

天津大学

《计算机网络》课程设计报告



HTTP 的设计与实现

学 号: 3020202184 3020244344

姓 名: 刘锦帆 李镇州

学 院: 智能与计算学部

专 业: 计算机科学与技术

年 级: 2020 级

任课教师: 石高涛

2022 年 4 月 12 日

目 录

第一章	报告摘要	1
第二章	任务需求分析	2
2.1	第一周：实现简单的 echo web server	2
2.2	第二周：实现 HEAD、GET、POST 方法	2
2.3	第三周：实现 HTTP 并发请求	2
2.4	第四周：实现多个客户端的并发处理	3
2.5	选做：CGI	3
第三章	协议设计	4
3.1	总体设计	4
3.1.1	基础代码源文件架构分析	4
3.2	数据结构设计	4
3.3	协议规则设计	5
3.3.1	使用 socket 进行网络通信的流程	6
3.3.2	消息解析方法	7
3.3.3	动态数组设计	9
3.3.4	请求报文与响应报文协议设计	9
3.3.5	流水线设计	10
3.3.6	多客户端并发处理协议	11
3.3.7	CGI 协议	11

第四章 协议实现	12
4.1 第一周——实现简单的 echo web server	12
4.1.1 消息解析	12
4.1.2 消息反馈	12
4.2 第二周——实现 HEAD、GET、POST 方法	14
4.2.1 方法实现	14
4.2.2 妥善管理缓冲区	16
4.2.3 日志记录模块实现	17
4.2.4 读写磁盘文件错误处理	17
4.3 第三周——实现 HTTP 的并发请求	17
4.4 第四周——实现多个客户端的并发处理	19
4.5 选做——CGI	20
第五章 实验结果及分析	22
5.1 Auto Lab 测试合集	22
5.2 第一周——实现简单的 echo web server	22
5.2.1 对消息的正确解析	22
5.2.2 echo web server	22
5.3 第二周——实现 HEAD、GET、POST 方法	24
5.3.1 结果分析	24
5.3.2 日志	25
5.4 第三周——实现 HTTP 的并发请求	25
5.4.1 pipelining 测试	25
5.5 第四周——实现多个客户端的并发处理	26
5.5.1 手工测试	26

5.5.2	Apache 测试	26
5.6	选做——CGI	27
第六章	个人总结	29
附录		

一 报告摘要

二 任务需求分析

在本次实验中，我们完成了一个完整的 HTTP 服务器。能够处理并发的，大量的符合 **HTTP 1.1** 的请求报文，并且能够支持对 **CGI** 请求做特殊处理。以下将分别介绍四周、以及选做的任务需求及分析。

2.1 第一周：实现简单的 echo web server

第一关的任务主要分为以下几点：

1. 搭建编程环境、熟悉 Socket 编程方法，为之后的任务奠定基础；
2. 完成消息解析模块，为服务器实现基本的功能做好准备；
3. 完成基本的错误码返回以及基础方法 HEAD、GET、POST 的判断，为以后分别处理不同的报文，同时测试模块化、解耦和的编程框架；

总地来看，第一周任务的主要目的，是让我们熟悉编程环境以及 socket 的基础功能。万事开头难，完成第一关的任务，将对我们完成剩下的任务奠定一个良好、坚实的基础。

2.2 第二周：实现 HEAD、GET、POST 方法

第二关的任务主要为以下几点：

1. 完善服务器的功能、能够建立并维系 HEAD、GET、POST 的持久连接；（此时还不需要进行 Pipelining）
2. 支持 4 种 HTTP 1.1 错误码、能够正确封装响应消息；
3. 妥善管理缓冲区、避免客户端请求消息过长导致缓冲区溢出，该项任务主要目的在于处理 1. 下一步 Pipelining 请求过长，以及 2. 后续 CGI 处理 POST 请求时，可能会有大量参数传递，导致溢出的情况；
4. 处理读写磁盘错误，防止因为错误导致服务器宕机等问题；
5. 进行格式化日志的记录，方便正式上线后的 Debug；

第二周的任务是对于 server 处理框架的优化，引导我们去实现独立的消息处理和日志模块等，解耦和、模块化的编程框架为之后的编程提供了便利和良好的可扩展性。

2.3 第三周：实现 HTTP 并发请求

第三关的任务主要为：

1. 服务器能够连续相应客户端的 Pipelining 请求；
2. 即使出现了错误的请求，也妨碍服务器进行剩下并发请求的处理；

第三周的任务是对于 server 端的能力进行提升，作为一个比较单独的功能，

是根据 RFC2616 文档标准设计出来的并发请求模式。是我们实现标准 HTTP 服务器的关键之处。

2.4 第四周：实现多个客户端的并发处理

第四关的任务主要是：

1. 使用 select 函数实现多用户并发请求，在其他用户暂停发送时，服务器能察觉并转而给其他用户提供服务；
2. 服务器的最大连接数量设置为 1024，为 Linux 最大的文件描述符；

第四周的任务是 HTTP 1.1 中较为独立且较为困难的协议之一。在理论上来讲，我们只需要在第三周止步即可实现一个完整的服务端程序。然而第四周的 select 为多用户并发的情况进行了非常好的优化，是向市面上的服务端靠齐的一个关键。在这一关中，我们还将使用 apache bench 对服务器性能进行评测，这也是向正规服务器看齐的关键。

2.5 选做：CGI

1. 根据 RFC3875 以及 RFC2396 文档，实现 CGI 的请求；
2. CGI 请求的处理主要靠 URI 区别，且将用一个新的线程处理该请求；

该任务的完成，将标志着我们的服务器走向新的一个台阶：它将能够正确处理 POST 请求，且拥有作为后端程序与前端界面进行交互的能力。当然，对于 session 等更高级的网络编程工具的支持，尚未开发。至少通过我们的实现，我们将能够实现一个非常基础的用户注册、登陆接口。

由于我们将采用 Python - 数据库的技术栈，所以我们能够对用户信息进行记录，同时方便对以后想添加的其他功能进行扩展。

如图 2-1 我们在自己的阿里云服务器上部署项目，方便测试与演示。

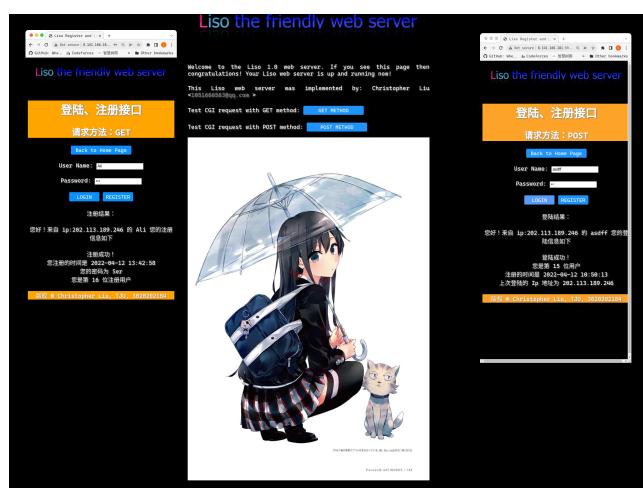


图 2-1 CGI 成品样例

三 协议设计

3.1 总体设计

3.1.1 基础代码源文件架构分析

首先我们通过 tree 命令获得如图

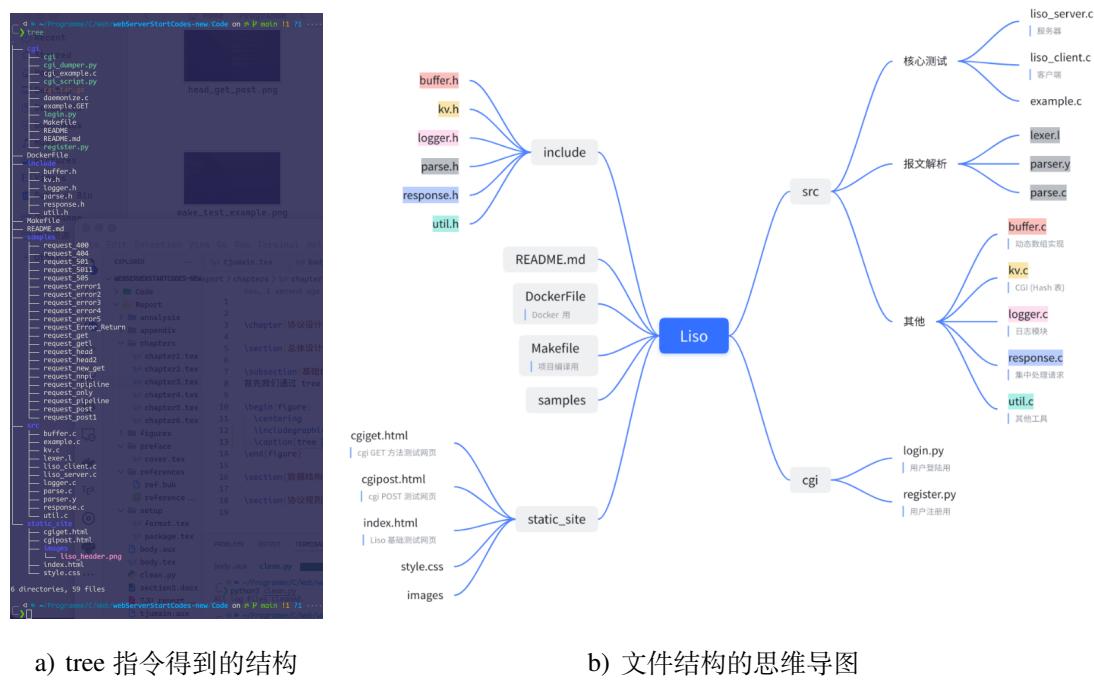


图 3-1 项目文件结构

可以看到，项目文件主要分为三大部分：静态网页、cgi 后端源码和服务器源码及其编译文件。三个部分是解耦和的，只需按照相应的接口进行测试，有相互依赖的问题。其中，我们增加了诸如 buffer, kv, logger, response 等文件，在表3-1 中详细给出其用途及其类型。

