四川大學

计算机网络和分布式系统
开发过程文档

题　　目 Multi-thread webserver

学　　院 建筑与环境学院

专　　业 力学-软件工程交叉实验班

学生姓名 管筠箫

学　　号 2019141470422 年级 2019

指导教师 宋万忠

二０二一年06月26日

目录

一、 程序讲解

1、多线程 web server 搭建

1.1 首先在 web 访问页面需要一个 server，这也是本次作业的基本要求。我们将 WebServer 设计为一个类，创建 server 对象时即完成部分 tcp 建立连接和绑定操作。Server 从调用 run 方法开始运行，run 方法的监听套接字循环监听是否有客户端连接，若有就分配一个 connection_socket，并创建一个线程来处理该客户端事件。（注意这里采用的是最简单的多线程处理方式，没有限制客户端的请求，理论上无限创建线程处理任务，这里关于线程及相关后续可优化处理见下文三、四两点）

1.2 线程的处理 target 指向的是 serve_client，我们来看这个方法，socket 接收客户端传来的 request 请求，utf-8 解码为字符串。（这里浏览器传来的请求不会有中文字符，故不需要使用 GBK，而且 utf-8 编码范围更广，可以避免一些错误，关于编码问题可见后文问题解决）收到浏览器的 request 如下图所示，对我们来说有用的信息其实只有第一行请求的文件名，于是我们先分割出第一行，然后很自然的想到用正则表达式匹配字符，提取出客户端想要的文件名，这里关于正则表达式的匹配就不做过多阐述，程序将客户端的请求文件名都打印在了控制台上，便于调试处理。

```
GET / HTTP/1.1
Host: 127.0.0.1:7777
Connection: keep-alive
sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="90", "Google Chrome";v="90"
sec-ch-ua-mobile: ?0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KH
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,ima
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8
```

如果客户端没有请求某个文件，而是直接访问服务端 ip,我们做一些处理将请求改为/login.html,页面自动跳转至登录页面。接着我们可以看到程序中出现了一段比较丑陋的代码，这是由于在前端 submit 提交时均默认采用了 get 方式，将用户名、密码都附在了 url 内部。

```
    <form action="/loginsuccess.html" id="login">
```

我们在处理时就要进行特殊判断用 deal_dynamic 来处理。如果想要优化应该使用前端 html 应该写为 post 来提交请求,服务器端也就要添加相应的对 post 报文的处理函数。接着我们就进行到了服务端对动态和静态资源的处理。

1.3 静态资源的请求处理比较简单，看到 deal_static 方法，首先以二进制形式打开 static 文件夹的文件资源，存在该资源状态码就为 200，不存在就为 404。这里对于 header 做了很多简化，只增加了 content-length 字段，这是保证长连接的一个重要字段。关于 header 字段比如 content-type,keep-Alive，后文问题解决中有更详细的阐述。总的来说服务端回应的报文就是 header 字段，客户端请求的文件内容就在 body 中，两者以空行分割。

1.4 动态资源的处理 deal_dynamic，首先解释一下什么是动态资源，动态资源通俗来讲就是不同用户访问看到的界面不一样，这是由于文件中包含了从数据库中读取的资源，故服务端先执行数据库请求处理，转化为静态资源后才发送给用户。（部分不涉及数据库操作的 html 文件实际上来说也应该算作静态资源的请求，即用户看到的界面是相同的，比如登陆界面，但对于 html 文件我们统一用动态资源方式处理）这里就涉及到与框架之间的叫交互，及 server 通过 WSGI 接口与框架调通。接下来我们就说到第二点框架 toy_frame 的搭建

1.5 最后 server 还遗留了一个问题，就是服务端何时调用 close，我们在程序中采用了最简单的短连接方式，就是发送完成后就直接关闭，但有关长连接短连接的问题也是在后文问题解决中有详细讨论。

