况统计报告[R].2016.

[2] Google Inc. Google Architecture[EB/OL]. http://highscalability.com/google-architecture,2016.

[3] Apache Software Foundation. Apache[EB/OL]. http://www.apache.org,2016.

[4] Neudesic, LLC. IIS[EB/OL]. http://www.iis.net,2016.

[5] IBM. WebSphere[EB/OL]. http://www.ibm.com/developerworks/cn/Websphere/,2016.

[6] Igor Sysoev. NGINX[EB/OL]. http://nginx.org/,2016.

[7] Jan Kneschke. Lighttpd[EB/OL]. http://www.lighttpd.net,2016.

[8] Apache Software Foundation. Apache Tomcat [EB/OL].http://tomcat.apache.org,2016.

[9] Q. Fan and Q. Wang. Performance Comparison of Web Servers with Different Architectures: A Case Study Using High Concurrency Workload[J]. IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb), 2015:37-42.

[10] Host server analysis[EB/OL]. http://www.netcraft.com/ /hosting-analysis,2016.

[11] The World Wide Web Consortium[EB/OL]. http://www w3.org, 2016.

[12] Z. Huang, C. Xia, B. Sun and H. Xue. Analyzing and summarizing the Web server detection technology based on HTTP[C]. 6th IEEE International Conference on Software Engineering and Service Science (ICSESS), 2015:1042-1045.

[13] 牛磊. 基于 REST 的服务器框架研究与实现[D].北京邮电大学,2010.

[14] Wright G R, Stevens R W. TCP/IP 详解卷 1:协议 [M]. 范建华等译.北京:机械工业出版社, 2004: 171-201.

[15] T. Mantoro, M. A. Ayu, A. Borovac and A. Z. Z. Zay. IPv4 and IPv6 client-server designs: The Sockets performance[C]. International Conference on Multimedia Computing and Systems, 2012:629-634.

[16] Wikimedia Foundation, Inc. Epoll[EB/OL]. http://en.wikipedia.org/wild/Epoll,2016.

[17] 王成浩. 基于 EPOLL 的网络游戏服务器通信架构的研究与设计[D].大连海事大学,2012.

[18] 陶辉. 深入理解 Nginx:模块开发与架构解析[M].北京:机械工业出版社,2013.

[19] J. Zhao and L. Qin, Design and implementation of static server based on event-driven[C], 2014 IEEE 5th International Conference on Software Engineering and Service Science, Beijing, 2014, pp. 679-682.

[20] 祝瑞,车敏. 基于 HTTP 协议的服务器程序分析[J]. 现代电子技术,2012,04:117-119+122.

[21] X. Wu, X. Long and L. Wang, Optimizing Event Polling for Network-Intensive Applications: A Case Study on Redis[C], 2013 International Conference on Parallel and Distributed Systems, Seoul, 2013, pp. 687-692.

[22] 郗晓娇. 基于线程池和 SSI 框架的内容审核系统的设计与实现[D].南京大学,2013.

[23] 聂兰兰. 考虑外界影响因子的动态线程池优化设计与实现[D].华中科技大学,2014.

[24] X. Wang, Technical Analysis of High-Capacity and Concurrency Server Groups[C], 2010 International Conference on E-Product E-Service and E-Entertainment, Henan, 2010, pp. 1-4.

[25] H. Wu, C. Tan and H. Wang, Building a High-performance Communication Framework for Network Isolation System[C], 2008 IEEE International Conference on Networking, Sensing and

