

**毕业论文(设计)**

**题目名称：基于Linux的WebServer服务器的设计与开发**

**题目类型： 毕业设计**

**学生姓名： 丁斌斌**

**院 (系)： 计算机科学学院**

**专业班级： 计科12002班**

**指导教师： 朱朝霞**

**辅导教师：**

**时 间： 2023-11-18 至 2024-05-30**

目 录

任务书………………………………………………………………………….Ⅰ

开题报告……………………………………………………………………….Ⅱ

指导教师审查意见…………………………………………………………….Ⅲ

评阅教师评语………………………………………………………………….Ⅳ

答辩会议记录 …………………………………………………………………Ⅴ

中文摘要……………………………………………………………………….Ⅵ

外文摘要……………………………………………………………………….Ⅶ

1 引言………………………………………….……………………………….1

1.1 选题背景…………………………………………………………………1

1.2 目的和意义………………………………………………………………1

1.3 技术要求和设计范围……………………………………………………1

1.4 发展现状…………………………………………………………………1

2 方案论证………………………………………….………………………….2

2.1 WEB 建站思想……………………………………………………………2

2.2 WEB 系统实现技术………………………………………………………4

2.3 运行环境和开发工具的选择……………………………………………5

3 过程论述………………………………………….………………………….7

3.1 需求分析…………………………………………………………………7

3.2 虚拟商场的总体设计 …………………………………………………11

3.3 虚拟商场的详细设计 …………………………………………………18

3.4 实现 ……………………………………………………………………20

3.5 网站发布与维护 ………………………………………………………20

3.6 虚拟商场安全性设计 …………………………………………………20

4 虚拟商场的实现过程………………………………………….……………21

4.1 数据库设计和连接的实现 ……………………………………………21

4.2 虚拟商场系统的实现 …………………………………………………22

5 站点运行情况………………………………………….……………………28

6 结束语………………………………………….……………………………28

参考文献……………………………………………………………………….28

致谢…………………………………………………………………………….29

附录…………………………………………………………………………….30

**基于Linux的WebServer服务器的设计与开发**

学 生：丁斌斌，计算机科学学院

指导教师：朱朝霞，计算机科学学院

【摘要】研究了用C++实现的Web服务器的关键技术，利用IO复用技术Epoll与线程池实现多线程的Reactor高并发模型，利用正则与状态机解析HTTP请求报文，实现处理静态资源的请求，通过理论知识的研究，在Linux系统下设计了一个Web服务器，在实现的过程中利用了研究的关键技术，并通过webbenchh压力测试可以实现上万的QPS，证明构建的服务器是可以承受一定的高并发访问的。

【关键词】Web服务器；高并发模型；线程池； IO复用技术；压力测试

**Design and Development of WebServer Server Based on Linux**

Student：Ding binbin，The College of Computer Science

Teacher：Zhu zhaoxia，The College of Computer Science

【Abstract】Studied the key technologies of a web server implemented in C++, implemented a multi threaded Reactor high concurrency model using IO reuse technology Epoll and thread pool, parsed HTTP request packets using regularization and state machines, and processed static resource requests. Through theoretical knowledge research, designed a web server in Linux system, utilized the key technologies studied in the implementation process, and achieved tens of thousands of QPS through webbench stress testing, proving that the constructed server can withstand certain high concurrency access.

【Keywords】 Web server；High concurrency model；thread pool；IO multiplexing technology；Stress testing

**基于Linux的WebServer服务器的设计与开发**

**1 绪论**

**1.1 课题背景及研究意义**

近些年来，随着互联网技术不断的发展和成熟，Web技术应用在各个领域，具有良好的发展趋势，包括电子商务、社交网络、在线教育、金融科技等。通过打开浏览器访问一个简单的网站，用户就能获取到丰富的信息，了解天下大事，而无需离开家门。中国互联网络信息中心2023年8月28日在京发布第52次《中国互联网络发展状况统计报告》。报告显示，截至2023年6月，我国网民规模达10.79亿人，较2022年12月增长1109万人，互联网普及率达76.4%，各类互联网应用正在持续发展。

Web服务器是Web应用的核心组件之一，扮演着至关重要的角色。传统的Web服务器通常采用基于进程的模型来处理请求。在这种模型中，对于每个传入的请求，服务器会为其分配一个独立的进程来处理。处理完结果之后再返回给客户端，这种处理模式效率是非常低下的，在互联网高速发展的时代，许多Web系统的访问量已经大幅增加，从百万次到千万次甚至更多。这种情况下，传统的基于进程的Web服务器模型可能面临一些挑战，因为它可能无法有效地处理如此大量的并发请求。因此，研究一些更高效的并发处理模型可能更为合适。

**1.2** **国内外研究现状**

自蒂姆·伯纳斯·李在1989年发明了万维网以来，经历了从最初的静态页面服务到高性能、高可靠性Web服务器的发展，Web服务器不仅能够提供稳定的服务和数据安全保障，还能够抵御网络攻击、优化性能，符合法规和标准等，为互联网企业的发展提供了有力支持。

早期的Web服务器中广泛采用带主进程的多进程服务器，这种模型也比较简单直观，易于实现和理解。其中，NCSA开发的httpd就是一个典型的带有主进程的多进程服务器。在这种服务器中，主进程负责监听和接收新的连接请求，而从属进程则负责实际的请求处理，从主进程获取请求并进行相应的事务处理，然后将处理结果返回给客户端。

现如今多进程服务器不再区分是否带有主进程，服务器预先创建了多个进程，并在每个进程中都开始监听来自客户端的连接请求。当有新的请求到达时，操作系统会

将请求分配给一个空闲的进程，这个进程调用accept()方法接收HTTP请求，然后进行处理和响应。这种模型下，每个进程都是相对独立的，可以并发处理多个请求，提高了服务器的并发能力和响应速度。Apache作为市场上应用最广泛的Web服务器之一，采用了这种预先派生多个进程的多进程模型。它具有稳定性高、性能优异、功能丰富等特点，被广泛应用于各种Web应用场景中。

由于多进程服务器在处理大量并发请求时存在一些缺点，单进程事件驱动（Single Process Event-Driven，SPED）服务器被设计出来以克服这些问题。在SPED服务器中，单个进程负责监听所有的HTTP请求，并采用事件驱动的方式进行处理。当有新的请求到达时，服务器会触发相应的事件，然后调用相应的处理程序进行处理。由于只有一个进程，避免了进程间的切换开销，同时利用事件驱动机制能够更有效地处理并发请求。尽管SPED服务器在处理大量并发请求时表现不如多进程服务器，但在请求量较小的情况下，由于其简单、轻量的设计，可以获得满意的性能表现。这种服务器模型在某些场景下具有一定的优势，尤其是对于资源受限、请求量不高的情况。