3.2 数据结构设计

为更加高效地编写出鲁棒的程序，我们主要设计并使用了三个数据结构：

- **parse.[c|h]**: 报文解析数据结构，用于解析报文；
 - **kv.[c|h]**: cgi 的参数及环境变量键值对；
 - **buffer.[c|h]**: 动态数组，用于高效地管理堆空间以及用户缓存；

图3-2 展示了三个数据结构的设计原型及结构体定义，方便浏览。

文件	用途描述	类型
1 buffer.[c h]	实现了 char* 数组增长、缩短、堆空间处理等功能的动态数组类型，用于处理缓冲区溢出问题	数据结构
2 kv.[c h]	实现了 CGI 环境变量、参数的数据结构，实现了键值对的添加以及堆空间的处理	数据结构
3 parse.[c h]	处理报文解析	数据结构
4 logger.[c h]	实现了简单的日志管理工具	工具
5 response.[c h]	处理报文以及生成回报文	函数集合
6 util.[c h]	其他小工具	工具宏
7 login.py	用户登陆的 python 程序	后端程序
8 register.py	用户注册的 python 程序	后端程序
9 cgiget.html	cgi METHOD 方法测试的网页	前端网页
10 cgipost.html	cgi POST 方法的测试网页	前端网页

表 3-1 部分文件及用途

图3.2a) 展示了一个存储报文的数据结构，包含 HTTP 的 Request 以及 Headers，一个 Request 包含方法、uri、版本以及 headers；一个 Headers 则包含一些键值对，描述传输的信息。同时，定义了处理报文的函数 parse() 原型。

图3.2b) 展示了一个存储 cgi 参数 (argc)、环境变量 (ENVP) 的数据结构；argc 就是简单的字符串数组，ENVP 是描述环境变量的键值对。同时，定义了添加参数、环境变量以及清除堆空间的函数。

图3.2c) 展示了一个动态数组的原型定义，包括其存储核心 buf，堆空间分配的大小 (capacity)，当前使用大小 (current)，以及方便后续多用户 cgi 处理时回退的一个标记“当前报文终止处”的标记 (access_end)；同时，定义了丰富完备的函数，用于妥善管理用户缓存区。

3.3 协议规则设计

以下将通过 socket 通信协议、消息解析协议、请求响应报文协议、流水线协议、多客户端并发请求协议以及 CGI 请求协议设计六个板块分别阐述 Liso server 运行的协议规则设计。

```

a) parse.h
b) kv.h
c) buffer.h

```

图 3-2 三个数据结构的原型

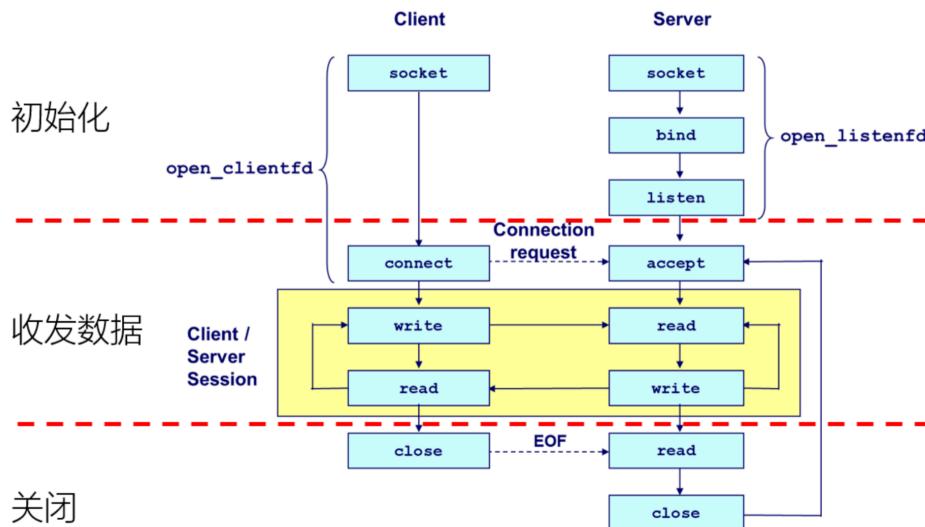


图 3-3 socket 编程流程图

3.3.1 使用 socket 进行网络通信的流程

在本次实验中，我们通过使用 socket 套接字进行网络编程。socket 的功能，是提供一个统一的 API 使得应用程序在统一的框架下实现数据传输、数据接收等功能，使得编程人员能够更加专注于数据的处理以及算法，而非数据传输、接收这些模式统一的问题。

首先，我们对 socket 通信的流程进行说明。如图3-3所示，使用 socket 套接字收发报文的流程主要分为三大部分，分为是：初始化、收发数据以及关闭阶段

初始化阶段：客户端通过 socket 方法创建自己的网络位置（即 Ip 和端口号）；服务器创建 socket 之后还需要 bind 和进入 listen 状态，即监听有没有客户端发出与之互动的请求。

收发阶段：客户端通过 `connect` 方法与远端主机的应用进程建立 TCP 连接，系统记录一个 `socket` 存储二者的 ip 及端口号；在该阶段，服务器先通过 `accept` 函数获取一个连接请求，并为其分配一个服务端的文件描述符，作为 `socket` 表的索引。连接建立完成后，客户端和服务器就进入了“读、写”和“写、读”的循环当中。值的注意的是：在该过程中，客户端和服务端的“读、写”一定要是不对等的，即一方先读、一方先写，否则会出现竞争条件，产生死锁。在具体实现方面，客户端通过 `socket` 套接字将数据发送给服务端缓冲区内；服务端则通过解析模块对客户端的报文进行解析，并根据其请求方法，生成响应报文。最后，服务端将响应报文通过 `socket` 套接字放入缓冲区、发送给客户端。客户端再通过 `socket` 套接字得到服务端的响应报文。至此，一次完整的通信结束，如此循环往复，直到一方发出包含“`connection: close`”的请求，`socket` 最终关闭。

关闭阶段：当一方提出关闭时（如果支持了持久化连接，则通过“`connection: close`”的头信息；否则默认只执行一次连接后就关闭 `socket`），结束通信，不得再发出任何报文，`socket` 被关闭。

3.3.2 消息解析方法

Yacc 和 Lex 简介 消息解析模块主要通过 `lex` 和 `yacc` 来实现。如图3-4 所示，`Lex` 主要负责对字符串的模式进行识别，将其拆分成 `token` 并赋值给固定模式的变量（如图中的 `id1`, `id2`, `id3` 和 `id4`）；`Yacc` 则对其中的关系进行运算，也就是在执行它的语法，如图中将固定模式的变量再次拆分成一颗运算树的结构。

实际应用 在我们的程序中，`lexer.l` 定义了词法规则，例如 `09` 就是 `digit` 等等的 `token`。而 `parser.y` 则定义了解析这些 `token` 的语法。例如以下的程序描述了一个关于请求头的语法定义，即一个请求开头，由 `{"token", "空格", "一段文字", "空格", "另一段文字", "换行符"}` 组成，且匹配到请求头时，要运行进行以下四行代码。

parser.y Example

```
1 request_line: token t_sp text t_sp text t_crlf {  
2     YPRINTF("request_Line:\n%s\n%s\n%s\n", $1, $3, $5);  
3     strcpy(parsing_request->http_method, $1);  
4     strcpy(parsing_request->http_uri, $3);  
5     strcpy(parsing_request->http_version, $5);  
6 };
```