2、Toy_frame 搭建

2.1WSGI 接口 application 函数。上文提到了 web 框架，利用框架所要实现的作用就是对不同的动态资源映射到不同的函数进行处理，server 和框架之间由 WSGI 接口中的一些协议进行调通，而最简单的就是实现一个 application 函数就可以完成两个模块的数据传递。

```
def application(env, set_response_header):
    # 'Content-Type:text/html;charset=utf8'这个header不能少，否则无法识别文件类型
    # 这里的charset编码实际上就是connection_socket稍后发送将采用的编码，需要保持一致
    # windows下不指定Content-Type默认就要采用GBK
    set_response_header("200 OK", [('Content-Type', 'text/html;charset=utf8')])
    file_name = env['PATH_INFO']
```

2.2header 的处理。程序中 env 字典就是 server 要传递给框架的部分数据，在本程序中为了简化操作只用到了客户请求的文件名 file_name。同样 toy_frame 框架也要在 application 函数内添加部分头部内容返回给 server 拼接为完整的 header，

set_response_header 就是 server 暴露给框架的函数修改 header。Application 的返回值就是 body,server 将 header+body 拼装发给客户端。

2.3body 的处理，对于客户端请求的 file_name，需要遍历全局字典变量 URL_DICT 查询。这里面存储了正则表达形式的 URL 和相应处理函数的映射，URL_DICT 内所存储的键值对是通过装饰器来完成的，程序解析到@route 时都会执行该装饰函数，注意到 call_func 内采用*args,**kwargs 是为了可以处理所有参数传递情况。另外一点，装饰器的参数采用的是正则形式的 url,这样是为了一个函数可以对应多个形式类似的 url,在后文中可以看到有些不同的 url 可以使用相同的函数进行处理。

3、前端页面及数据库查询

3.1 登录页面，用户由 7777 端口访问服务器 ip,首先默认就跳转至登陆页面，管理员和用户就简单的通过用户名来区分，这里为了简单起见前端 submit 提交数据使用 get 方式，作为请求附在 url 那日以?和&来分隔，再将用户名和密码加密存储进数据库。实际上用 post 提交更好，但这又涉及到提交数据格式等相关问题，而且程序对于用户未登录直接访问某个 url 也不会进行跳转处理。这些限于本人前端不太熟悉，程序也不太完善，所以页面仅仅呈现一个展示作用。

3.2 index 首页，首页这里呈现的数据就是通过从 stock 数据库的 Information 表内查询得到，在框架内的处理函数中用这些数据替换 index.html 页面的内容，完成数据的展示，增删关注股票就是相应的对数据库进行修改，这里涉及到的是数据库相关操作，也就不过多的阐述。非管理员用户的界面也很类似，只是没有修改的功能。

3.3 center 首页，这里功能主要就是修改股票备注和查看股票走势，这两者跳转到新的 url 只有对应的 code 不同，这里就对应到了上文不同的 url 使用相同的函数进行处理，因为处理过程都是类似的，只是浏览器发起请求的 url 不同。

3.4 pydb 文件内的 MysqlFunc 类就是封装的对数据库的一些基础操作，在 toy_frame 中内还有 fetchone_sql fetchall_sql 返回查询到的信息。

4、股票数据和每只股票历年数据的获得

关于数据的获得可见 stock_data 下的两个文件，一个爬取股票的信息，存储进数据库的话就要进行一些数据清洗工作，另一个获得的是某只股票的历年信息，但每个 ip 的信息获取在每个时间段内有限制，不能无限制的获取。

5、数据库的设计

```
mysql> desc information;
+-----------+---------------+------+-----+---------+----------------+
| Field     | Type          | Null | Key | Default | Extra          |
+-----------+---------------+------+-----+---------+----------------+
| id        | int unsigned  | NO   | PRI | NULL    | auto_increment |
| code      | varchar(6)    | NO   |     | NULL    |                |
| name      | varchar(10)   | NO   |     | NULL    |                |
| price     | decimal(10,2) | NO   |     | NULL    |                |
| amplitude | varchar(10)   | NO   |     | NULL    |                |
| turnover  | varchar(10)   | YES  |     | NULL    |                |
| high      | decimal(10,2) | NO   |     | NULL    |                |
| low       | decimal(10,2) | NO   |     | NULL    |                |
+-----------+---------------+------+-----+---------+----------------+
```