在面对高访问量的情况下，单进程服务器无法满足需求，需要采用更高效的并发处理模型。为了尽可能地解决单进程服务器的性能瓶颈问题，多线程模型被提了出来。典型的多线程服务器包括俄罗斯研究员Igor Sysoev开发的Nginx，以及德国领导开发的Lighttpd等。这些服务器采用了多线程的并发模型，能够更好地适应大规模并发的请求场景，同时充分利用多核处理器的性能优势，提高了服务器的响应速度和并发处理能力。

对比多进程服务器和多线程服务器，可以发现它们各有优缺点，适用于不同的应用场景。对于需要稳定性和可靠性较高、能够充分利用多核CPU的场景，多进程服务器可能更合适；而对于对并发处理能力和资源利用率要求较高的场景，多线程服务器可能更适合。

多线程服务器在应对高并发请求、提高服务器性能和资源利用率等方面的优势在市场上不断地扩大，许多国内外知名站点如淘宝、网易、优酷、Facebook、GitHub等都搭建了基于Nginx的服务器框架。Nginx作为一个轻量级、高性能的多线程服务器，在处理大规模并发请求时表现出色，因此受到了越来越多网站的青睐，成为了当今互联网行业中的主流服务器之一。

**1.3 论文主要内容及创新点**

本论文主要研究了一种基于Linux的高性能多线程WebServer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）研究了重要的网络通信协议，包括HTTP、TCP等重要协议，这是建立高性能Web服务器的基础。通过研究这些内容，可以建立起对Web通信协议和服务器架构的深入理解。最后研究出一种基于Linux的高性能多线程WebServer服务器，利用IO复用技术Epoll与线程池实现多线程的Reactor高并发模型。

（2）了解如何有效地管理和复用线程资源，实现线程池功能。设计了一种智能调整规模的线程池：它会根据系统的负载情况灵活地增减线程数量。当系统负载上升或者下降时，线程池会相应扩大或减小规模以应对更多的任务需求，保证系统性能；

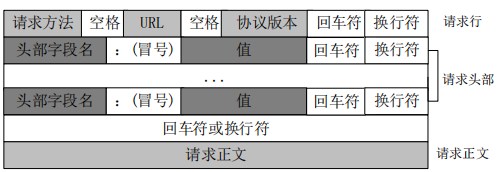
（3）编写了服务器的源代码并进行了测试，以确保其在不同条件下的正常运行和性能表现。经过测试，WebServer服务器表现出良好的性能，在面对高并发情况下仍能正常运行。

**2 Web服务器的原理**

**2.1 计算机网络协议的深入分析和探究**

**2.1.1 HTTP协议**

HTTP（Hypertext Transfer Protocol，超文本传输协议）是一种用于传输超文本数据（例如HTML、图片、视频等）的应用层协议。它是互联网上应用最为广泛的协议之一，主要用于Web浏览器和Web服务器之间的通信。HTTP协议是无连接的，即每次请求和响应之间都是独立的，服务器在响应后会立即关闭连接。同时HTTP协议是无状态的，即服务器不会保存任何关于客户端的状态信息。每个请求都是独立的，服务器无法识别两个请求是否来自同一个客户端。



**图1 HTTP请求报文格式**

HTTP协议主要负责客户端向服务器发送请求，并接收服务器返回的响应。如图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。



**图2 HTTP响应报文格式**

如图2，HTTP响应报文是服务器发送给客户端的消息，它由三个主要部分组成：状态行、响应头部和消息体。其中，状态代码只有三位，表1展示了HTTP响应状态码的5中取值。

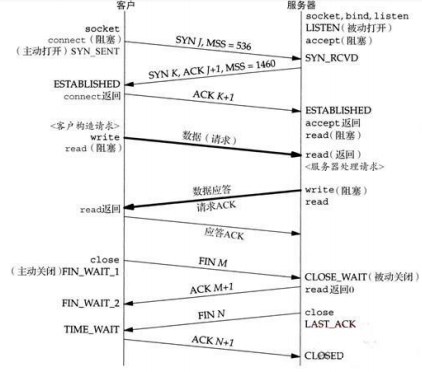
|  |  |  |
| --- | --- | --- |
| 数据库表 | 名称 | 作用描述 |
| Goods | 商品 | 存储商品信息 |
| GoodsBigType | 商品大类 | 商品大类信息 |
| GoodsSmallType | 商品小类 | 商品小类信息 |
| Basket | 购物车 | 顾客已选购的物品 |

**表1 HTTP响应**

**2.1.2 TCP协议**

TCP（Transmission Control Protocol，传输控制协议）是一种面向连接的、可靠的、基于字节流的传输层协议。它是互联网协议套件中的一个核心协议，负责在网络上可靠地传输数据。TCP协议在互联网中的应用非常广泛，例如在Web浏览器和服务器之间的HTTP通信、电子邮件传输协议SMTP、文件传输协议FTP等都是基于TCP协议的。由于TCP提供了可靠性、流量控制和拥塞控制等功能，因此它适用于各种需要可靠传输的应用场景。

TCP连接的建立与销毁是TCP通信的重要过程，包括连接的建立和连接的关闭。下面是TCP连接的建立与销毁过程的详细描述。



**图3 TCP的连接与断开**

流量控制是TCP协议中的一项重要功能，用于控制数据的传输速率，以避免发送方发送数据过快导致接收方无法处理的情况，或者网络拥塞引起的数据丢失。TCP通过使用滑动窗口机制来实现流量控制。

拥塞控制是TCP协议中的一项重要功能，用于调节数据在网络中的传输速率，以避免网络拥塞导致的数据丢失、延迟和性能下降。TCP通过使用拥塞窗口和拥塞避免算法来实现拥塞控制，保证数据的可靠传输和网络的稳定性。

**2.2 网络程序设计基础**

**2.2.1 Socket编程**

Socket（套接字）是一种用于网络通信的编程接口，它允许程序员创建网络连接、发送和接收数据。Socket在网络编程中起着至关重要的作用，它提供了一种通用的方法来实现不同机器间的通信。

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 {

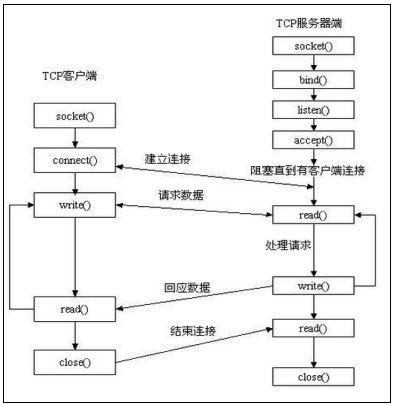
short int sin\_family; // 地址族，通常设置为AF\_INET表示IPv4

sin\_port; // 端口号，使用网络字节序（大端序）表示

struct in\_addr sin\_addr; // IP地址

};

