天津大学

《计算机网络》课程设计报告

Web Server 的设计与实现

学　　号_____

姓　　名_____

学　　院　智能与计算

专　　业　软件工程

年　　级　　大二

任课教师_____

2025 年 4 月 29 日

一、报告摘要

第一周主要内容是搭建编程环境与实现简单的 echo web server，在 Server 收到 client 的请求行信息后，能够真正确解析并返回响应。

第二周主要内容是能实现 HTTP/1.1 的基本功能，实现对 GET/HEAD/POST 请求的解析和返回，能够正确处理 400、404、500、501、505 等错误响应，并且要支持 keep-alive 连接。

第三周主要内容是实现 HTTP/1.1 的 Pipelining 管线化技术，实现对多个并发请求的顺序统一回复，减少网络传输开销。

第四周的内容已融入了前三周的实验中，全程带着比 select/poll 性能更强的 epoll 来做的高并发处理，类似于 redis 的单线程模型，用通知机制提供了应对高并发场景的方法。

此外，该报告除了第一周的实验内容，还有整个 WebServer 实验的任务需求分析以及总体设计。

二、任务需求分析

第一周学习 HTTP/1.1 协议的内容，利用所学知识设计 WebServer 实现协议对应的功能。通过对 HTTP 经典协议的复现，掌握应用层协议的设计和实现方法，加强学习能力与自主设计能力，同时对计算机网络有更深层次的理解。

第二周学习 HTTP/1.1 对 GET/HEAD/POST 请求和响应的定义和规范，利用所学设计服务端处理逻辑，正确的按照规范返回三种请求的响应，并能够支持持久连接以节省资源。

第三周学习 Pipelining 管线化技术，理解这个技术的实现与缺陷，正确处理并发的多个请求并按照顺序返回响应。

第四周学习 select/poll/epoll 的 Linux I/O 多路复用机制，理解他们之间的关系并设计出多路复用可应对高并发场景的 HTTP/1.1 服务器。

三、协议设计

3.1 总体设计

设计类似 Redis 单线程模型的系统，基于 Linux 的 epoll I/O 多路复用机制

1. 初始化服务器时创建 event_poll，并监听 Server 端的 Socket
2. 每当服务端 Socket 状态变化，接受客户端连接将其或放入 epoll 监听中，每当其状态变化处理请求/响应
3. 准确处理返回的响应与缓冲区，确保可靠性与处理速度

3.2 简单 echo web server 的设计

1.仿照 redis 的 epoll IO 多路复用事件驱动模型做了单线程的、可支持高并发的设计(相当于把 3.5 也做了)，使用 event_poll 标记事件为可读/可写，使用 LT 通知机制以确保可用