Informatioin 表存储的就是完整的股票，使用了 id 作为自增主键，其余字段分别为股票代码 code、股票名称 name、股票今日开盘价格 price、涨跌幅度 amplitude、换手率 turnover、高点 high、低点 low。有百分比的数据使用 varchar，数字采用 decimal,自增 id 使用 int unsigned。

```
mysql> desc focus;
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| id        | int unsigned | NO   | PRI | NULL    |       |
| note_info | varchar(200) | YES  |     |         |       |
| info_id   | int unsigned | YES  | MUL | NULL    |       |
+-----------+--------------+------+-----+---------+-------+
```

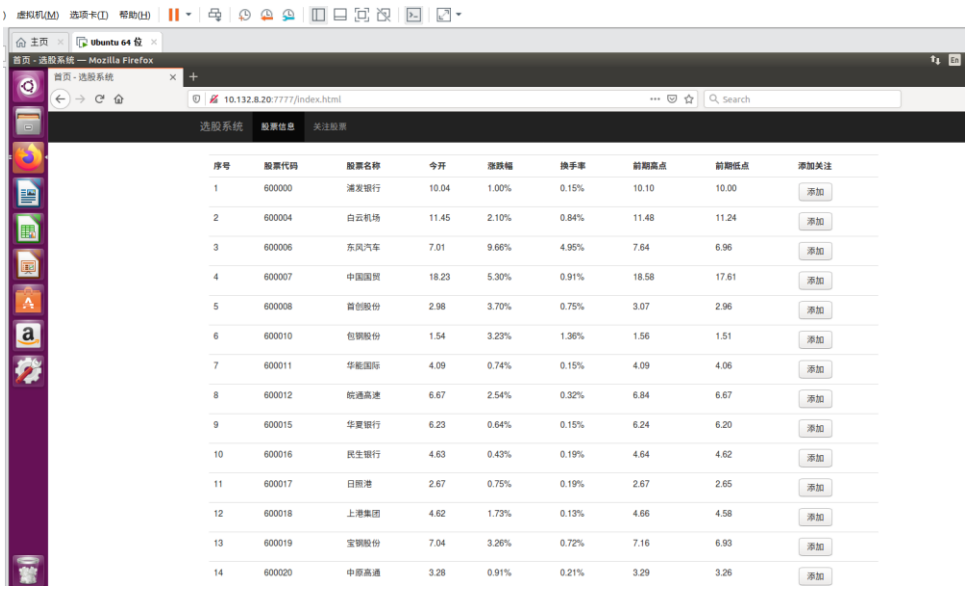
Focus 表有 info_id 引自 information 表中的 code