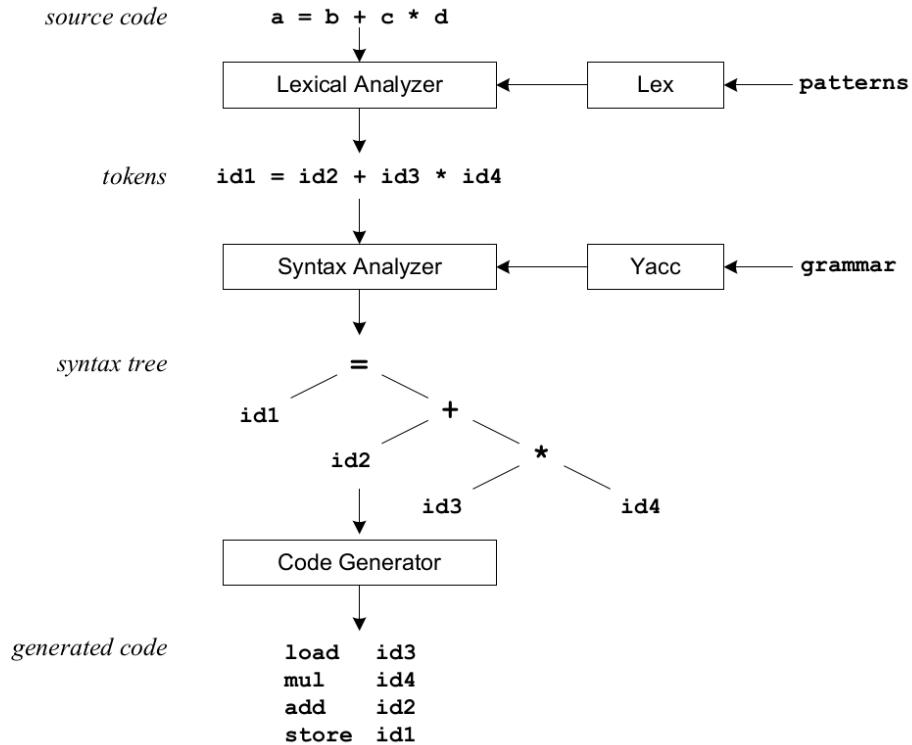


图 3-4 yacc 和 lex 协同作用原理

值的注意的是，yacc 支持一种递归定义的语法，让我们完成对多请求头语法的递归定义。如下代码即是示例。可以看到，111 行定义了基本的 request_header，13 行则循环定义了多条 request_header 的语法。

parser.y Example of Recursive Definition

```

1 request_header: token ows t_colon ows text ows t_crlf {
2     YPRINTF("request_Header:\n%s\n%s\n", $1, $5);
3     strcpy(parsing_request->headers[parsing_request->header_count]
4         .header_name, $1);
5     strcpy(parsing_request->headers[parsing_request->header_count]
6         .header_value, $5);
7     parsing_request->headers = (Request_header *) realloc(
8         parsing_request->headers,
9         (1+(++parsing_request->header_count)) * sizeof(Request_header)
10    );
11 }
12
13 request_header: request_header request_header ;

```

图 3-5 GET 请求测试

至此，我们应当能够完成报文的解析，获得了相应的 header 键值对了。

3.3.3 动态数组设计

为了更方便、严谨地管理缓冲区以及返回信息等操作，我们编写了 buffer 工具。通过对堆内存的申请、操作、释放来控制流、字符串以及报文结构整理等操作。工具操作的函数在数据结构处已经描述过，主要是通过 malloc 和 realloc 来实现。

3.3.4 请求报文与响应报文协议设计

解析报文，只是我们服务端基础功能，接下来将以 GET 方法为例子，介绍基本的请求报文、响应报文的协议设计。在 client 端有如图3-5 的运行效果。

第一行是的状态，"HTTP/1.1 200 OK"，代表服务器的版本为 HTTP 1.1，状态码 200，状态信息 OK。对于服务器而言，只需在能够正确返回时将这些信息按顺序填入缓冲区即可。然后我们可以看到之后有 6 行键值对信息，即 {"Date", "Server", "Last-Modified", "Content-Length", "Content-Type", "Connection"}，代表了 {" 该请求的日期", " 服务器名称", " 最后编辑时间", "Body 部分的长度", "Body 部分数据的类型", " 连接状态"}。且通过 Postman 的测试，我们发现有些键值对并不是必须的，且顺序并不影响客户端的到正确的响应信息。当然，作为例子，我们将详细讲解这六个部分是如何得到并加入到响应报文中的。

- **Date:** 通过 C 函数 time(), localtime() 以及 strftime() 将当前时间转化成一个字符串，然后通过统一函数调用 set_header() 函数添加到响应报文的

headers 部分：

- **Server:** 服务器信息为完全一致的，不会因为请求的不同而有区别，故我们可以存放在宏中，使用时直接调用就可以了；
- **Last-Modified 等文件相关:** 标志着请求文件最近修改的时间，可以通过 stat() 函数得到；
- **Connection:** 一般来说不需要在意这个选项，只需设置为 keep-alive 就行，但我们考虑到做一些整体服务性能的优化，遇到错误的请求，就关闭这个连接，能够增加整体受到正确请求的概率，于是在受到错误请求时，将该选项设置为 close 否则就设置为缺省；

可以看到，在 Body 部分，我们加入了整个 HTML 文件，这是因为 GET 请求规则所致。所以如果是 HEAD 请求，我们就不会增加这个 Body 部分的内容。至于文件是如何加入到 Body 的，简单通过 mmap() 进行一个内存的映射即可，这个方法比 read() 更好的点在于不需要进行图片的特判而使用 fread() 等， mmap() 函数是通用的。唯一需要注意的是 mmap() 函数使用后还需要使用 munmap() 函数进行内存的释放，否则会出现 segment fault。

3.3.5 流水线设计

实现流水线作业的核心，在于从 Pipelining 的信息中拆分出单个报文。从 RFC2616 文档中得知其规则后，我们发现可以使用"\r\n\r\n" 分割单个报文。所以我们通过 strstr() 函数，查出一个"\r\n\r\n" 的位置，即是一个报文结尾处（严格来说还需要加上这个"\r\n\r\n" 才算结束）。

解析出一个报文后，我们通过自己实现的动态数组工具创建一个新的动态数组 newbuffer，然后通过函数 append_dynamic_buffer()，将报文加入到 newbuffer 中，然后通过设置 access_end 记录当前报文在整个 pipelining 的位置。为什么要设计这个 access_end 呢，因为在之后 cgi 中，可能会在 body 部分传输参数，然而这个 body 部分的内容会被"\r\n\r\n" 拆分到下一条报文的开头，并不会添加到 newbuffer 中，也就是说我们只有在处理完一条报文之后才能确定该条报文的真实长度。为了统一性，我们才定义了这个 access_end 结构体变量表示当前报文的结束位置。在完成该报文的所有工作之后，我们需要释放新创建的动态数组，并更新用来存放 pipelining 报文流的动态数组：通过函数 update_dynamic_buffer() 将 access_end 前面的部分全部去掉。

这里需要注意的是，如果 body 部分被接收完全就开始对该报文进行处理时，我们需要正确返回，并回退到拆分报文之前的位置，待下一次接收后再处理该报文。

3.3.6 多客户端并发处理协议

对于多客户端并发访问，我们采用 select() 函数以及配套的 fd_set 进行处理。核心思想是：通过 select() 函数监控有没有发出请求的客户端，然后通过他们的文件描述符判断是来发出 connection 请求还是收发数据，最后通过其接收的数据是否为 0 判断其是否离开。流程图如图3-6 所示。

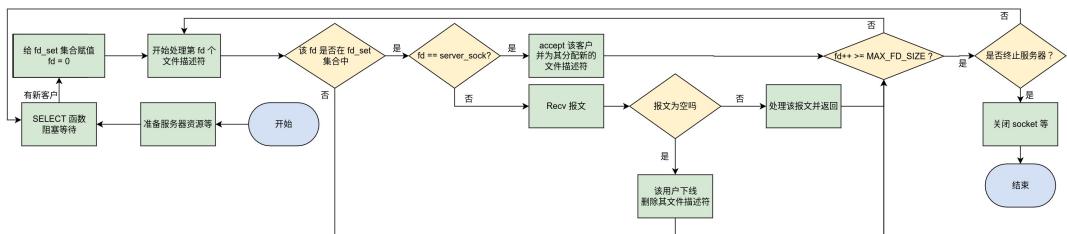


图 3-6 多客户端并发流程图

应该指出，我们通过 select() 函数，会得到一个用户池 fd_set，之后只需要遍历该池的所有位置即可。需要注意的是，我们需要为每个用户预先开辟一个空的缓冲区，即 1024 个 dynamic_buffer，来存放各自的数据。同时还需要维护每个用户的地址信息等。

3.3.7 CGI 协议

CGI 全称为 Common Gateway Interface，是实现服务器功能扩展的重要端口。其原理是：通过解析用户请求的 uri 是否含有 cgi 关键字，判断是否为 cgi 请求；如果是，则调用相应的 cgi 程序为其服务，否则按照正常情况处理。

根据实验指导书的意见，我们实现了如图的用户注册、登陆接口。在后端程序的编写中，我们采用了有良好跨平台性的 Python 以及 MySQL 数据库对用户信息进行存储，以便日后扩展时可以有良好的扩展接口。

CGI 实现的一般流程如下：

1. 解析并处理该报文
2. 判断 URI 中是否含有 cgi 的关键字（本次定义 /cgi/ 为关键字）；
3. 判断方法类型（仅支持 GET 和 POST）；
4. 判断请求参数是否完整（对于 POST 而言，可能尚未传输完成）；
5. 判断请求执行的文件是否存在、是否有权限执行；
6. 执行请求执行的 CGI 脚本（或程序），并将输出数据贴在返回报文的 Body 部分；
7. 按照正常的报文进行返回。

四 协议实现

以下将分别介绍本次实验的五个部分的具体实现方式。

4.1 第一周——实现简单的 echo web server

第一周的协议实现主要分为“消息解析”和“消息反馈”两个部分。

4.1.1 消息解析

```
# ./example ./samples/request_head
> GET / HTTP/1.1
> Host: localhost:8080
> Connection: keep-alive
> Accept: application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
> User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/537.36
> Accept-Encoding: gzip, deflate
> Accept-Language: en-US,en;q=0.8

[1] 1644 segmentation fault (Core dumped) ./example ./samples/request_head
d = ./samples/request_headServerStartCodes-new on 0x 0xa0 at 0:11:55.01
```