2.实现了实验要求的解析功能(见 4.1)，为之后的实验打下基础

3.3 基本 HEAD、GET、POST 方法的设计

1.对于合法的 HEAD 请求，通过 st 结构体获得文件的状态信息，直接返回响应头，包含必要的文件信息等

2.对于合法的 GET 请求，返回响应头后，如果返回的文件较大超过缓冲区大小，分批次写入；如果可以一次返回，则与响应头一起写入

3.对于 POST 请求，与第一周的实现一致，直接 echo back

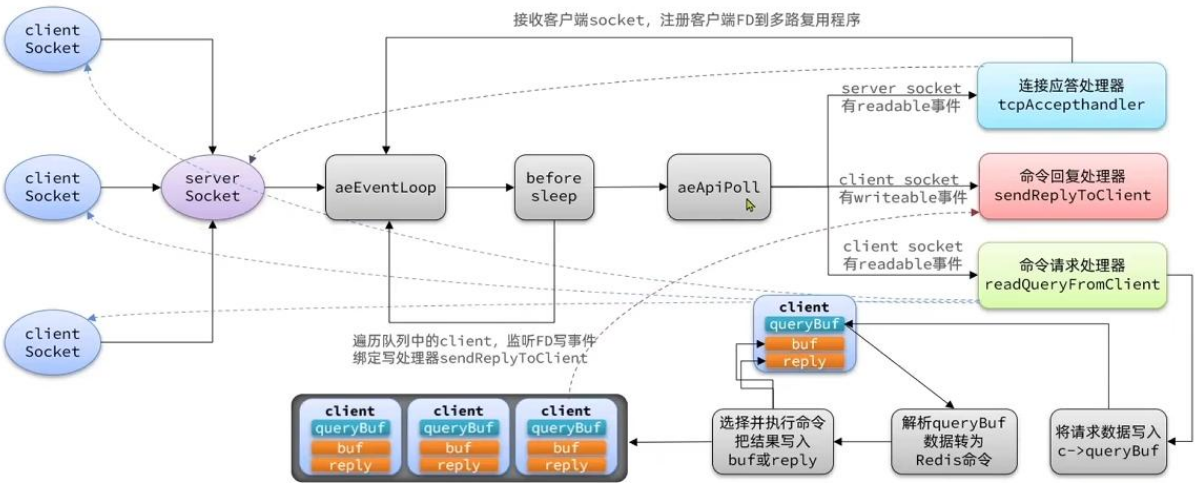
3.4 HTTP Pipelining 的设计

1.首先分析要实现功能的情境：一次向缓冲区中发送多个请求，然后往响应中按照顺序返回多个响应

2.设计实现：请求发送过来以后，先解析有几个\r\n\r\n 符号，了解有多少个请求，如果是单一请求，走之前的逻辑；如果是多个请求，使用新设计的多请求处理的逻辑

3.深入理解 Pipelining 技术，理解为什么现在没有多少使用，原因是会导致队头阻塞

3.5 多个客户端的并发处理的设计



如图是 redis 的单线程高并发 I/O 多路复用处理机制，这里简化了一些机制。

3.6 CGI 的设计（选做）

未做

四、协议实现

4.1 简单 echo web server 的实现

设计存储 HTTP 请求的结构体，解析 TCP 传输的内容并正确设置 HTTP 请求的状态和各项属性：

请求到达时，通过一个自定义的 parse 函数解析 HTTP 请求并设置其状态，然后通过自定义的 build_response 函数设置对应的响应，并在 socket 可写时返回响应。

4.2 基本 HEAD、GET、POST 方法的实现

HEAD/GET 方法：通过 struct stat 获取对应资源的属性，封装到 Content-Type 和 Content-Length 响应头中以进行正确响应(这里还封装了生成 HTTP 标准时间格式)，并根据请求头的 Connection 字段设置响应头的 Connection 字段和决定断开连接与否；对于 HEAD 方法，设置完响应头后直接返回。

POST 方法：直接用第一周实现的代码 echo back

4.3 HTTP Pipelining 的实现

通过在读取缓冲区时判断有多少请求，决定是否要使用管线化技术：若有多个请求，则按顺序把请求放入管线化处理的缓冲区中，并逐个处理请求，放入响应缓冲区中；在处理完请求后，将响应缓冲区中的内容写入 client->buf 中，发送给客户端完成响应。

4.4 多个客户端的并发处理

使用 epoll I/O 多路复用机制，监听客户端 socket 的可读、可写事件，避免 recv 等命令阻塞，进而提高并发量，增强可用性。在客户端可读时处理新连接或解析连接，设置响应；在客户端可写时写入响应并发送。

4.5 CGI 的实现（选做）

未做

五、实验结果及分析

5.1 简单 echo web server 的实验结果与分析

刚开始我是用 curl -v -X PUT http://localhost:9999 请求时客户端会直接卡住，是因为没有返回 Content-Length 属性，返回这个属性并设置为 0 时，就不会卡住了。
下面是一些执行的结果：
正常执行

```
root@8471daff2e46:/home/project-1# ./echo_server
Server running on port 9999 (single-threaded epoll), author:shr1mp
New client: 127.0.0.1:58908 (fd=5)
Client 127.0.0.1:58908 disconnected
```

2. Server

```
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 9999 (#0)
> GET / HTTP/1.1
> Host: localhost:9999
> User-Agent: curl/7.58.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain
< Content-Length: 78
<
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 78

cept: */*

* Connection #0 to host localhost left intact
root@8471daff2e46:/home/project-1#
```