Control, Sanya, 2008, pp. 1086-1091.
[26] 余光远. 基于 Epoll 的消息推送系统的设计与实现[D].华中科技大学,2011.
[26] C. Hankendi and A. K. Coskun, Energy-efficient server consolidation for multi-threaded applications in the cloud[C], 2013 International Green Computing Conference Proceedings, Arlington, VA, 2013, pp. 1-8.
[27] 杨小娇. 轻量级高并发 Web 服务器的研究与实现[D].南京邮电大学,2014.
[28] Shuai Zhang, Tao Li, Qiankun Dong, Xuechen Liu and Yulu Yang, CPU-assisted GPU thread pool model for dynamic task parallelism[C], 2015 IEEE International Conference on Networking, Architecture and Storage (NAS), Boston, MA, 2015, pp. 135-140.
[29] 侯捷.STL 源码分析[M].北京:华中科技大学出版社.2006:113-129.
[30] 黄欣. 自适应网络 I/O 模型的研究与实现[D].华南理工大学,2012.
[31] 陈硕. Linux 多线程服务端编程[M].电子工业出版社,2013.
[32] M. D. Syer, B. Adams and A. E. Hassan, Identifying performance deviations in thread pools[C], 2011 27th IEEE International Conference on Software Maintenance (ICSM), Williamsburg, VI, 2011, pp. 83-92.
[33] W Richard Stevens and Stephen A. Advanced Programming in the UNIX [M].人民邮电出版社.2006:287-311.
[34] 马毅. 轻量级 Web 服务器的实现与应用[D].西北大学,2008.
[35] 宋立昊. 基于线程池的 Web 服务器实现和监测[D].吉林大学,2011.
[36] H. Wu, C. Tan and H. Wang. Building a High-performance Communication Framework for Network Isolation System[C]. IEEE International Conference on Networking, Sensing and Control, 2008:1086-1091.
[37] G. X. Qin, J. Wang, T. Yuan, Q. Lv and X. B. Li. Design of Workflow Working Time Server Supporting Highly Concurrent Request[J]. 2009 WRI World Congress on Software Engineering, Xiamen, 2009:104-108.
[38] V. Phatak and B. R. Dongaonkar Jr. Applying Hyperthreading Technology for Evaluating the Performance of HTTP Server for Stored Audio/Video Retrieval[C]. Second International Conference on Emerging Trends in Engineering & Technology, 2009:644-647.
[39] X. Wang. Technical Analysis of High-Capacity and Concurrency Server Groups[C], 2010 International Conference on E-Product E-Service and E-Entertainment, 2010:1-4.
[40] P. Ying and W. Fang. Design of and Research on Distributed OJ System Based on Linux[C]. Third International Conference on Measuring Technology and Mechatronics Automation, 2011: 942-945.
[41] E. Tope, P. Zavarsky, R. Ruhl and D. Lindskog. Performance and Scalability Evaluation of Oracle VM Server Software Virtualization in a 64 Bit Linux Environment[C]. IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing, 2011:1119-1124.
[42] H. Lan and X. Wang.Research and Design of Concurrent Web Server on Linux System[C]. International Conference on Computer Science and Service System, 2012:734-737.
[43] X. Wu and R. Gao. The Design and Analysis of High Performance Online Game Server Concurrent Architecture[C]. International Conference on Computer Science and Service System,

2012:1662-1665.

[44] Jun Li and Menghan Lu. The performance optimization and modeling analysis based on the Apache Web Server[C]. Proceedings of the 32nd Chinese Control Conference, 2013:1712-1716.

[45] A. M. D. Aljohani, D. R. W. Holton and I. Awan. Modeling and Performance Analysis of Scalable Web Servers Deployed on the Cloud[C]. Eighth International Conference on Broadband and Wireless Computing, Communication and Applications, 2013:238-242.

[46] Y. Jin and M. Tomoishi. Web server performance enhancement by suppressing network traffic for high performance client[J]. 17th Asia-Pacific Network Operations and Management Symposium (APNOMS), Busan, 2015, pp. 448-451.

[47] 沈晓林. Web 服务器性能分析研究及优化方案[D].华东师范大学,2009.

[48] 刘雪梅. 服务器端软件性能分析和诊断方法研究[D].哈尔滨工程大学,2010.

[49] Prakash P, Biju R and M. Kamath. Performance analysis of process driven and event driven Web servers[C]. IEEE 9th International Conference on Intelligent Systems and Control (ISCO), 2015:1-7.

致谢

光阴荏苒，两年半的研究生生涯转瞬即逝，在这短短的两年半里，我不仅在课堂上学习了专业知识，也通过实习认识到了工作的不易，经历了学校到社会的转变，成长了许多，也收获了很多。谨以本文向所有关心我与我关心的人们表示最诚挚的感谢与最美好的祝愿。

首先，最需要感谢的是我的导师，马卫东博士。在研究生的学习与实习期间，马博士给予了我大量的指导与帮助，宽容与理解。马博士扎实的专业知识、严谨的科学态度、精益求精的科研精神和宽广的胸怀都让我受益匪浅，需要我在以后的工作生活中去不断学习与模仿。无论是论文选题、项目开发还是论文撰写，都是在马博士的悉心指导下完成。所以在论文完成之际，对马老师表示由衷的感谢与祝福！

在此由衷感谢仇树田高工和梁雪瑞师兄、翟宇佳师兄以及研究生部的老师与同学们，正是你们开阔我的思路，丰富我的视野，一步一步引领我完成设计，在学习和生活上给与了我很大帮助，愿友谊长存。

另外，我要感谢我的父母。感谢他们在精神上和生活上对我的支持与鼓励，让我全身心投入到学习和工作中。使得我能够给学习生涯画上一个完美的句号。最后向百忙之中评审论文和参加答辩的老师表示深深的感谢。

附录 1 攻读硕士学位期间参与的项目和发表的论文

[1] 陈沛,一种基于 Nginx 的负载均衡算法实现.[J],电子设计工程,2016.

附录 2 主要英文缩写语对照表

缩略语	中文全称	英文全称
AMPED	非对称多进程事件驱动结构	Asymmetric Multi-Process Event Driven
API	应用程序编程接口	Application Programming Interface
DNS	域名系统	Domain Name System
HTTP	超文本传输协议	Hyper Text Transfer Protocol
JSON	JS 对象表示法	JavaScript Object Notation
QPS	每秒查询率	Query Per Second
SPED	单进程事件驱动	Single Process Event Driven
SSL	安全套接层	Secure Sockets Layer
TCP	传输控制协议	Transmission Control Protocol
TTL	生存时间值	Time To Live
URL	统一资源定位符	Uniform Resource Locator