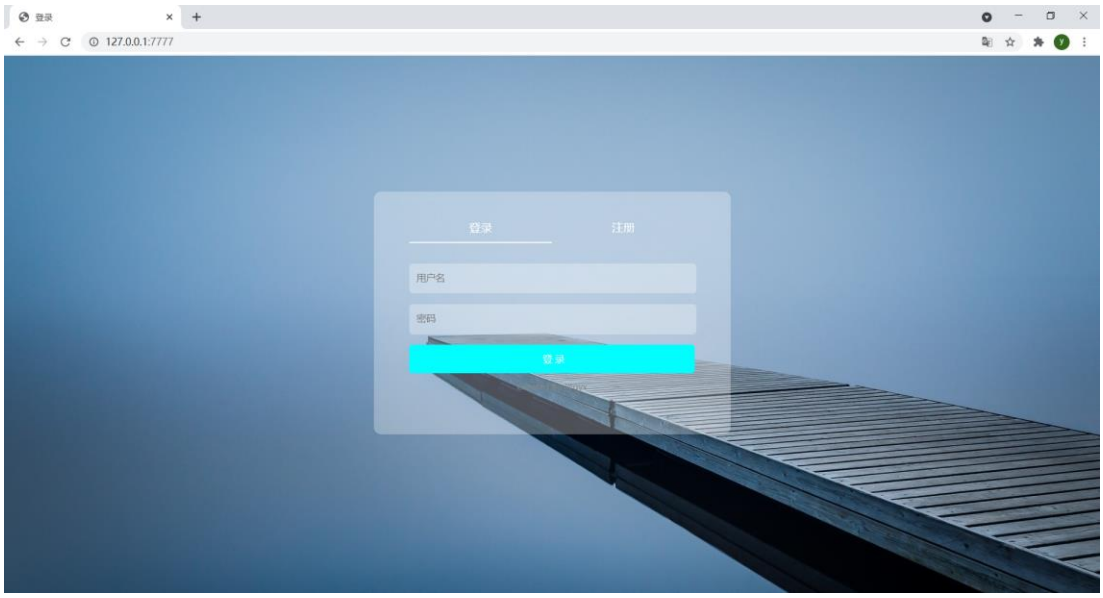
Stock_user 表存储用户，加密存储密码

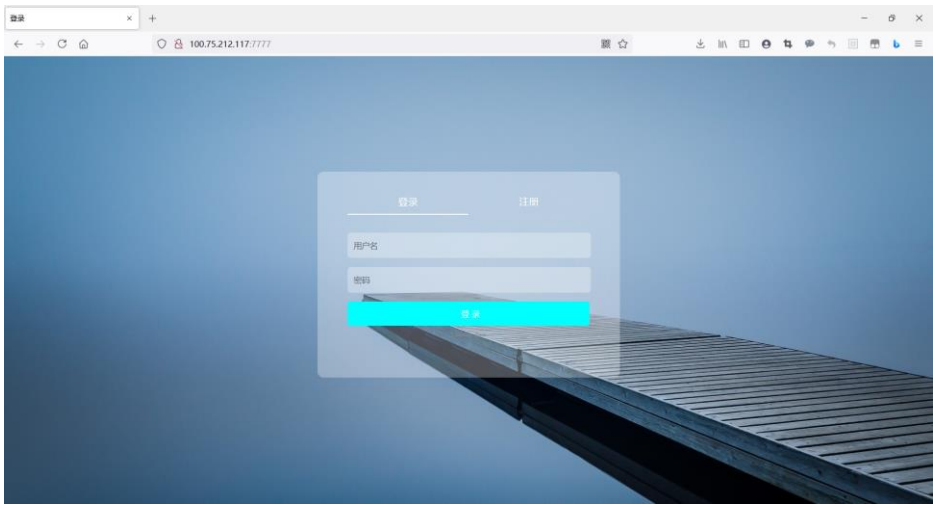
```
+-------+-----------------+------+-----+---------+-------+
| Field | Type            | Null | Key | Default | Extra |
+-------+-----------------+------+-----+---------+-------+
| name  | varchar(50)     | NO   | PRI | NULL    |       |
| pwd   | varbinary(255) | YES  |     | NULL    |       |
+-------+-----------------+------+-----+---------+-------+
```

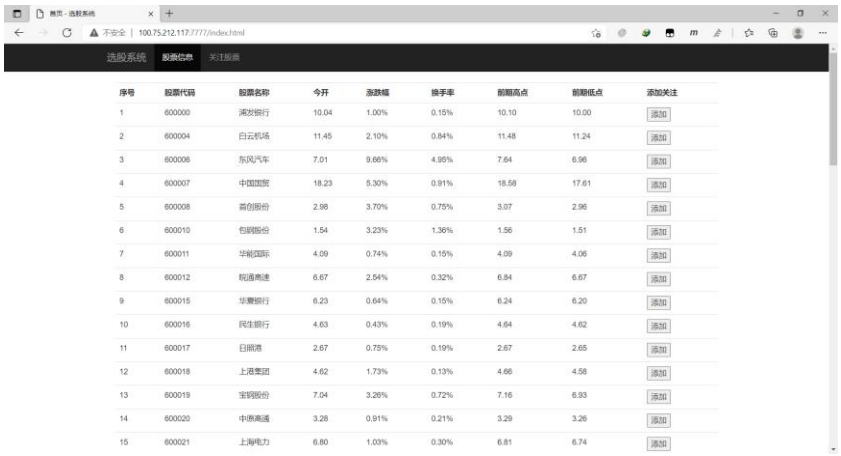
```
+------+-------------------+
| name | pwd               |
+------+-------------------+
| abc  | ?_??|???u???????  |
| gyx  | ?h??ЛM?--????PQ   |
| root | ?h??ЛM?--????PQ   |
| sad  | r?3?^s)?????      |
+------+-------------------+
```