3. Client1

Bad Request

```
root@8471daff2e46:/home/project-1# ./echo_server
Server running on port 9999 (single-threaded epoll), author:shr1mp
New client: 127.0.0.1:58908 (fd=5)
Client 127.0.0.1:58908 disconnected
New client: 127.0.0.1:41376 (fd=5)
Client 127.0.0.1:41376 disconnected
```

2. Server

```
root@8471daff2e46:/home/project-1# curl -v -X get http://localhost:9999
Note: Unnecessary use of -X or --request, GET is already inferred.
* Rebuilt URL to: http://localhost:9999/
*   Trying ::1...
* TCP_NODELAY set
* connect to ::1 port 9999 failed: Connection refused
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 9999 (#0)
> get / HTTP/1.1
> Host: localhost:9999
> User-Agent: curl/7.58.0
> Accept: */*
>
< HTTP/1.1 400 Bad Request
< Content-Length: 0
<
* Connection #0 to host localhost left intact
root@8471daff2e46:/home/project-1#
```

3. Client1

Not Implemented

```
root@8471daff2e46:/home/project-1# ./echo_server
Server running on port 9999 (single-threaded epoll), author:shr1mp
New client: 127.0.0.1:58908 (fd=5)
Client 127.0.0.1:58908 disconnected
New client: 127.0.0.1:41376 (fd=5)
Client 127.0.0.1:41376 disconnected
New client: 127.0.0.1:43268 (fd=5)
Client 127.0.0.1:43268 disconnected
```

2. Server

```
root@8471daff2e46:/home/project-1# curl -v -X PUT http://localhost:9999
* Rebuilt URL to: http://localhost:9999/
*   Trying ::1...
* TCP_NODELAY set
* connect to ::1 port 9999 failed: Connection refused
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 9999 (#0)
> PUT / HTTP/1.1
> Host: localhost:9999
> User-Agent: curl/7.58.0
> Accept: */*
>
< HTTP/1.1 501 Not Implemented
< Content-Length: 0
<
* Connection #0 to host localhost left intact
root@8471daff2e46:/home/project-1#
```

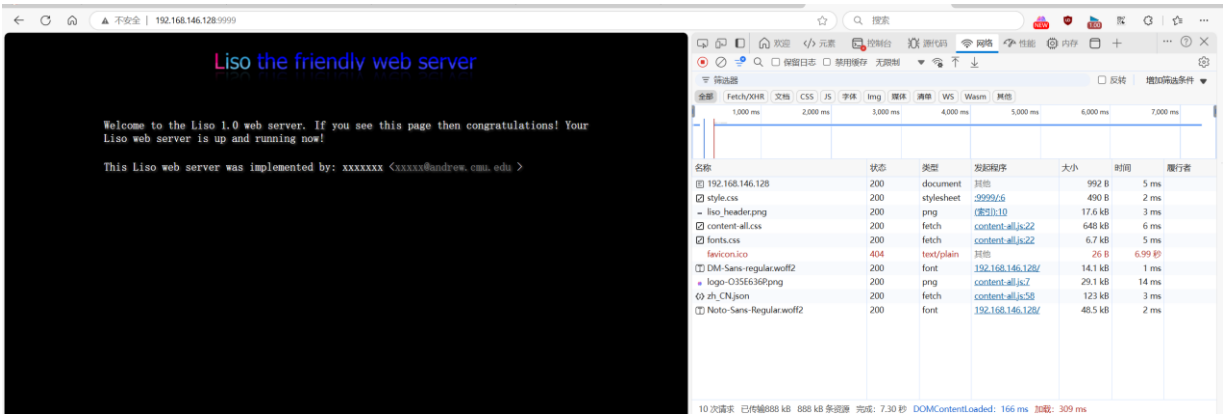
3. Client1

5.2 HEAD、GET、POST 方法的实验结果与分析

浏览器测试与服务端日志记录

```
Server running on port 9999 (single-threaded epoll), author:shr1mp
New client: 192.168.146.1:3305 (fd=5)
Generate the response to client 192.168.146.1:3305/(fd=5)
Generate the response to client 192.168.146.1:3305/style.css(fd=5)
Generate the response to client 192.168.146.1:3305/images/liso_header.png
(fd=5)
bytes_read = 4096
bytes_read = 4096
bytes_read = 4096
bytes_read = 4096
bytes_read = 1047
Complete ! file_offset - >17431 , file_size -> 17431
Generate the response to client 192.168.146.1:3305/favicon.ico(fd=5)
file not found
```