图 4-1 Segment Fault

如图4-1，通过 example.c 的测试，我们发现了第一个错误：Segment fault，且该错误总是在解析第三行消息时出现。

通过对 parser.y 和 parse.h 文件的理解，我们认为此问题应该出自其 Request 中 Request_header，它在最初定义时是一个指针 *header，所以在解析第二行时没有错误，而在进行第三行即以后的解释时，如果不进行人为的空间分配，则当然会出现上述的 Segment Fault。

所以对于这个问题，我们只需在 `parser.y` 中进行修改，增加 `realloc` 的函数，对 `header` 进行扩容，并且相应的 `grammar` 与之对应即可。

当然，还可以在 example.c 中处理一下返回错误的情况，避免直接访问 NULL 的空间导致 Segment Fault。使用批量脚本测试则效果如图4-2。

4.1.2 消息反馈

在完成消息解析后，我们对 echo_server.c 进行编程，完成消息的解析、处理与返回。研读源代码，梳理出如"echo server" 代码段的伪代码。

我们只需要在第 7 行“TODO: DEAL WITH MESSAGE”处增加如代码段“处理数据”的“解析、分类、封装返回值”即可。

- 具体而言，先通过 example.c一样的方法解析buf中收到的数据。在该步骤中，可能会出现依赖问题，对此我们需要在Makefile中增加相关的依

图 4-2 Example Test

赖。让 echo_server 在连接时加上 parse.o 的文件即可。

- 得到 request 后，我们对 request 的两种情况进行分析
 - 如果 request 为 NULL，则解析不成功，即返回 400 错误
 - 如果 reqeust 成功，但是发现其中 method 暂不支持，则返回 500 错误
 - 否则直接返回受到的数据

echo_server

```
1 create socket
2 bind socket
3 listen on socket
4 while(1):
5     accept connection
6     while(recieve message):
7         TODO: DEAL WITH MESSAGE
8         send back
9     close client socket
10    close socket
```

dealing with Data

```
1 def deal_with_request(buf):  
2     Request *request = parse(buf, BUF_SIZE, 8192)  
3     if request is NULL:  
4         return _400msg  
5     if method not support:
```

```

6     return _500msg
7     return buf

```

4.2 第二周——实现 HEAD、GET、POST 方法

第二周的协议实现主要分为“基本方法实现”、“缓冲区处理”、“日志记录模块”、“读写文件”这四个部分。

4.2.1 方法实现

服务器处理请求的方法被集中封装到文件 response.c 和 response.h 中。函数原型如图4.3a) 所示。

具体而言，liso_server 首先使用 while 来接收信息，存储到 buffer 中（此处采用自定义的动态缓存，在下一节详细介绍）。然后通过一个 strstr() 函数获取确定一个完整的请求后，将它加载到动态缓存dbuf 中，并调用函数 handle_request 处理，返回的信息将继续通过dbuf 带出。由于测试集的要求，此处不需要实现 Persistent Connection 中对连接情况的判断。下面将分别介绍错误码和基本方法的实现。

4.2.1.1 错误码

400 400 错误是解析错误，由于之前在 Lab 1 已经实现了较为完备的 parse() 函数，于是在 Lab 2 中将不再 parse() 函数之内做修改。首先，通过图4.3a)左侧第 37 行调用 parse() 函数对请求进行解析，并通过第 40 行的判断确定一个 400 的错误，并随即调用函数 handle_400 处理之。然后，通过对自定义函数的调用，对动态缓冲数组dbuf 进行清空、加载返回信息等操作。

404 404 错误是指无法找到请求的文件，目前只会在 HEAD 和 GET 方法处理时出现。在处理 GET 和 HEAD 请求时，我们会通过 stat 等函数对要请求的文件进行预处理，此时即可判断该文件是否存在、是否可读等权限问题，然后决定是否返回 404 错误。

505 505 错误是指协议版本不支持，通过图4.3a) 左侧第 53 行判断 my_http_version 与请求的 version 是否相同即可。

501 501 错误是指请求方法不支持，也可以通过一个 switch 判断出来。此处的 method_switch() 函数，将一个表示方法的字符串 ("GET", "POST", etc) 转化为图4.3a) 右侧第 35 行定义的枚举类 METHOD。

4.2.1.2 基本方法

HEAD HEAD 方法的具体实现由函数 handle_head() 实现。首先，通过函数 strcmp() 判断请求路径是否为默认路径，如果是，则直接请求文件 index.html，否则请求路径添加到子文件夹"./static_site" 之下。然后，通过函数 stat() 判断是否为 404 错误。如果一切顺利，则通过在下一节介绍的动态缓冲区处理函数按照前文提到的格式，向返回值 dbuf 中添加一系列 response 和 headers。具体如图4.4b) 所示。

```

// Copyright (C) 2022 Liu Shilun. All rights reserved.
// 文件名: response.c
// 作者: Liu Shilun
// 邮箱: 132323123@163.com
// 版本: 2022年3月1日
// 用途: 处理HTTP请求
// 说明: 该文件实现了HTTP请求的处理逻辑，包括解析请求头、读取请求体、根据请求方法调用相应的处理函数等。
//       其中，handle_head() 函数负责处理 HEAD 请求，将请求路径添加到 static_site 目录下，并根据响应状态码生成响应报文。
//       其他方法如 handle_get()、handle_post() 等则根据请求方法调用不同的处理逻辑。
//       代码注释详细说明了每一步骤的功能和参数含义。
//       请根据需求进行修改和扩展。
//       注意：本项目仅作为学习参考，不建议直接使用于生产环境。
//       作者：Liu Shilun
//       邮箱：132323123@163.com
//       版本：2022年3月1日
//       用途：处理HTTP请求
//       说明：该文件实现了HTTP请求的处理逻辑，包括解析请求头、读取请求体、根据请求方法调用相应的处理函数等。
//             其中，handle_head() 函数负责处理 HEAD 请求，将请求路径添加到 static_site 目录下，并根据响应状态码生成响应报文。
//             其他方法如 handle_get()、handle_post() 等则根据请求方法调用不同的处理逻辑。
//             代码注释详细说明了每一步骤的功能和参数含义。
//             请根据需求进行修改和扩展。
//             注意：本项目仅作为学习参考，不建议直接使用于生产环境。

```

a) Response.c 和 Response.h

```

// Copyright (C) 2022 Liu Shilun. All rights reserved.
// 文件名: liso_server.c
// 作者: Liu Shilun
// 邮箱: 132323123@163.com
// 版本: 2022年3月1日
// 用途: 实现一个简单的HTTP服务器
// 说明: 该文件实现了服务器端的主要逻辑，包括监听端口、接受连接、读取请求、处理请求、写回响应等。
//       使用了非阻塞IO模型（epoll）来管理多个连接。
//       代码注释清晰，易于理解。
//       作者：Liu Shilun
//       邮箱：132323123@163.com
//       版本：2022年3月1日
//       用途：实现一个简单的HTTP服务器
//       说明：该文件实现了服务器端的主要逻辑，包括监听端口、接受连接、读取请求、处理请求、写回响应等。
//             使用了非阻塞IO模型（epoll）来管理多个连接。
//             代码注释清晰，易于理解。

```

b) liso_server.c 和 liso_client.c

图 4-3 核心代码

GET GET 方法在 HEAD 方法的基础之上，添加了读取文件的操作，根据模块化变成的思想，我们将其封装到函数 `get_file_content()` 之中。为了实现对缓冲区的管理、避免 Segment Fault，我们仍先通过 `stat` 函数对文件情况做初步确定，并先对动态数组进行扩容，再读入文件。文件读入部分将在后续小节进行介绍。

POST POST 方法的实现，根据实验要求，直接返回接收到的报文，于是在判断没有 400, 505，以及 501 错误之后，我们即可将携带请求信息的动态数组 `dbuf` 原路返回。

4.2.2 妥善管理缓冲区

由于 socket 编程本身的性质，在 `send` 函数中会主动的将信息拆分成合适大小再分别发送。所以我们只需要针对 **1.** 发送前，用户的缓冲区；**2.** 接收后集中存放的用户缓冲区；两个缓冲区进行管理。

4.2.2.1 动态数组定义与实现

我们的策略，是通过自定义的动态数组实现对一个动态长度的缓冲区的支持。具体定义及部分实现如图4.4a)所示。

部分结构体、函数介绍如下：

struct dynamic_buffer 该结构体定义了一个基地址 `*buf`，以及它的总大小 `capacity` 和当前大小 `current`。

init_dynamic_buffer() 该函数使用系统调用 `malloc` 将为一个新定义的动态数组分配由宏"DEFAULT_CAPACITY" 定义的空间。即对动态数组进行初始化。

append_dynamic_buffer() 该函数将一个字符串拼接到动态数组后面。首先判断当前空间是否够用，如果不夠，则调用函数 `realloc` 为其增添空间，然后通过 `memcpy()` 函数，将新添加的字符串拼接到动态数组后面。