Socket的本质是一种编程接口，用于实现网络通信。它在操作系统内核中提供了一组函数或系统调用，使程序能够创建、连接、发送和接收数据。通过Socket接口，程序可以与其他计算机上的进程进行通信，实现数据交换和协作。图4展示了客户端与服务器通过Socket建立连接的过程。首先，客户端和服务器分别创建一个未定义的Socket套接字。接着，它们分别使用bind()函数将Socket返回的文件描述符fd与特定的网络地址和端口进行绑定，从而完成套接字的初始化。然后，服务器端使用listen()函数开始监听Socket套接字，等待客户端的连接请求。当服务器监听到客户端发送的连接请求时，它调用accept()函数接受客户端的请求，并开始处理客户端的连接。



**图4 Socket连接过程**

**2.2.2 进程与线程**

进程是操作系统中的一个基本概念，它是程序执行的一个实例。换句话说，进程是正在运行的程序的抽象。每个进程都有自己的地址空间、内存、代码和数据，以及操作系统所需的其他资源。

线程是程序执行流的最小单元，也是操作系统能够进行运算调度的最小单位。在一个进程中，可以包含多个线程，这些线程共享进程的资源，如内存空间、文件句柄等。线程之间可以并发执行，从而提高了程序的性能和效率。

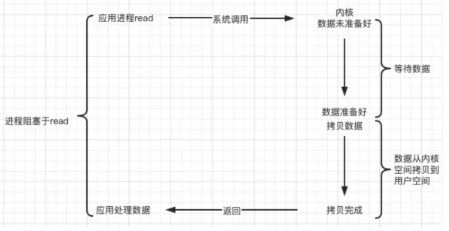
相比于进程，线程具有以下几个优势：

1. 资源开销较小：线程是进程的子集，线程之间共享进程的资源，如内存空间、文件句柄等。因此，创建和销毁线程的开销通常比创建和销毁进程要小得多，线程的切换也更加快速高效。
2. 并发执行：由于线程共享进程的地址空间和其他资源，线程之间可以并发执行，从而提高了程序的性能和效率。多个线程可以同时执行不同的任务，实现并行处理，加快了程序的执行速度。
3. 更好的响应性：由于线程可以并发执行，可以将耗时的任务放在后台线程中执行，从而提高了程序的响应性。例如，可以使用一个线程处理用户界面的交互操作，另一个线程处理后台的数据处理任务，以保持界面的流畅和响应。
4. 更方便的数据共享：线程之间共享进程的地址空间，因此它们可以更方便地共享数据和通信。这使得线程之间的数据共享更加简单直接，不需要使用额外的机制来实现进程间通信。
5. 更好的资源利用率：由于线程共享进程的资源，可以更有效地利用系统资源。例如，多个线程可以共享相同的代码段和数据段，减少了内存的开销，提高了系统的资源利用率。

**2.2.3 服务器 IO 模型**

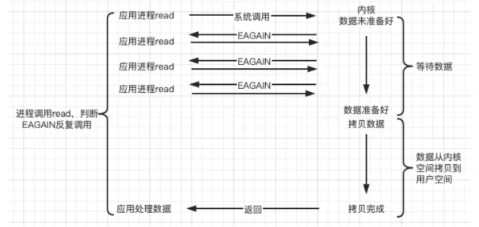
高性能WebServer服务器的设计需要高性能的IO模型，常见的Unix/Linux上的五种IO模型有以下几种：

（1）阻塞（blocking）：调用者调用了某个函数，等待这个函数返回，期间什么也不做，不停的去检查这个函数有没有返回，必须等这个函数返回才能进行下一步动作。



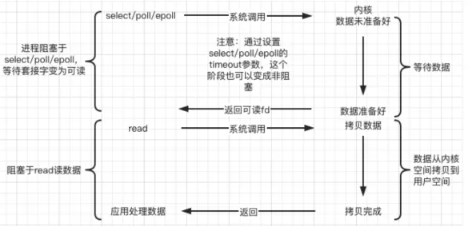
**图5 阻塞模型**

1. 非阻塞（non-blocking）：非阻塞等待，每隔一段时间就去检测IO事件是否就绪。没有就绪就可以做其他事。非阻塞I/O执行系统调用总是立即返回，不管事件是否已经发生，若事件没有发生，则返回-1，此时可以根据 errno 区分这两种情况，对于accept，recv 和 send，事件未发生时，errno 通常被设置成 EAGAIN。



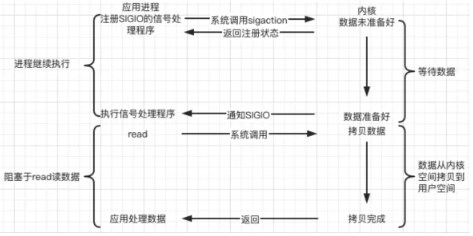
**图6 非阻塞模型**

（3）IO复用（IO multiplexing）：Linux 用 select/poll/epoll 函数实现 IO 复用模型，这些函数也会使进程阻塞，但是和阻塞IO所不同的是这些函数可以同时阻塞多个IO操作。而且可以同时对多个读操作、写操作的IO函数进行检测。直到有数据可读或可写时，才真正调用IO操作函数。



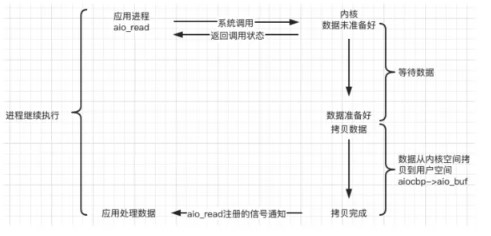
**图7 IO复用模型**

（4）信号驱动（signal-driven）：Linux 用套接口进行信号驱动 IO，安装一个信号处理函数，进程继续运行并不阻塞，当IO事件就绪，进程收到SIGIO 信号，然后处理 IO 事件。内核在第一个阶段是异步，在第二个阶段是同步；与非阻塞IO的区别在于它提供了消息通知机制，不需要用户进程不断的轮询检查，减少了系统API的调用次数，提高了效率。



**图8 信号驱动模型**

（5）异步（asynchronous）：Linux中，可以调用 aio\_read 函数告诉内核描述字缓冲区指针和缓冲区的大小、文件偏移及通知的方式，然后立即返回，当内核将数据拷贝到缓冲区后，再通知应用程序。

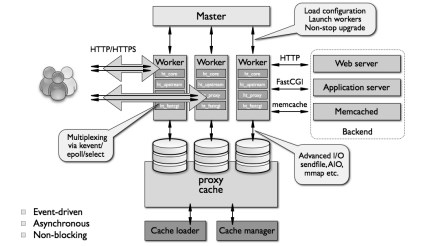


**图9 异步模型**

**2.3 Nginx 架构研究**

Nginx是一个高性能的开源Web服务器，也是一个反向代理服务器和负载均衡器。它最初由Igor Sysoev创建，并于2004年首次公开发布。Nginx专注于高并发性能和低内存消耗，因此在处理静态文件、反向代理和负载均衡等方面表现出色。它已经成为许多网站和应用程序的首选Web服务器之一，因为它的性能和可靠性。所以对Nginx的研究有助于学习多线程高性能Web服务器的设计。