2. VM Ubuntu



可以看到，服务端正确解析了浏览器的请求。由于 HTTP/1.1 的请求默认是 keep-alive 的，从日志记录也可以看出重用连接成功，都在 fd=5 的 client socket 上执行读取、响应

下面对于 POST 请求和 HEAD 请求，使用 curl 命令进行测试：(curl 的 HEAD 请求会提示没有返回请求体)

HEAD 请求：

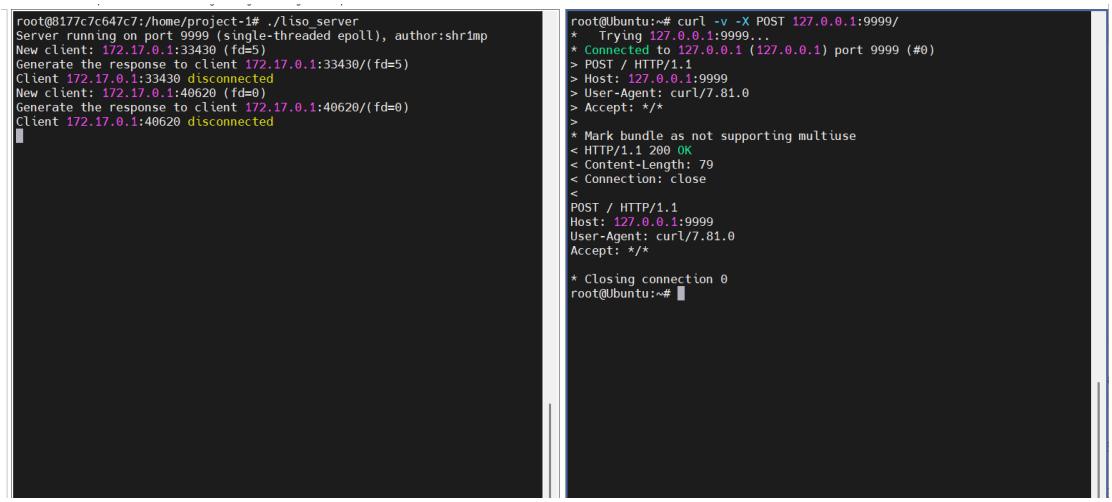
```
root@8177c7c647c7:/home/project-1# ./liso_server
Server running on port 9999 (single-threaded epoll), author:shr1mp
New client: 172.17.0.1:33430 (fd=5)
Generate the response to client 172.17.0.1:33430/(fd=5)
Client 172.17.0.1:33430 disconnected
```

2. VM Ubuntu

```
root@Ubuntu:~# curl -v -X HEAD 127.0.0.1:9999/
Warning: Setting custom HTTP method to HEAD with -X/--request may not wo
rk the
Warning: way you want. Consider using -I/--head instead.
*   Trying 127.0.0.1:9999...
* Connected to 127.0.0.1 (127.0.0.1) port 9999 (#0)
> HEAD / HTTP/1.1
> Host: 127.0.0.1:9999
> User-Agent: curl/7.81.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Server: liso/1.1
< Date: Fri, 18 Apr 2025 08:19:46 GMT
< Content-Type: text/html
< Content-Length: 802
< Last-Modified: Thu, 17 Apr 2025 16:56:38 GMT
< Connection: close
<
* transfer closed with 802 bytes remaining to read
* Closing connection 0
curl: (18) transfer closed with 802 bytes remaining to read
root@Ubuntu:~#
```