4.2.2.2 动态数组使用

动态数组的作用是解决缓冲区溢出问题，前文提到的两个可能出现溢出的地点，分别是读入文件时由于文件过大（或将其加入返回信息时）而溢出，另一个是在接收端虽然一次性接收的最大值由 `BUFSIZE` 控制，但是用户发送的请求大小不可定（例如 Lab3 中的 pipeline），导致缓冲区溢出。最终为了整体的和谐性，我们在大多数使用字符串数组的地方，都改用我们的动态数组。例如：

读入文件时 首先使用 `stat()` 获取文件大小，然后调用函数 `add_dynamic_buffer()` 函数对我们的动态数组进行可用空间的检验。最后再通过 `read()` 或者 `mmap()` 将文件写入动态数组。

Server 端 在 server 端，我们使用动态数组，存储即将返回给 client 的信息。由于 GET 可能会请求大小超标的文件，于是我们通过有保护机制的动态数组，安全地对它进行读取与返回。

接收 socket 信息时 由前文提到的 `recv()` 函数的性质，我们知道应该通过 `while` 来不断读取管道里的信息，如图4.3b) 右侧第 171 行。通过 `append_dynamic_buffer()` 函数，将读取的信息添加到当前动态数组 `dbuf` 中。每次处理缓冲区，即是对 `dbuf` 中的数据进行拆分（确定一份请求报文）、解析与返回，最后调用 `update_dynamic_buffer()` 函数，将该报文从动态数组中丢弃。最大限度地节约空间，并解决缓冲区溢出问题。

验证

如图4.5a)，即使将缓冲区大小设置为 1，我们的 server 依旧正常处理并返回信息（测试样例为 pipeline，由于和 Persistent Connection 相关联，故一起实现了）。

4.2.3 日志记录模块实现

日志记录模块借用了 Apache 标准，使用 C 宏定义完成第一层封装，采用 `logger.c` 与 `logger.h` 配合，进行向特定文件的输入，实现了 log 的可跟踪性。具体宏定义如图4.4c) 所示。

4.2.4 读写磁盘文件错误处理

只有 GET 和 HEAD 请求涉及对文件的处理。根据之前的描述，我们首先使用 `stat()` 函数对磁盘文件进行试探性的访问。如果在此时出错，可能有 1. 路径不存在；2. 没有查看权限。在服务器看来，我们只需返回 404 错误即可。

在 GET 的处理中，我们还需对文件进行读取，此时可能又会产生 3. 无法打开的错误；4. 文件为空；以及 5. 无法正确映射到内存；等问题（由于我们使用了上一节提到的动态数组，不会存在溢出问题）。

考虑完上述问题，我们实现了如图4.5b) 的代码。

4.3 第三周——实现 HTTP 的并发请求

第三周的实现较为简单。由于在 Lab2 中我们实现了缓冲区的自动化更新、防溢出。可以支持任意大小的文件传入。所以本周的任务中，我们只需要关注如

a) dynamic_buffer.c 和 dynamic_buffer.h

b) handle_head 函数

c) 日志文件

图 4-4 动态数组、日志文件以及 Handle Head 函数

a) buffer_test, BUF_SIZE=1

b) get_file_content 函数

图 4-5 测试 buffer 以及 get_file_content 函数

何将收到的信息拆分为单个单个的报文。然后处理之即可。

我们的实现方式，是通过函数 strstr() 获得拆分每个报文。strstr(str,dest) 函数得到 dest 在 str 第一次出现的位置。于是就能通过 strstr() 得到报文结束前" \r\n\r\n " 的位置。然后通过简单的字符串处理，就能开始处理单个报文了。部分代码如图4-6。

```

54 /**
55  * @brief Deal with buffer --> which contains (multiple) requests
56  * if it's a pipeline request
57  */
58 #param dbuf ... dynamic_buffer, requests
59 #param readret ... size of buffer
60 #param client_sock ... socket descriptor
61 #param sock ... server's sock
62 #param clt_addr ... client's address
63 */
64 int deal_buf(dynamic_buffer *dbuf, size_t readret, int client_sock, int sock, struct sockaddr_in clt_addr)
65 {
66     /* --> PERSISTENT : Go on waiting for msg
67     * --> EXPIRE_FAILURE : End this connection
68     * --> CLOSE : Close current connection (Which can be ignored in our lab)
69     */
70     if (dbuf->temp->buf->buf == NULL)
71     {
72         /* If we're in pipeline */
73         dynamic_buffer *return_buffer = (dynamic_buffer *) malloc(sizeof(dynamic_buffer));
74         init_dynamic_buffer(return_buffer);
75         while (client_sock >= 0 && client_sock != clt_addr.sin_port)
76         {
77             int len = t - temp;
78             if (len > 0)
79             {
80                 /* Append to return buffer */
81                 dynamic_buffer *each = (dynamic_buffer *) malloc(sizeof(dynamic_buffer));
82                 init_dynamic_buffer(each);
83                 append_dynamic_buffer(each, temp, len);
84                 append_dynamic_buffer(each, dest, strlen(dest));
85                 temp = t + strlen(dest);
86             }
87             #ifdef DEBUG
88             PRNT("*****CURRENT CNT: %d*****\n", cnt++);
89             #endif
90         }
91         free_dynamic_buffer(return_buffer);
92         update_dynamic_buffer(dbuf, temp);
93     }
94     #ifdef DEBUG
95     LOG("msg to be sent: %s\n", buf->buf);
96     #endif
97     if (return_buffer->current)
98     {
99         #ifdef DEBUG
100         LOG("Not complete\n");
101         free_dynamic_buffer(return_buffer);
102     }
103     else
104     {
105         if (send(client_sock, return_buffer->buf, return_buffer->current, 0) != return_buffer->current)
106         {
107             close_socket(client_sock);
108             close_socket(sock);
109             free_dynamic_buffer(return_buffer);
110         }
111     }
112 }

```

图 4-6 Liso Server Pipelining

4.4 第四周——实现多个客户端的并发处理

第四周修改的主要是 liso_server 的接收请求部分的代码。在 while(1) 的服务器处理循环中添加以 select 开头的预处理等操作，使服务器能够在等待一个客户端发送下一个请求时，同时处理来自其他客户端的请求，使服务器能够同时处理多个并发的客户端。

流程图如图3-6所示。首先使用 `cnt=select(MAX_FD_SIZE+1, &tmp_fds, NULL, NULL, NULL)` 获取当前总共有多少个客户端有请求，同时将他们的文件描述符添加到类型为 fd_set 的集合中，将用户总数返回给 cnt 变量。

然后循环遍历 fd_set 集合中 0-MAX_FD_SIZE 个位置。通过函数 FD_ISSET() 判断当前位置是否有客户。如果有客户，则进行处理，否则跳过。

在处理时，先判断其文件描述符是否和服务器的 sock 相同，如果相同则表明是一个新的连接，需要进行 accept 操作，并为其分配一个新的文件描述符。如果和服务器的 sock 不相同，则按照一般流程处理。如果使用 select 选择了这个客户，且后续 recv 函数接收到的信息长度为 0，我们就可以认为这个客户已经离去，可以关掉它的 socket 了。

和前几次的一个小区别在于服务器需要为每个客户分配一个私有的空间来存储它的地址、端口以及缓存。对此，我们的实现方式，是使用地址类型和动态伸缩的数组。（也可以使用结构体，不过原理是一样的）。

4.5 选做——CGI

CGI 的实现将标志着我们的 Liso Server 走向成熟。能支持 CGI 程序的调用，标志着 Liso Server 拥有无穷的扩展性，不仅能够完成基本的报文解析、GET 消息的回传，还能支持多种应用。在我们的设计中，我们采用了前后端分离的思想，将 CGI 程序设计为后端应用，提供给在服务器端实现的 CGI 接口调用，为前端的请求服务。

具体而言，我们通过在第三部分介绍的数据结构 kv.[c|h] 来方便地对环境变量以及参数进行设置与传递。借鉴 cgi 文件夹中的例子，我们完成了全栈的开发。

前端部分 我们设计了 HTML 网页来与客户交互。通过修改 index.html 的代码，用“表单”将我们的 POST 和 GET 请求方法的页面入口放进去并实现跳转。在实际的测试页面，我们通过 JavaScript 的 "XMLHttpRequest()" 方法向我们的服务器发送 cgi 请求，并通过 document 方法对页面的信息进行动态的修改。

接口部分 后端设计的接口为两个，一个"/cgi/register.py" 另一个为"/cgi/login.py" 接口文档如图4-7 所示。

CGI 登陆

接口描述:

- 进行登陆

请求URL:

- /cgi/login.py

请求方式:

- POST
- GET

参数:

参数名	参数类型	是否必填	参数说明
uName	String	是	用户名
uPass	String	是	用户密码

参数说明:

无

返回示例

- 成功示例

```
[{"Code": "200", "Msg": "User Name Registered before.", "Ip": "202.113.188.249", "Method": "GET", "Name": "Christopher", "Password": "9405Christopher", "Time": "2022-04-12 10:01:54", "UpdatedAt": "2022-04-12 10:01:54", "UpdatedAt": "2022-04-12 10:01:54", "UserName": "Christopher"}]
```