Nginx支持多进程与多线程工作方式，在Nginx的架构中，Master进程负责管理和监控多个Worker进程，而每个Worker进程都是独立的进程，负责处理客户端的请求。这种多进程的设计有助于提高Nginx的并发处理能力和稳定性，因为每个Worker进程都是相互独立的，它们之间不会共享内存空间，从而减少了因共享状态而可能引起的竞争和死锁等问题。



**图10 Nginx架构模型**

Nginx默认采用的是pre-fork方式工作，这是一种常见的多进程模型。在pre-fork模型中，Master进程在启动时会创建多个Worker进程，每个Worker进程独立处理客户端请求。这些Worker进程在处理请求时是相互独立的，它们之间不会共享内存空间，因此不会出现因共享状态而引起的竞争和冲突问题。

多路IO复用的核心思想是通过一个系统调用，让一个进程能同时监控多个IO事件，而不是每个事件都单独地等待。在Nginx的场景中，它利用了Linux系统提供的Epoll机制来实现多路IO复用。当一个客户端连接到服务器时，服务器会将这个连接添加到Epoll的监听列表中。而后，Epoll会监视这个连接上的各种IO事件，比如可读事件（表示有数据可以从连接中读取）、可写事件（表示可以向连接中写入数据）等。当有任何一个事件发生时，Epoll会通知Nginx主进程，然后Nginx主进程会将相应的任务分配给可用的Worker进程来处理。通过利用Epoll的多路IO复用机制，Nginx可以同时监听大量的连接，并且能够高效地处理这些连接上的IO事件，从而实现高性能和高并发处理能力。这种设计使得Nginx能够在保持低资源消耗的同时，处理大规模的并发请求。

**2.4 本章小结**

本章首先介绍了服务器中的关键网络协议，包括HTTP和TCP，这是理解网络通信的基础。接着对网络程序设计的基础进行了分析，涉及了Socket编程通信、进程、线程和线程池的概念，以及各种服务器IO模式的原理和应用。最后，通过学习当今最流行的轻量级服务器Nginx的架构设计，深入了解了Nginx如何实现IO多路复用。综合而言，本章系统介绍了Web服务器的关键原理与技术，为进一步设计轻量级高性能的Web服务器打下了基础。

**3 Web 服务器的需求分析与设计**

**3.1 高性能Web服务器需求分析**

**1. 参数动态配置**

在许多UNIX服务器中，用户通常使用命令行与服务器进行交互。为了确保服务器在启动之前完成必要的配置，我们需要设计服务器能够在启动时接受命令行输入，以便用户可以指定一个或多个参数来进行动态配置。如果没有提供特定的配置参数，服务器将会以默认方式启动。

命令行参数配置的格式如下：该命令表示服务器的端口号为1316，开启ET模式，超时时间为60000毫秒，关闭优雅退出，Mysql端口3306，账号，密码，数据库，sql连接池数量为12，线程池线程数量为6，开启日志，日志系统等级为1，日志异步队列容量为1024。

WebServer server(  
 1316, 3, 60000, false,   
 3306, "root", "root", "webserver",   
 12, 6, true, 1, 1024);

**2. 支持并发连接**

支持并发连接是Web服务器的基本要求之一。这意味着服务器能够同时处理多个客户端的连接请求，而不会因此而降低性能或出现响应延迟。通过支持并发连接，服务器可以更高效地处理大量的请求，提高用户的访问体验，并确保网站或应用程序的稳定性和可靠性。

**3. HTTP 协议支持**

HTTP/0.9仅允许使用GET方法，但已经较少使用。目前，主流的HTTP协议版本包括HTTP/1.0和HTTP/1.1。通常情况下，HTTP/1.0主要应用于代理服务器，而HTTP/1.1则是目前最为普遍的版本。HTTP/1.1默认启用持久连接，这意味着在单个连接上可以发送和接收多个HTTP请求和响应。在我们设计的服务器中，只需支持最为普遍的HTTP/1.1版本。如果客户端请求的协议版本与服务器不兼容，则服务器应通过适当的状态码向用户返回相应的错误信息。

**4. 请求方法支持**

HTTP/1.1共定义了八种请求方法，它们分别是：GET、PUT、POST、DELETE、HEAD、TRACE、OPTIONS、CONNECT，这些请求方法定义了客户端与服务器之间的交互方式，允许客户端对服务器上的资源进行各种不同类型的操作。对于本文设计的基于Linux的WebServer服务器来说，其主要职责是为用户提供静态或动态的网络页面服务，因此，GET和POST请求方法是必需的，而其他方法是可选择支持的。

**5. 支持线程池**

通过使用线程池，可以避免频繁地创建和销毁线程，减少了系统资源的消耗和性能开销。同时，线程池可以根据实际需求动态调整线程数量，以适应不同负载下的需求，提高系统的性能和稳定性。

**6. 日志功能**

服务器日志功能是服务器软件记录和存储关于其运行状态和用户请求的信息的功能。这些日志对于监控服务器的性能、诊断问题以及跟踪用户活动都非常重要。服务器日志功能通常包括：访问日志、错误日志、性能日志、安全日志、自定义日志。通过分析和解释服务器日志，管理员可以了解服务器的运行情况和用户行为，及时发现和解决问题，提高服务器的性能、稳定性和安全性。

**7. 安全性保证**

OpenSSL提供了一套安全套接字编程接口（SSL/TLS API），可以方便地在应用程序中实现安全的网络通信。开发人员可以使用这些API来构建安全的客户端和服务器程序，保护敏感数据的传输和存储。OpenSSL是一个功能强大的加密库，为开发人员提供了丰富的安全功能和工具，帮助他们构建安全可靠的应用程序和网络系统。

**8. 平台兼容性支持**

服务器平台兼容性支持指的是服务器软件或应用程序能够在不同操作系统和硬件平台上正常运行和提供相同的功能。在开发服务器软件时，通常会考虑到跨平台兼容性，以确保尽可能多的用户能够使用该软件。

**3.2 Web服务器总体设计**

本文的主要任务是设计一个基于Linux的WebServer服务器并编写代码实现，在此基础之上研究一种线程池，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11，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

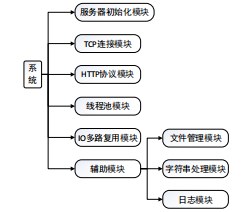
Socket连接断开与接收发送 TCP 报文数据等工作；

（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；

（6）日志模块：用于记录和存储系统的运行状态、事件和错误信息。



**图11 服务器总体设计图**

**3.3 Web 服务器主要模块设计**

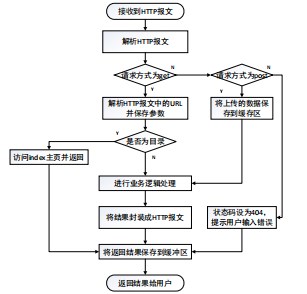
**3.3.1 服务器初始化模块**

本文中设计的服务器在实现HTTP服务器功能时，直接在传输层（Transport Layer）上进行操作。服务器利用TCP连接作为基础，建立与客户端的通信通道。在TCP连接建立后，服务器通过解析客户端发送的HTTP请求报文，执行相应的操作并生成HTTP响应报文，然后将响应报文发送回客户端。