4. VM Ubuntu

POST 请求：(echo back)



下面再对一些异常情况进行分析：(使用 printf 结合 netcat 发送请求)

① 400 Bad request

```
root@Ubuntu:~# printf "HEAD /abc HTTP/1.1\r\n\r\n" | nc 127.0.0.1 9999
HTTP/1.1 400 Bad request

root@Ubuntu:~#
```

② 404 Not Found

```
root@Ubuntu:~# printf "HEAD /abc HTTP/1.1\r\n\r\n" | nc 127.0.0.1 9999
HTTP/1.1 404 Not Found

root@Ubuntu:~#
```

③ 501 Not Implemented

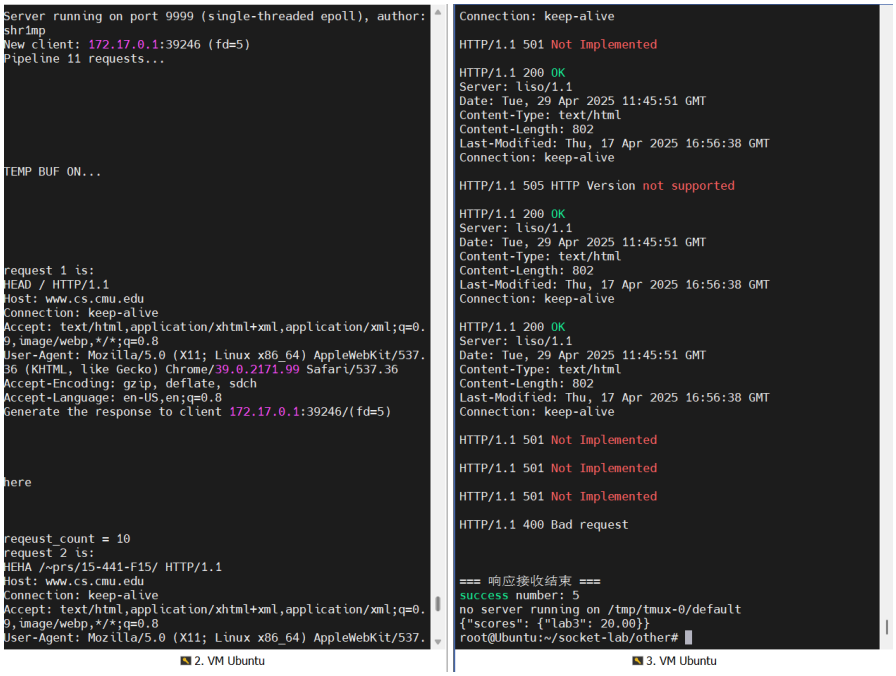
```
root@Ubuntu:~# printf "GETE / HTTP/1.1\r\n\r\n" | nc 127.0.0.1 9999
HTTP/1.1 501 Not Implemented

root@Ubuntu:~#
```

④ 505 Http Version not suppported

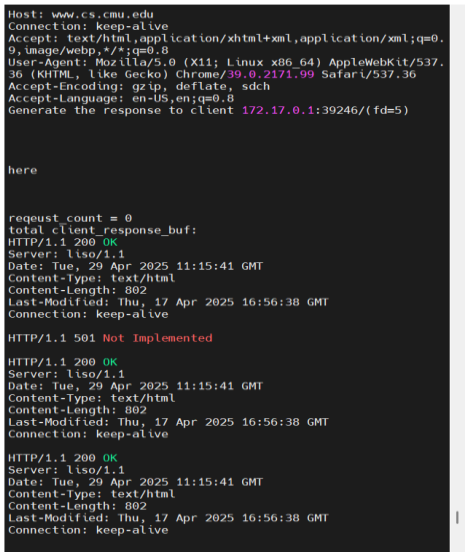
```
root@Ubuntu:~# printf "HEAD / HTTP/1.0\r\n\r\n" | nc 127.0.0.1 9999
HTTP/1.1 505 HTTP Version not supported

root@Ubuntu:~#
```