- 失败示例

```
[{"Code": "400", "Msg": "Invalid User name or Password.", "Ip": "202.113.188.249", "Method": "GET"}]
```

返回参数说明

Code: 代表当前的状态，如果是200则正常，否则不正常

备注

无

CGI 注册

接口描述:

- 进行注册

请求URL:

- /cgi/register.py

请求方式:

- POST

参数:

参数名	参数类型	是否必填	参数说明
uName	String	是	用户名
uPass	String	是	用户密码

参数说明:

无

返回示例

- 成功示例

```
[{"Code": "200", "Msg": "You are now registered.", "Ip": "202.113.188.249", "Method": "POST", "Name": "Christopher", "Password": "9405Christopher", "Time": "2022-04-12 10:01:54", "UpdatedAt": "2022-04-12 10:01:54", "UpdatedAt": "2022-04-12 10:01:54", "UserName": "Christopher"}]
```

- 失败示例

```
[{"Code": "400", "Msg": "Invalid User name or User Password.", "Ip": "202.113.188.249", "Method": "POST"}]
```

返回参数说明

Code: 代表当前的状态，如果是200则正常，否则不正常

备注

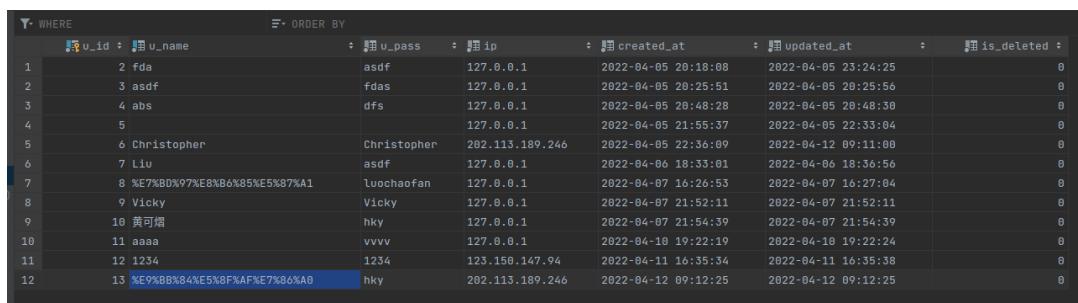
无

a) CGI 登陆接口文档

b) CGI 注册接口文档

图 4-7 CGI 程序接口文档

后端部分 设计了后端数据库结构（如图4-8所示）来存储用户信息，实现了两个 Python 程序，分别处理用户注册和登陆的事务。Python 程序中没有采用高级的编程架构，只采用了基本的编程逻辑：使用 pymysql 库对数据库进行连接，然后通过相应的接口对数据库进行操作。进行数据库操作前需要检查参数是不是空的，避免数据库错误。同时还需要检查有没有数据库注入攻击的情况。对于返回类，我们通过一个“字典”类型的数据结构封装需要传回的信息。对于前端来说，收到的信息就是 Json 格式，能够很方便地通过相关方法进行参数的调用和访问。



The screenshot shows a MySQL database table with 13 rows of data. The columns are labeled: u_id, u_name, u_pass, ip, created_at, updated_at, and is_deleted. The data includes various user entries such as 'fda', 'asdf', 'abs', 'Christopher', 'Liu', 'Luochaofan', 'Vicky', '黄可熠', 'aaaa', '1234', and 'hky'. The 'is_deleted' column contains values 0 or 1.

	u_id	u_name	u_pass	ip	created_at	updated_at	is_deleted
1	2	fda	asdf	127.0.0.1	2022-04-05 20:18:08	2022-04-05 23:24:25	0
2	3	asdf	fdas	127.0.0.1	2022-04-05 20:25:51	2022-04-05 20:25:56	0
3	4	abs	dfs	127.0.0.1	2022-04-05 20:48:28	2022-04-05 20:48:30	0
4	5			127.0.0.1	2022-04-05 21:55:37	2022-04-05 22:33:04	0
5	6	Christopher	Christopher	202.113.189.246	2022-04-05 22:36:09	2022-04-12 09:11:00	0
6	7	Liu	asdf	127.0.0.1	2022-04-06 18:33:01	2022-04-06 18:36:56	0
7	8	%E7%BD%97%E8%B6%85%E5%87%A1	Luochaofan	127.0.0.1	2022-04-07 16:26:53	2022-04-07 16:27:04	0
8	9	Vicky	Vicky	127.0.0.1	2022-04-07 21:52:11	2022-04-07 21:52:11	0
9	10	黄可熠	hky	127.0.0.1	2022-04-07 21:54:39	2022-04-07 21:54:39	0
10	11	aaaa	vvvv	127.0.0.1	2022-04-10 19:22:19	2022-04-10 19:22:24	0
11	12	1234	1234	123.150.147.94	2022-04-11 16:35:34	2022-04-11 16:35:38	0
12	13	黄可熠	hky	202.113.189.246	2022-04-12 09:12:25	2022-04-12 09:12:25	0

图 4-8 Database for CGI

部署阶段 程序部署，采用了 Docker 技术，使用端口映射的技术将 9999 端口进行映射，然后通过 DockerFile 规范每次部署的顺序。通过如图4-9的 Liso.sh 的脚本便于每次更新。更新流程为：在本地编辑好后，向 gitee 仓库提交最新版本。在服务器上运行脚本：先进行 make clean，然后进行 git pull，从仓库中收集新版本，然后进行 docker 进程的删除、docker 镜像的删除。最后进行新镜像的创建以及容器的运行。之所以需要重新创建镜像，是因为一、我们的更新可能会涉及一部分容器依赖的更新；二、Docker 本身是有 Cache 的，如果和之前的镜像创建相同，则会使用 Cache 加速，并不会有太多的时间开销。



```
liso.sh
1 #!/bin/bash
2 make clean
3 git pull
4 make
5 imname=15-441/641-project-1:latest
6 name=Liso
7
8 docker stop ${name}
9 docker rm ${name}
10 docker rmi ${imname}
11
12 path=`pwd`
13 echo docker build -t ${imname} -f ./Dockerfile .
14 docker build -t ${imname} -f ./Dockerfile .
15
16 echo docker run -d -p 9999:9999 -v `pwd`:/home/project-1/ --name ${name} ${imname} ./liso_server
17 docker run -d -p 9999:9999 -v `pwd`:/home/project-1/ --name ${name} ${imname} ./liso_server
18
```

图 4-9 Liso.sh

五 实验结果及分析

5.1 Auto Lab 测试合集

四周 Lab 的 Auto Lab 测试合集如图5-1所示。我们按时，提前完成了每次的 AutoLab 测试。

5.2 第一周——实现简单的 echo web server

在第一周中，我们主要对消息解析和 Echo Web Server 做了本地的测试。

5.2.1 对消息的正确解析

在 Makefile 中增加一项 test_example 来批量测试 example，结果如图4-2所示。其中 Makefile 增加的脚本如下：

Makefile: test example

```
1 test_example: example
2 @for f in $(shell ls samples); do \
3     echo "=====Test file" $$f ====="; \
4     ./example samples/$$f | grep Segmentation --color; \
5     echo "-----\n"; \
6 done
```

结果分析

可以看到结果只在 error2, error3, error4 和 error5 四个情况下出现了 syntax error 的返回值，因为我们在 parser.y 增加了对测试样例中的 400 做了特殊处理，所以不会出现解析错误。且在 Makefile 中将 parse.c 原本的输出给屏蔽了，所以也不会有多余输出。

5.2.2 echo web server

实验操作如下：

1. 在 echo_client 中增加直接从文件中读取的支持；
2. 先打开 echo_server；
3. 然后通过 echo_client 发送一系列文件，查看 echo_server 的结果。

实验结果截图及分析

实验结果如图5-2所示。可以看到，在 head, get 和 post 方法的实验中，我们实现了 echo 的效果。在 400 和 501 的输入中，我们实现了相应的返回消息。在

《计算机网络》课程设计报告

AUTOLAB

grading Gradebook clear_all Jobs 3020202184 刘锦帆 arrow_drop_down

home » SocketProgramming-sgt (2022spring) » week1-echo server » Handin History

Ver	File	Submission Date	lab1 (100.0)	Late Days Used	Total Score
2	1051666563@qq.com_2_Liso.tar	2022-03-16 18:47:15 +0800	100.0	Submitted 0 days late	100.0
1	1051666563@qq.com_1_Liso.tar	2022-03-15 22:42:18 +0800	100.0	Submitted 0 days late	100.0