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。TCP协议模块处理流程图如图12。

**3.3.2 HTTP协议模块**

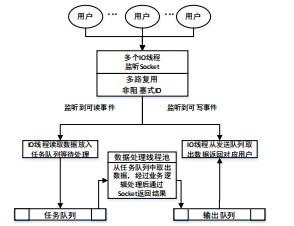
HTTP协议模块是负责处理和解析HTTP请求和响应的组件。它通常包括请求解析、响应生成、请求处理、响应处理、错误处理、会话管理的功能。它是服务器中的一个核心组件，负责处理和解析HTTP请求和响应，以及执行相应的请求处理逻辑。它通过与其他组件协同工作，实现了服务器对HTTP协议的完整支持，从而提供了高效、可靠的Web服务。图13展示了HTTP协议模块处理的一般流程。



**图13 HTTP请求处理流程图**

**3.3.3 高性能IO模块**

本文研究了一种基于Reactor的高并发模型。通过利用IO复用技术，如Epoll，在单个线程中处理多个并发连接。线程池中的线程负责具体的事件处理。例如，当有新连接到达时，线程池中的某个线程接受连接并将新的连接套接字添加到Epoll监听器中；当套接字上有数据可读时，线程池中的某个线程读取数据并进行处理；当套接字上可写时，线程池中的某个线程发送数据给客户端。通过这种方式，服务器能够利用Epoll实现事件驱动的高效IO操作，同时通过线程池实现多线程并发处理，提高了系统的并发性能和吞吐量。具体示意图如下：



**图14 服务器编程模型示意图**

**3.3.4 线程池模块**

线程池模块是服务器系统中的重要组件，用于管理和调度线程，实现任务的并发处理。它能够提高系统的并发性能和吞吐量，同时有效地利用系统资源，是实现高性能服务器的关键技术之一。

本文研究了一种可高效运行的线程池模型，该线程池模型采用一个任务队列来存储待执行的任务。任务队列采用先进先出（FIFO）的原则，确保任务按照到达顺序执行。任务队列的大小可以根据系统的负载情况和性能需求进行动态调整，以确保线程池的高效运行。线程池模型通过一个线程池管理器来创建和管理线程。线程池管理器根据配置参数创建一定数量的线程，并在需要时动态地调整线程数量。线程池管理器还负责监控线程的状态和活跃度，及时回收空闲线程，以减少资源的浪费。当任务到达时，线程池将任务分配给空闲线程进行处理。每个线程在处理完任务后会从任务队列中取出下一个任务，直到任务队列为空或线程池关闭为止。线程池采用预先创建线程的方式来避免线程创建和销毁的开销，提高了任务的处理效率。

**3.3.4 日志模块**

服务器中的日志模块是一个非常重要的组件，它负责记录和管理服务器运行时产生的各种日志信息，为系统的监控、故障排查和性能优化提供了重要的支持。一个高效可靠的日志模块可以帮助管理员及时发现和解决问题，提高系统的稳定性和可靠性。

本文设计的日志模块实现以下基础功能：

（1）设置了四种日志级别，包括"ERROR"，"WARN"，"INFO"，"DEBUG"；

（2）日志模块可以将记录的日志信息输出到多种目标，包括控制台、日志文件、远程服务器、数据库等。

（3）可以对记录的日志信息进行格式化处理。

（4）日志模块不阻塞正常的执行流程；

（5）在记录大量日志的同时对系统性能的影响小，不影响系统的稳定性和可靠性。

**3.3.5 缓冲区模块**

利用标准库容器std::vector<char>封装char，实现自动增长的缓冲区，可以方便地实现动态增长的缓冲区，避免了手动管理内存空间的复杂性，提高了代码的可读性和可维护性。

**3.3.6 定时器模块**

每个连接对象需要包含一个定时器结构，用于记录连接的超时时间点。使用小根堆（最小堆）来管理定时器。小根堆可以保证堆顶元素是最小的，也就是最先要超时的连接。当新的连接建立时，根据连接的超时时间，在小根堆中插入一个定时器节点。定时器模块会定期检查小根堆的堆顶元素，如果堆顶元素的超时时间小于当前时间，则表示该连接已经超时，需要关闭。定时器模块将超时的连接对象从小根堆中删除，并执行关闭连接的操作。通过以上步骤，基于小根堆实现的定时器可以高效地管理连接的超时，实现自动关闭超时的非活动连接，从而提高服务器的性能和稳定性。

**3.4 本章小结**

本章首先对需求分析了解了高性能轻量级Web服务器的要求，接着在此基础上进行了整体设计，然后对服务器进行了模块化设计。研究过程中提出了一种创新的服务器编程模型，它将线程池与非阻塞IO、IO多路复用技术相结合，以确保在数据计算和网络IO方面都具有高效率。另外，还设计了一种灵活动态的线程池，并结合负载均衡算法，使得任务能够分配到合适的线程上，从而提升服务器性能。接下来的步骤是将所设计的Web服务器进行具体的代码实现。

**4 Web 服务器的具体实现**

**4.1 服务器初始化模块的实现**

服务器初始化时，首先需要加载配置文件，读取服务器的配置信息，包括网络端口、IP地址、日志路径、线程池大小、数据库连接信息等。这些配置信息可以影响服务器的运行方式和性能参数，因此配置加载是初始化的第一步。在服务器启动时，需要初始化日志模块，配置日志的输出方式、日志级别、日志格式等参数。服务器通常是基于网络进行通信的，因此需要初始化网络模块，包括创建监听套接字、绑定IP地址和端口、设置套接字选项、启动监听等操作。网络模块的初始化是服务器初始化的重要组成部分，它负责建立服务器与客户端之间的通信通道。

服务器通常使用线程池来处理客户端请求，因此需要初始化线程池。线程池的初始化包括创建一定数量的线程、设置线程属性、初始化任务队列等操作。线程池的初始化是服务器初始化的关键步骤，它决定了服务器能够处理的并发请求数量。如果服务器需要与数据库进行交互，就需要初始化数据库连接池。数据库连接池的初始化包括创建一定数量的数据库连接、设置连接属性、初始化连接池管理器等操作。数据库连接池的初始化是服务器初始化的重要组成部分，它提高了服务器与数据库之间的交互效率和性能。

服务器初始化模块可以完成对服务器各个组件的初始化工作，使服务器能够在启动时正常运行，并为后续的请求处理提供必要的基础环境。以下对服务器初始化模块中的主要函数及其作用作简单介绍：