二、 测试结果









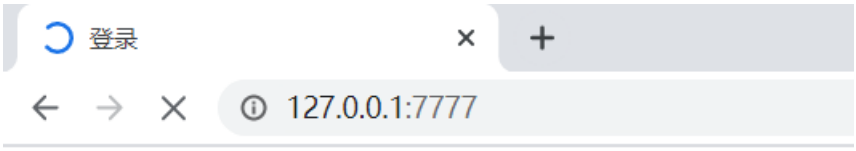
该程序在同一局域网下都可以进行访问，在 linux windows 下均可以正确显示, firefox chrome edge 也均进行测试。

三、 问题及解决

1、关于多线程

本次作业中要求使用多线程来完成 webserver 任务，但为了程序可以实现并发可以有多种方式，比如线程池、协程、多进程或进程池，或者可以采用非阻塞的 recv()来处理，监听到一个客户端就放进一个 socket 列表中，进行轮询处理。

1.1 多进程处理。在对比多进程和多线程时可以发现，仅仅把多线程的代码作如下修改，浏览器会出现一直转圈加载的情况，这种情况就是四次握手的没有完成。通过 wireshark 抓包也确实证明了这一观点，最后的四次挥手没有结束。




```
t1 = threading.Thread(target=self.serve_client, args=(connection_socket,))
t1.start()
# p1 = multiprocessing.Process(target=self.serve_client, args=(connection_socket,))
# p1.start()
```

```
  44 51438 → cbt(7777) [FIN, ACK] Seq=829 Ack=705 Win=2618880 Len=0
  44 cbt(7777) → 51438 [ACK] Seq=705 Ack=830 Win=2619648 Len=0
  63 cbt(7777) → 51439 [PSH, ACK] Seq=1 Ack=836 Win=2619648 Len=19 [TCP se
  44 51439 → cbt(7777) [ACK] Seq=836 Ack=20 Win=2619648 Len=0
1556 cbt(7777) → 51439 [PSH, ACK] Seq=20 Ack=836 Win=2619648 Len=1512 [TCP
  44 51439 → cbt(7777) [ACK] Seq=836 Ack=1532 Win=2618112 Len=0
  45 [TCP Keep-Alive] 51439 → cbt(7777) [ACK] Seq=835 Ack=1532 Win=2618112
  56 [TCP Keep-Alive ACK] cbt(7777) → 51439 [ACK] Seq=1532 Ack=836 Win=261
  45 [TCP Keep-Alive] 51439 → cbt(7777) [ACK] Seq=835 Ack=1532 Win=2618112
  56 [TCP Keep-Alive ACK] cbt(7777) → 51439 [ACK] Seq=1532 Ack=836 Win=261
```

于是查阅相关资料发现所谓的 socket 在底层也就是一个文件，由文件描述符 file descriptor 表示。而多进程是写时拷贝，不共享全局变量的，单个进程调用 close 关闭

并不能使得 socket 关闭，需要两个指向 socket 的进程都关闭才是真的关闭。因此使用多进程，除了在子进程函数 serve_client 中调用 close，还需要在主进程也就是 main 函数中调用 close 才能真正关闭这个进程连接，完成四次挥手。

而在多线程情况下由于是共享全局变量的，故只需要关闭一次，即只有一个主进程指向底层 socket 文件，其余子线程都共享的是主进程那块内存空间。

1.2 非阻塞 recv，进程池

其实我们可以看到无论是多进程还是多线程，都使用到了阻塞函数来说接收请求，也就是说客户端连接上后，如果 client 迟迟不发数据，该线程或进程就是处于阻塞状态的，资源就是被浪费的。那我们想的是充分利用这段被阻塞的时间，所以就有了非阻塞的 recv 来接收数据。