Page loaded in 0.016131756 seconds

Autolab Project · Contact · GitHub · Facebook · Logout

v2.7.0

a) Lab1 AutoLab 测试结果

AUTOLAB

grading Gradebook clear_all Jobs 3020202184 刘锦帆 arrow_drop_down

home » SocketProgramming-sgt (2022spring) » week2-get head post » Handin History

Ver	File	Submission Date	lab2 (100.0)	Late Days Used	Total Score
9	1051666563@qq.com_9_Liso.tar	2022-03-24 13:38:09 +0800	100.0	Submitted 0 days late	100.0
8	1051666563@qq.com_8_Liso.tar	2022-03-24 13:29:16 +0800	70.0	Submitted 0 days late	70.0
7	1051666563@qq.com_7_Liso.tar	2022-03-24 13:28:06 +0800	100.0	Submitted 0 days late	100.0
6	1051666563@qq.com_6_Liso.tar	2022-03-23 16:32:00 +0800	0.0	Submitted 0 days late	0.0
5	1051666563@qq.com_5_Liso.tar	2022-03-22 14:49:09 +0800	0.0	Submitted 0 days late	0.0
4	1051666563@qq.com_4_Liso.tar	2022-03-22 19:21:18 +0800	0.0	Submitted 0 days late	0.0
3	1051666563@qq.com_3_Liso.tar	2022-03-22 18:18:14 +0800	0.0	Submitted 0 days late	0.0
2	1051666563@qq.com_2_Liso.tar	2022-03-22 18:17:21 +0800	0.0	Submitted 0 days late	0.0
1	1051666563@qq.com_1_Liso.tar	2022-03-22 18:14:46 +0800	0.0	Submitted 0 days late	0.0

Page loaded in 0.021023627 seconds

Autolab Project · Contact · GitHub · Facebook · Logout

v2.7.0

b) Lab2 AutoLab 测试结果

AUTOLAB

grading Gradebook clear_all Jobs 3020202184 刘锦帆 arrow_drop_down

home » SocketProgramming-sgt (2022spring) » week3-pipeline » Handin History

Ver	File	Submission Date	lab3 (100.0)	Late Days Used	Total Score
3	1051666563@qq.com_3_Liso.tar	2022-04-01 22:04:12 +0800	100.0	Submitted 4 days late	100.0
2	1051666563@qq.com_2_Liso.tar	2022-04-01 22:02:02 +0800	0.0	Submitted 4 days late	0.0
1	1051666563@qq.com_1_Liso.tar	2022-04-01 16:21:49 +0800	100.0	Submitted 0 days late	100.0

Page loaded in 0.019090271 seconds

Autolab Project · Contact · GitHub · Facebook · Logout

v2.7.0

c) Lab3 AutoLab 测试结果

AUTOLAB

grading Gradebook clear_all Jobs 3020202184 刘锦帆 arrow_drop_down

home » SocketProgramming-sgt (2022spring) » week4-concurrent » Handin History

Ver	File	Submission Date	lab4 (100.0)	Late Days Used	Total Score
4	1051666563@qq.com_4_Liso.tar	2022-04-01 22:02:27 +0800	100.0	Submitted 0 days late	100.0
3	1051666563@qq.com_3_Liso.tar	2022-04-01 19:08:19 +0800	100.0	Submitted 0 days late	100.0
2	1051666563@qq.com_2_Liso.tar	2022-04-01 19:08:16 +0800	100.0	Submitted 0 days late	100.0
1	1051666563@qq.com_1_Liso.tar	2022-03-29 17:02:44 +0800	100.0	Submitted 0 days late	100.0

Page loaded in 0.012866053 seconds

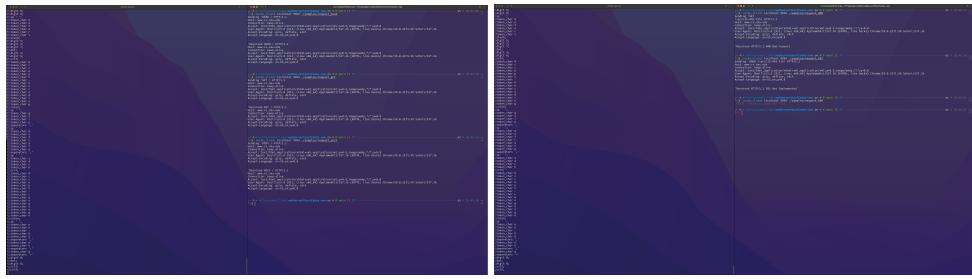
Autolab Project · Contact · GitHub · Facebook · Logout

v2.7.0

d) Lab4 AutoLab 测试结果

图 5-1 AutoLab 测试合集

输入文件不存在时，我们也得到了相应的正确返回，体现了我们程序的鲁棒性。



a) Head get and post actions

b) 400 and 501

图 5-2 echo web server

5.3 第二周——实现 HEAD、GET、POST 方法

对于第二周的实验，我们使用了 API FOX 测试 GET, HEAD 以及 404 的返回（如图5.4a）；使用浏览器测试服务器是否顺利返回文本、图片等（如图5-3）；使用 liso_client 测试新的缓冲区管理办法（如图4.5a）。均通过。

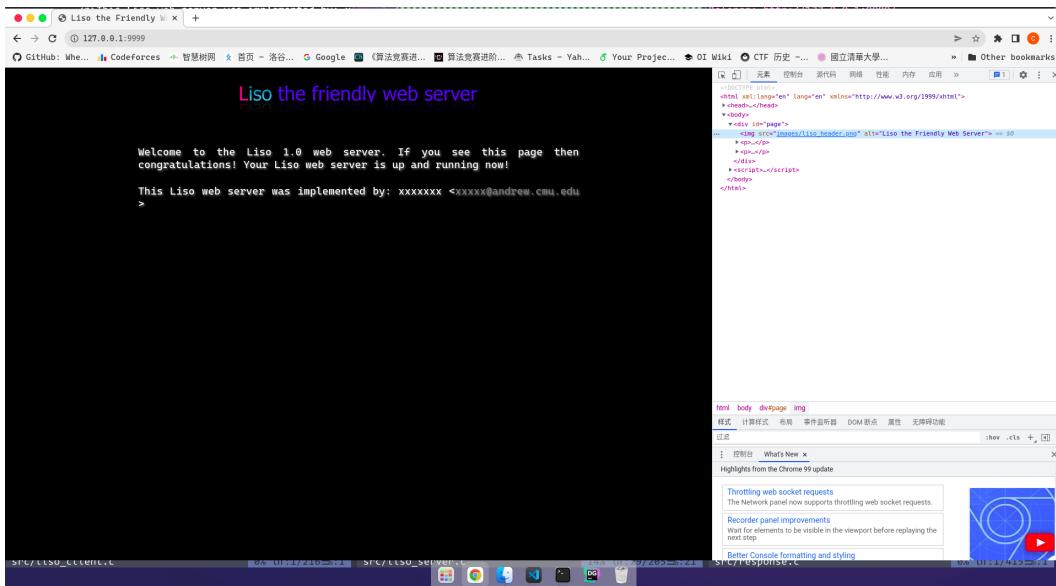


图 5-3 Liso Web Test

5.3.1 结果分析

在浏览器的测试中，发现和使用 liso_client 测试不同，当出现有 404 的情况时，我们的 liso_server 不会立刻进入下一轮监听，而是一直处于处理请求的情况。我们推测，是由于没实现 chunk 导致的。

5.3.2 日志

日志信息如图5.4c) 所示。在日志信息里可以看到，我们的 recv 操作，每次最多接收 1025 字节的信息，单次可能不能完全接收一份报文的所有信息，但通过我们动态数组的管理，我们能够轻易完成 Pipeline 的测试。

a) Liso API FOX Test

b) Liso Test

c) 部分 log 输出

```

G:\...\liso>cd /d liso\server
G:\...\liso\server>java -jar webServerStartCodes-new.jar on & P mode 125 77
...
Echo Server ...
[Sat Mar 26 23:35:21 2022] [DEBUG LOG] Start Listening at port 9999
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] Msg received: "1025" from client: 127.0.0.1:34286
...
PRINT Dynamic Buffer 1024
...
GET / HTTP/1.1
Host: www.cs.cmu.edu
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.99 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8

HEAD / HTTP/1.1
Host: www.cs.cmu.edu
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.99 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8

DELETE /-prv/15-441-F15/ HTTP/1.1
Host: www.cs.cmu.edu
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.99 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8

...
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] TRY TO PARSE
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] Parsing MSG
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] Status Line: GET / HTTP/1.1
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] Host: www.cs.cmu.edu
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] Connection: keep-alive
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.99 Safari/537.36
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] Accept-Encoding: gzip, deflate, sdch
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] Accept-Language: en-US,en;q=0.8

[Sat Mar 26 23:35:32 2022] [Access] [pid 232031;id 140310104237976] [Client 127.0.0.1:34286] GET 200 OK
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] Msg Appended
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] Starting Free db expect request-header
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] CURRENT_CNT: 1
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] Starting dealing with msg
...
338
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] TRY TO PARSE
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] Parsing MSG
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] Status Line: HEAD / HTTP/1.1
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] Host: www.cs.cmu.edu
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] Connection: keep-alive
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.99 Safari/537.36
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] Accept-Encoding: gzip, deflate, sdch
[Sat Mar 26 23:35:32 2022] [DEBUG LOG] Accept-Language: en-US,en;q=0.8

```

图 5-4 测试结果展示

5.4 第三周——实现 HTTP 的并发请求

5.4.1 pipelining 测试

如图5-5，我们在 request_pipeline 中客户端同一个 TCP 向服务器发送多个并发请求。我们根据服务器返回的实验结果来看（以最后几个为例）。第一个请求是 GET 方法，格式等也均是正确的，服务器也能正确解析返回给客户端消息；而第二个是 HAHA 方法，这个我们没有实现过，即返回 “HTTP/1.1 501 Not Implemented\r\n\r\n”。