1. WebServer(  
    int port, int trigMode, int timeoutMS, bool OptLinger,  
    int sqlPort, const char \*sqlUser, const char \*sqlPwd,  
    const char \*dbName, int connPoolNum, int threadNum,  
    bool openLog, int logLevel, int logQueSize);

这段代码是一个Web服务器的构造函数，负责对服务器进行初始化。在构造函数中，首先使用参数列表对成员变量进行了初始化，然后调用SqlConnPool::Instance()->Init()方法，传入数据库连接相关参数，对数据库连接池进行初始化，调用InitEventMode\_()方法，根据传入的触发模式参数初始化事件模式为ET模式，调用InitSocket\_()方法初始化套接字，调用Log::Instance()->init()方法初始化日志记录器，并输出初始化信息。总之，该构造函数完成了对服务器各个组件的初始化工作，包括网络模块、线程池、定时器、日志记录等，为服务器的正常运行奠定了基础。

1. void WebServer::Start();

通过epoll实现了事件驱动的服务器主循环，当有事件发生时，根据事件类型调用相应的处理函数进行处理。首先，设置epoll等待超时时间，如果timeoutMS\_大于0，表示设置了超时时间，调用timer\_->GetNextTick()获取下一个计时器超时时间。将该超时时间作为参数传入epoller\_->Wait(timeMS)，以等待事件发生。调用epoller\_->Wait(timeMS)等待事件发生，返回事件数量eventCnt。在循环中处理事件，通过epoller\_->GetEventFd(i)获取事件对应的文件描述符fd，通过epoller\_->GetEvents(i)获取事件类型events。如果事件是监听套接字实例listenFd\_，则调用DealListen\_()处理监听事件。如果事件是错误事件或连接关闭事件，则调用CloseConn\_(&users\_[fd])关闭用户连接。如果事件是可读事件，则调用DealRead\_(&users\_[fd])处理读取事件。如果事件是可写事件，则调用DealWrite\_(&users\_[fd])处理可写事件。如果出现了意外的事件类型，则输出错误日志。循环直到isClose\_标志为true，即服务器关闭。

1. bool WebServer::InitSocket\_();

Web服务器的套接字初始化函数InitSocket\_()，负责创建、设置、绑定和监听套接字，以及将套接字添加到epoll实例中进行事件监听。

（4）void WebServer::DealWrite\_(HttpConn \*client);

将写事件处理任务添加到线程池中，实现了异步处理客户端写事件的功能，避免了在主线程中阻塞等待写事件的完成。

(5)void WebServer::DealRead\_(HttpConn \*client);

将读事件处理任务添加到线程池中，实现了异步处理客户端读事件的功能，避免了在主线程中阻塞等待读事件的完成。

**4.2 HTTP模块的实现**

HTTP模块是负责处理客户端发来的HTTP请求和向客户端发送HTTP响应的部分。本文设计了httpconn、httprequest、httpresponse这三种结构体来实现HTTP模块相应功能的实现。其中，httpconn类是对一个客户端连接的描述，而httprequest类和httpresponse类则分别是对请求报文和响应报文的描述。

以下是对HTTP模块中的主要函数及其作用作简单介绍：

1. void HttpConn::init(int fd, const sockaddr\_in& addr);

HTTP连接对象(HttpConn)的初始化函数init()，用于初始化一个HTTP连接对象。在函数中，使用assert()函数确保传入的文件描述符fd大于0，以确保文件描述符有效。每次初始化一个新的HTTP连接对象，连接计数器userCount会增加。这个计数器通常用于记录当前活跃的连接数。将传入的远程地址信息addr复制给HTTP连接对象的成员变量addr\_，用于记录客户端的地址信息。将传入的文件描述符fd赋值给HTTP连接对象的成员变量fd\_，用于标识该连接。清空HTTP连接对象的读写缓冲区，以确保在使用之前没有遗留的数据。将连接状态isClose\_设置为false，表示连接处于打开状态。最后，使用日志记录函数LOG\_INFO()记录连接建立的日志信息。

1. ssize\_t HttpConn::write(int\* saveErrno);

这段代码实现了HTTP连接对象(HttpConn)的写操作函数write()，用于将数据写入到套接字文件描述符中。使用do-while循环进行数据写入操作，直到所有数据都被写入或写入过程中出现错误为止。使用writev()函数将分散的数据一并写入到套接字文件描述符中，并返回写入的字节数。iov\_是一个iovec结构体数组，用于存储待发送数据的分散信息，iovCnt\_表示数组中元素的数量。如果写入的字节数小于等于0，表示写入失败，将错误码保存到指定的变量中，并跳出循环。如果待发送的数据量为0，表示传输结束，跳出循环。如果写入的字节数大于第一个缓冲区中的数据量，表示第一个缓冲区中的数据已经全部发送完毕，更新iov\_数组中的数据信息，并将写缓冲区中已发送的数据部分删除。否则，更新iov\_数组中的数据信息，并将写缓冲区中已发送的数据删除。如果是ET模式（边缘触发模式），或者待写入的数据量超过一定阈值（10KB），则继续循环写入数据。最后，返回实际写入的字节数len。

（3）bool HttpRequest::parse(Buffer &buff);

实现了HTTP请求对象(HttpRequest)的解析函数parse()，用于解析从客户端接收到的HTTP请求报文。

（4）bool HttpRequest::ParseRequestLine\_(const string &line);

实现了解析HTTP请求行的函数ParseRequestLine\_()，使用了正则表达式来匹配HTTP请求行中的方法、路径和HTTP版本。

（5）void HttpResponse::MakeResponse(Buffer& buff);

HTTP响应对象(HttpResponse)中的MakeResponse()函数，用于生成HTTP响应报文。

（6）void HttpResponse::AddStateLine\_(Buffer& buff)

HTTP响应对象(HttpResponse)中的AddStateLine\_()函数，用于向缓冲区中添加HTTP状态行。

**4.3 IO模块的实现**

IO模块通常负责处理与网络IO相关的操作，包括接收请求、发送响应等。这个模块通常是整个服务器的核心部分之一。本文使用了多路IO复用技术Epoll来实现高效处理大量并发连接的功能。设计了epoller类来保证Reactor 模式的实现。 在服务器启动后，通过epoll实现事件驱动的服务器主循环，当有事件发生时，根据事件类型调用相应的处理函数进行处理。如果事件是监听套接字实例listenFd\_，则调用DealListen\_()处理监听事件。如果事件是可读事件，则调用DealRead\_(&users\_[fd])处理读取事件。如果事件是可写事件，则调用DealWrite\_(&users\_[fd])处理可写事件。

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)；

创建一个新的 epoll 实例，并返回一个文件描述符，该描述符可以用于后续的 epoll 系统调用。

1. int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event)；

用于控制 epoll 实例上的事件。它可以用来注册新的文件描述符、修改已注册的文件描述符上的事件或者删除已注册的文件描述符。epfd 参数是 epoll 实例的文件描述符，通过 epoll\_create() 创建得到。op 参数表示要执行的操作，fd 参数是要添加、修改或删除的文件描述符。event 参数是一个指向 epoll\_event 结构体的指针，其中包含了要注册或修改的事件信息。函数成功执行返回 0，失败返回 -1，并设置相应的错误码。