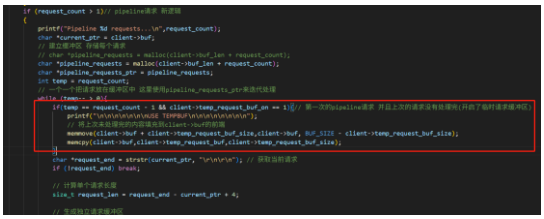
5.3 HTTP 的并发请求的实验结果与分析

这里使用 samples 中提供的 request_pipeline 中的请求，使用 python 脚本发起请求



(这里的分数由于请求不是测评平台的分数 所以忽略即可)



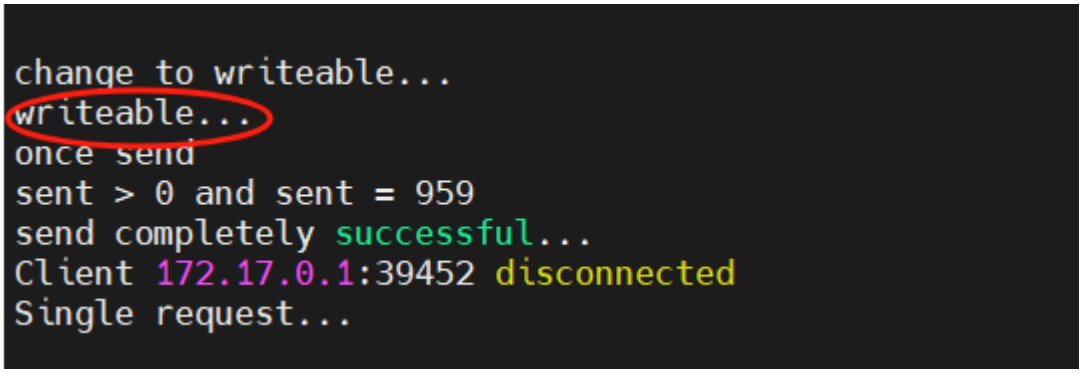


后端写了日志来记录每个请求对应的请求解析出来的样子和对应返回的结果，方便调试，最终达到了预期的效果

对于请求发不完，缓冲区不够用的情况，使用了另外一个缓冲区来存储不完整的请求，让 pipeline 可以处理两次缓冲区大小的并发请求，进而通过测试

5.4 多个客户端的并发处理的实验结果与分析

已融入了前三周的实验中，日志记录中会出现 readable/writeable 的字样以标记。结果是较高性能的 web 服务器，可以处理较多的并发请求。



5.5 CGI 的实验结果与分析（选做）

未做

六、总结

第一周的实验我巩固了我之前学习的 Linux 基于 epoll 的 IO 多路复用的实现，加深了对网络请求、处理和协议的概念、协议的设计的理解，通过创建支持并发的单线程模型，实现对 http 请求的解析，我学会了很多 C 语言、计算机网络相关的知识。

第二周的实验使我更加深刻理解了 epoll 的 LT 和 ET 模式以及通知机制，理解了 keep-alive 的底层实现与 HTTP 协议底层的基本通信原理，也加深了对 GET/HEAD/POST 三种方法的适用场景和定位的理解。

第三周的实现让我了解到了不怎么被广泛使用的 pipeline 管线化技术的原理并动手实现了 pipelining 处理，理解了为什么会队头阻塞，掌握了缓冲区处理与拼接的诀窍。

第四周的实验让我学习到了 select、poll、epoll 等机制的实现与差别，并在整个 web server 实现的过程中融入了 epoll I/O 多路复用机制以提高服务器性能与应对高并发场景的能力。

总的来说，通过这四周的实验，我系统性地掌握了 Linux 高并发网络编程的核心技术。从基础的 I/O 多路复用实现到 HTTP 协议深度解析，再到管线化优化和性能调优，不仅夯实了 epoll、select/poll 等底层机制的理解，更培养了解决队头阻塞、缓冲区管理等实际工程问题的能力。这些实践让我认识到，优秀的网络服务需要在协议规范、并发模型和系统调优之间取得精妙平衡，为后续开发高性能服务器打下坚实基础。