而第三个是/ prs/15-441-F15/HTTP/1.1，存在格式错误，因此服务器返回“HTTP/1.1 400 Bad request\r\n\r\n”。服务器能够识别这个错误请求并拒绝了这个请求，也能继续识别并发的下一条请求。下一条请求 HTTP/1.3.0 也存在格式错误，同样，服务器能够在拒绝上一条错误请求之后解析这条请求。根据结果来看，我们的实现是正确的。对于这个结果，我们也容易从代码分析出来这是正确的。我们引入了 strstr(str,dest) 函数，我们就能找到”\r\n\r\n”的位置。找到了”\r\n\r\n”的位置之后，也很容易找到单个报文，我们之前已经实现了正确解析单个报文。所以我们处理出来了单个报文之后也就能正确处理这些并发的请求了。

The screenshot shows a terminal window with two panes. The left pane contains the source code for `liso_server.c` and `liso_client.c`. The right pane shows a browser window displaying the Liso server's response to a pipeline test. The browser output includes several log entries from the server, such as "Starting Free db" and "HTTP/1.1 400 Bad request". The terminal also shows command-line arguments like `-ECHO_PORT 9999` and `-N 4`.

```

src/liso_server.c
src/liso_client.c

```

图 5-5 pipelining test

5.5 第四周——实现多个客户端的并发处理

5.5.1 手工测试

我们将 `liso_client` 设置为忙等待，及其不会自动关闭 socket 并退出，而是占用着服务器的资源等待着。如图5-6，我们同时开启了 4 个陷入忙等待的客户端对服务器发出请求，服务器仍然能正常处理发出 pipeline 请求的客户端。

5.5.2 Apache 测试

第三周的 `liso_server` 由于没有进行并发优化。在测试时由于会一直等待客户端提出退出申请，在测试时会出现 Apache Bench 忙等待。我们分别进行了并发压力测试以及多请求压力测试。测试截图统一放在附录中。

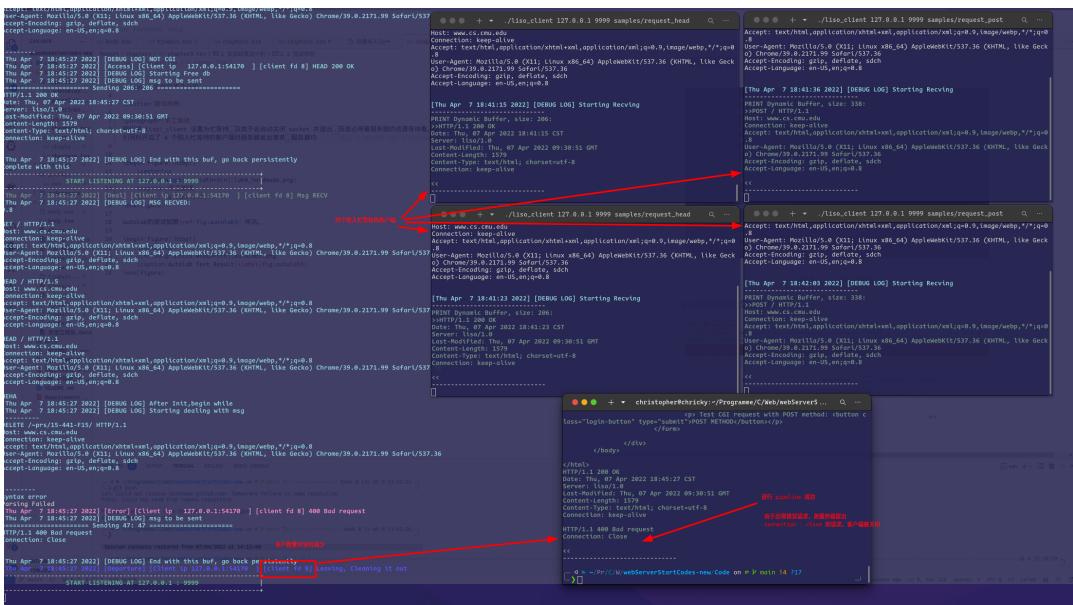


图 5-6 手工测试

并发压力测试 控制总请求数不变的情况下，我们分别进行了并发数量为 10、100 以及 1000 的测试，结果如图所示。如表5-1 所示，总请求数量不变情况下，随着并发等级的提高，单个请求的平均反应时间并没有太大的变化。所以可以认为我们的方案是能够解决高并发问题的。

并发等级	总请求数量	TPR(ms)
1 10	1000	0.378
2 100	1000	0.390
3 1000	1000	0.429

表 5-1 并发压力测试结果

多请求压力测试 通过控制并发等级为 10 不变，改变总请求数量，我们得到了多请求压力测试的结果，如表5-2所示。可以轻易看出，在并发等级一定的情况下，总请求数量在 100,000 量级及以下时，单个请求的平均反应时间都能在正常的范围内。但当请求量达到 1,000,000 数量级时，单个请求的平均相应时间则会增加很多，推测是因为请求数量太多，导致频繁接受、删除用户，进而导致反应时间增加。

5.6 选做——CGI

我们在上一章中介绍了我们 CGI 程序完整的实现流程。再服务器部署完成后，我们在手机上进行测试。截图如下。

并发等级		总请求数量	TPR(ms)
1	10	100	0.395
2	10	1000	0.378
3	10	10000	0.364
4	10	100000	0.445
5	10	1000000	0.781

表 5-2 多请求压力测试



a) Login 成功

b) Password 错误

图 5-7 测试 1

六 个人总结

附 录

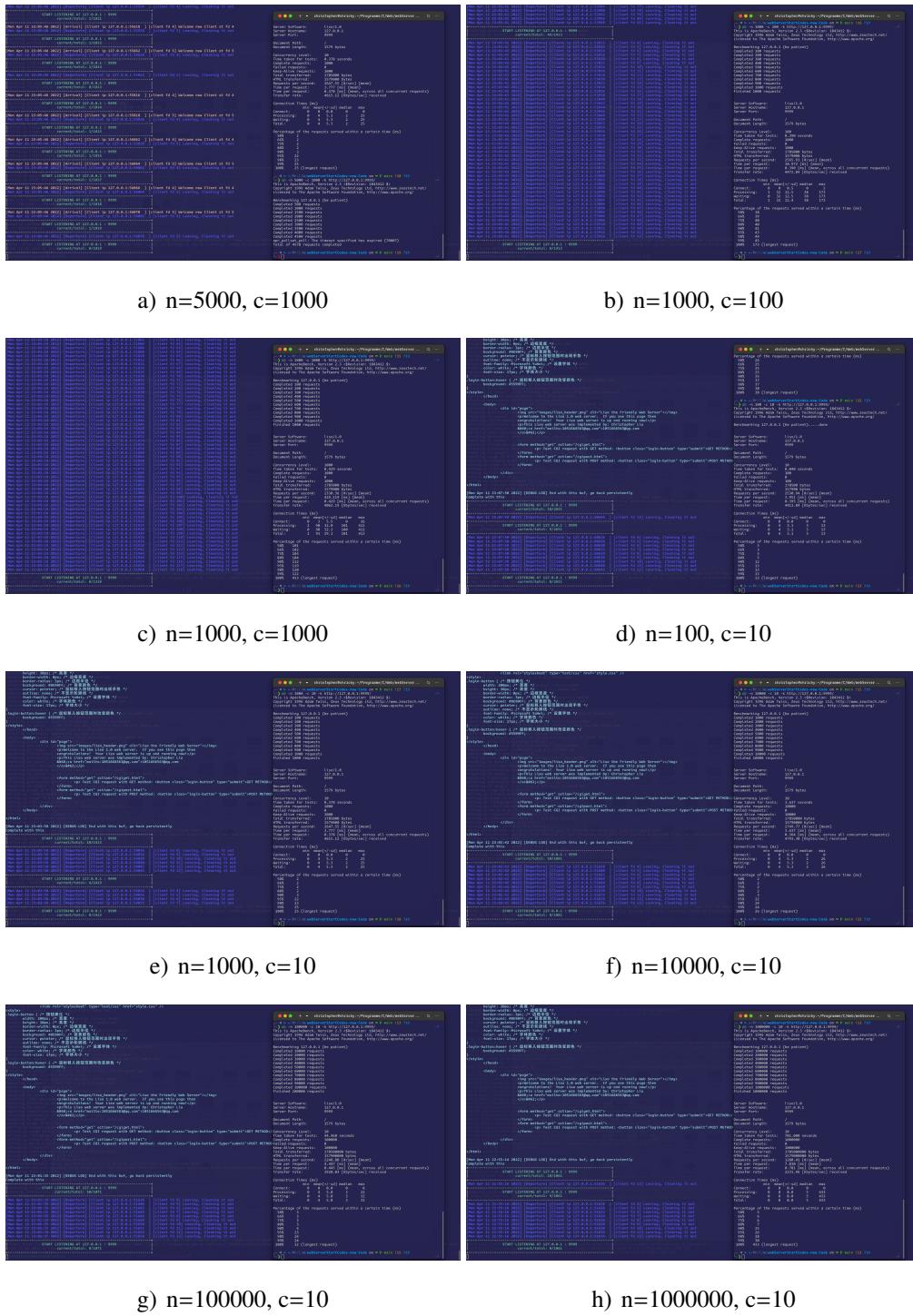


图 6-1 Apache Bench 测试