（3） int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int

timeout)；

epoll\_wait() 函数用于等待 epoll 实例上的事件发生，并将就绪的事件填充到指定的事件数组中。epfd 参数是 epoll 实例的文件描述符，通过 epoll\_create() 创建得到。events 参数是一个指向 epoll\_event 结构体数组的指针，用于存储就绪的事件信息。maxevents 参数表示 events 数组的大小，即最多可以存储多少个就绪事件。timeout 参数指定等待事件发生的超时时间，单位为毫秒。可以有以下几种情况，如果 timeout 为正数，则表示等待指定的毫秒数后返回，即超时等待。如果 timeout 为 0，则表示立即返回，即非阻塞模式。如果 timeout 为 -1，则表示无限等待，直到有事件发生。函数成功执行返回就绪事件的数量，失败返回 -1，并设置相应的错误码。就绪事件的具体信息将存储在 events 数组中。

**4.4 线程池模块的实现**

线程池是一种管理线程的机制，用于提高并发任务处理的效率。其主要组成部分包括线程池管理器、工作队列和线程池工作线程。线程池管理器负责管理线程池的生命周期，包括创建、销毁线程池。工作队列用于存储需要执行的任务。通常采用队列结构，支持任务的入队和出队操作。当有任务需要执行时，会被放入工作队列中等待执行。线程池中的工作线程负责从工作队列中获取任务并执行。同时，使用互斥锁和条件变量等机制实现线程之间的同步和互斥，确保多个线程之间对共享资源的安全访问。

以下简单介绍下线程池模块中的主要函数以及结构体：

1. struct Pool {  
    std::mutex mtx;   
    std::condition\_variable cond;   
    bool isClosed;   
    std::queue<std::function<void()>> tasks;   
    };

这段代码定义了一个简单的线程池结构体 Pool，std::mutex mtx表示互斥量，用于实现对线程池的互斥访问，确保在多线程环境下对任务队列的安全访问。std::condition\_variable cond表示条件变量，用于实现线程的同步等待，当任务队列为空时，工作线程将会等待条件变量的通知。bool isClosed表示标志位，指示线程池是否已关闭。std::queue<std::function<void()>> tasks表示任务队列，存储待执行的任务。使用 std::function<void()> 表示任务的类型，支持任意可调用对象作为任务。

（2）explicit ThreadPool(size\_t threadCount = 8): pool\_(std::make\_shared<Pool>()) {  
 assert(threadCount > 0);  
 for(size\_t i = 0; i < threadCount; i++) {  
 std::thread([pool = pool\_] {  
 std::unique\_lock<std::mutex> locker(pool->mtx);  
 while(true) {  
 if(!pool->tasks.empty()) {  
 auto task = std::move(pool->tasks.front());  
 pool->tasks.pop();  
 locker.unlock();  
 task();  
 locker.lock();  
 }   
 else if(pool->isClosed) break;  
 else pool->cond.wait(locker);  
 }  
 }).detach();  
 }  
 }

这段代码定义了线程池的构造函数，使用了 C++11 中的 lambda 表达式来创建工作线程，并将其逻辑包装在一个 while 循环中。在构造函数中，接受一个默认参数 threadCount，表示线程池中的线程数量，默认值为 8。利用 std::make\_shared 创建了一个 Pool 对象，并通过 std::shared\_ptr 进行管理，pool\_ 是线程池的成员变量，用于共享线程池对象。根据传入的线程数量，循环创建工作线程。使用 lambda 表达式创建了一个新的线程，并立即分离（detach）它，以允许该线程独立执行。通过lambda 表达式的捕获列表捕获了 pool\_，以确保在 lambda 函数体中能够访问到线程池对象。

在每个工作线程中创建了一个独占锁 locker，以锁定线程池的互斥量，确保多个线程同时访问任务队列时的安全性。然后循环执行工作线程的主体逻辑。当任务队列不为空时，线程会从队列中取出任务并执行。当线程池已关闭时，退出循环。当任务队列为空且线程池未关闭时，线程进入等待状态，等待条件变量的通知。这样，每个工作线程在一个循环中不断地从任务队列中取出任务执行，如果任务队列为空且线程池未关闭，则进入等待状态，直到有新任务加入或线程池被关闭。

**4.5 日志模块的实现**

本项目中，使用单例模式创建日志系统，对服务器运行状态、错误信息和访问数据进行记录，该系统可以实现按天分类，超行分类功能，可以根据实际情况分别使用同步和异步写入两种方式。其中异步写入方式，将生产者-消费者模型封装为阻塞队列，创建一个写线程，工作线程将要写的内容push进队列，写线程从队列中取出内容，写入日志文件。

日志模块的实现较为简单，本文在服务器初始化的同时可以设置日志记录的等级，同时本文设计包括"ERROR"，"WARN"，"INFO"，"DEBUG"的四种日志等级，同时为了保证日志系统的功能正常运转，实现了log类来保证，在log类中可以设置log文件存放的位置。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，以便后期开发人员查阅。

**4.6 定时器模块的实现**

在本项目中，服务器主循环为每一个连接创建一个定时器，并对每个连接进行定时。另外，利用升序时间链表容器将所有定时器串联起来，若主循环接收到定时通知，则在链表中依次执行定时任务。

以下简单介绍下定时器模块中的主要函数：

（1）void HeapTimer::siftup\_(size\_t i);

实现了堆的向上调整操作。

1. void HeapTimer::add(int id, int timeout, const TimeoutCallBack& cb);

当服务器有新的客户端添加进来时，创建新的定时器节点。

1. void HeapTimer::tick();

处理堆中已经超时的定时器节点。

**4.7 本章小结**

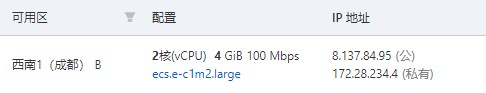
本章主要内容是在前文的需求分析与设计的基础上，对基于Linux的WebServer服务器的主要模块进行了具体实现与详细介绍，包括服务器初始化模块、HTTP协议模块、高性能IO模块、线程池模块、日志模块已经定时器模块。至此完成了本文对于基于Linux的WebServer服务器的研究及实现，接下来的章节是对于所设计Web服务器的性能测试以及优化。

**5 性能测试与结果分析**

**5.1 测试环境**

对于服务端而言，本文采用阿里云服务器，操作系统为ubuntu18.04LTS,在此操作系统上自动配置了数据库mysql5.7.41。如图15，创建服务器实例。对于客户端而言，使用了Webbench对服务器的性能进行测试，Webbench 是 Linux 上一款知名的、优秀的 web 性能压力测试工具。它是由Lionbridge公司开发。它的基本原理：Webbench首先fork出多个子进程，每个子进程都循环做 web 访问测试。子进程把访问的结果通过pipe告诉父进程，父进程做最终的统计结果。





**图15 服务器实例**

所以，对本文设计的 Web 服务器进行测试的硬件环境如图16。



**图16 服务器测试环境**

**5.2 Web功能测试**

首先启动阿里云服务器，进入服务器实例，进行服务器的初始化以及启动，等待HTTP请求的到来，核心代码如下，测试结果的结果我们通过网页抓包来显示，图17为GET方法测试结果。

创建server实例，设置初始化参数，端口为1316，事件模式为ET模式，超时时间为60000毫秒，优雅退出设置为否，mysql端口3306，账号为root，密码为root，数据库为webserver，连接池数量为12，线程池线程数量为6，日志开关为开，日志等级为1，日志异步队列容量为1024，最后启动服务器等待客户端的连接。

WebServer server(

            1316, 3, 60000, false,

            3306, "root", "root", "webserver",

            12, 6, true, 1, 1024);

void WebServer::Start() {

    int timeMS = -1;

    if (!isClose\_) { LOG\_INFO("========== Server start =========="); }

    while (!isClose\_) {

        if (timeoutMS\_ > 0) { // 如果设置了超时时间

            timeMS = timer\_->GetNextTick(); // 获取下一个计时器超时时间

        }

        int eventCnt = epoller\_->Wait(timeMS); // 等待事件发生

        for (int i = 0; i < eventCnt; i++) { // 遍历处理每个事件

            int fd = epoller\_->GetEventFd(i);

            uint32\_t events = epoller\_->GetEvents(i);

            if (fd == listenFd\_) { // 事件是监听套接字实例

                DealListen\_(); // 处理监听事件

            } else if (events & (EPOLLRDHUP | EPOLLHUP | EPOLLERR)) {

                assert(users\_.count(fd) > 0);

                CloseConn\_(&users\_[fd]); // 关闭用户连接

            } else if (events & EPOLLIN) { // 可读事件

                assert(users\_.count(fd) > 0);

                DealRead\_(&users\_[fd]); // 处理读取事件

            } else if (events & EPOLLOUT) { // 可写事件

                assert(users\_.count(fd) > 0);

                DealWrite\_(&users\_[fd]); // 处理可写事件

            } else {

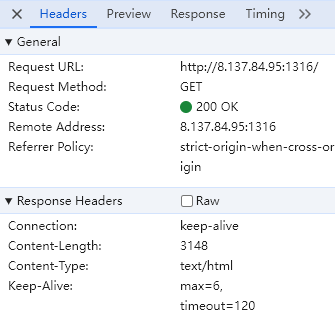
                LOG\_ERROR("Unexpected event"); // 错误日志

            }

        }

    }

}



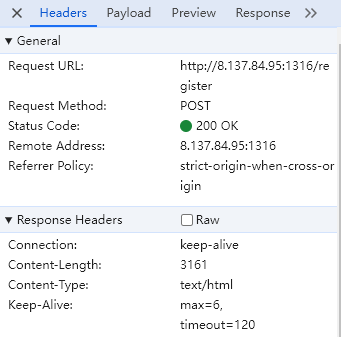
**图17 HTTP GET方法测试结果**

从图17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://8.137.84.95:1316/；响应报文中状态码为200表示正常工作，返回报文格式为text/html，长度为3148。可知服务器能正确响应HTTP GET请求报文。



**图18 通过浏览器访问服务器**

通过浏览器访问服务器，结果如图18所示，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。接在网页中点击注册按钮，注册一个账号保存到数据库中，测试POST方法，结果如图19所示。



**图19 HTTP POST 方法测试结果**

从图19中可以看到请求方法为 POST，URL为http://8.137.84.95:1316/register，状态码为200表示正常响应请求，返回报文格式为text/html，Content-Length表示响应报文中返回的数据长度为3161。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 服务器吞吐率测试**

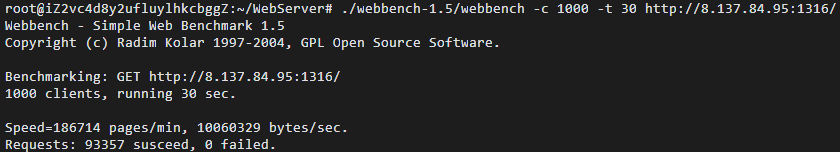
**5.3.1 吞吐率测试**

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。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。启动阿里云服务器并分配足够的内存，接着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://8.137.84.95:1316/

如图20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。



**图20 Webbench 性能测试**

**5.3.2 实验结果分析**

实验结果显示：经过webbenchh压力测试可以实现上万的QPS。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。

**5.4 本章小结**

本章对设计的Web服务器的系统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。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优异。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。

**6 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，对服务器的基本原理以及

网络应用设计基础进行了阐释，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Linux的WebServer服务器的设计与开发，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，吞吐率表现优异，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。

(3) 进一步测试优化

本文采用Linu平台的一些开源工具进行吞吐率等指标的测试，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]中国就业指导中心，电子商务师，北京：中央广播电视大学出版社，2002

[2]胡志详，范志伟，3D虚拟商场，091 CSIE-S020，1~2

[3]Web设计里的软件工程思想，<http://www.csdn.net>

[4]张友生，Web工程过程，计算机工程与应用，2003，1~8

[5]聂宏珉，ASP技术在电子商务中的应用，天津电大学报，2003年6月，第七卷（2）：1~3

[6]郑阿奇，SQLServer实用教程，北京：电子工业出版社，2003

[7]胡松涛，产品展示能手——Cult3D入门与提高，中国电力出版社，2003

[8]谭浩强，网页编程技术，北京：清华大学出版社，2002

[9]李晓黎，张巍，ASP+SQLServer网络应用系统开发与实例，北京：人民邮电出版社，2004

[10]黄心渊，3ds max5 标准教程，北京：人民邮电出版社，2003

[11]苏威洲，实现网络三维互动－Cult3D应用指南，北京：清华大学出版社，2003

[12]张海藩，软件工程导论（第四版），北京：清华大学出版社，2003

[13]萨世煊，王珊，数据库系统概论（第三版），北京：高等教育出版社，2002

[14]三维互动，科迪三维互动，<http://www.ekedi.com>

[15]吴明辉，ASP网络办公及商务应用系统开发实例导航，北京邮电出版社，2004

[16]JavaScript专区，月光工作室，http://www.moon-soft.com

致谢

首先，我要真诚的感谢指导老师朱朝霞老师。本设计在选题和完成都是在他的精心指导下完成的。同时我也深深的感谢这几年来所有代课老师和共处的同学给我的无私的帮助。

在实际完成论文的过程中，由于受个人的知识、经验和能力的限制，论文肯定存在不足之处，我恳请各位老师提出批评和指正。我会在以后的学习和工作中严格要求自己，努力提高自己的专业水平，以不辜负老师对我的期望。

丁斌斌

2024年5月

附录：