在单进程情况下，我们将建立好连接的 socket 加进连接列表 client_socket_list 内，然后不断轮询该列表，只有轮询道了某个 client，并且 client 发送了数据才会为这个 socket 进行处理。这样服务端的资源就得到了充分利用，只是某个 socket 如果已经发送了数据，但是还没有被轮询到的话，就处于等待状态，核心测试代码如下。

```
while True:
    try:
        # 监听套接字等待连接
        connection_socket, client_addr = tcp_socket.accept()
    except Exception as ret:
        pass
    else:
        # 设置为非阻塞
        connection_socket.setblocking(False)
        client_socket_list.append(connection_socket)

    for client_socket in client_socket_list:
        try:
            recv_data = client_socket.recv(1024).decode("GBK")
        except Exception as ret:
            pass
        else:
            if recv_data:
                serve_client(client_socket, recv_data)
                print("列表长度为%d " % len(client_socket_list))
                print(client_socket_list)
            else:
                # 若浏览器发来空数据默认客户端关闭，将连接socket移除轮询列表
                client_socket_list.remove(client_socket)
                print("else列表长度为%d " % len(client_socket_list))
                client_socket.close()
```

如上单进程的轮询用户的体验感并不好，使用这种形式的一个问题是如何与多线程或多进程进行结合考虑充分利用资源，这里我认为其实就和进程池、线程池很类似，每个 pool 中最多加进多少，而每个每个线程或进程管理多大的 socket 列表比较合适。关于处理多少个 socket 的还没有找到一个比较好的解释，但也算是一个小的想法，以后可以尝试解决。

1.3 协程

协程又是一个更小的实体单位，我们可以粗浅的理解为协程依附于线程，而线程依附于进程。协程所要处理的问题也是充分利用线程阻塞所浪费的那段时间，可以查阅到 python 中已将封装好相对应的库 gevent，将线程改成对应使用 gevent 的函数即可，但注意要将所有的延时操作都换为 gevent 中重写的延时操作，gevent 库提供了便捷的替

换方式，执行 monkey .patch_all 即可全部替换。同样协程也可以用多协程来提高效率，这里就不多赘述。

```
t1 = threading.Thread(target=self.serve_client, args=(connection_socket,))
t1.start()
```

```
gevent.spawn(serve_client, new_socket)
```

```
from gevent import monkey

monkey.patch_all()
```

1.4 linux 下 epoll 与 windows 下 IOCP

除了以上处理，我们当然还得去看看比较优秀的开源代码是如何完成 web server 功能的，可以查阅到 nginx 使用的是 epoll 机制来实现的，而不是我们以为的协程或是多进程一类的机制，epoll 的实现我们可以简单进行一些解释。

在上文我们谈到使用非阻塞的 recv 函数时，曾经提到过因为要对维护的 socket 列表进行轮询，所以效率其实不够高，有数据发送的 client 体验感并不好。Epoll 实现的机制就是不用轮询，那个 client 有数据发送了，就由操作系统告知 server 进行处理，省去了轮询没有数据发送的 socket 的时间。在 windows 下我尝试了使用 epoll，但发现无法使用。原因是 windows 和 linux 下对于 select 模块的实现有些区别，所以 windows 下是无法使用 epoll 机制的，相对应的实现可能是 IOCP 模块，但这里由于事件原因我也就没有再去进行更多的尝试。

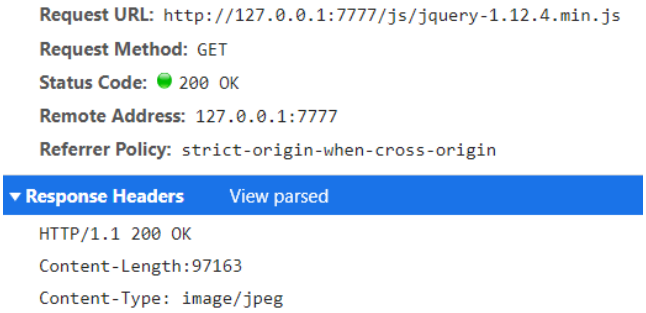
2、关于 header 字段

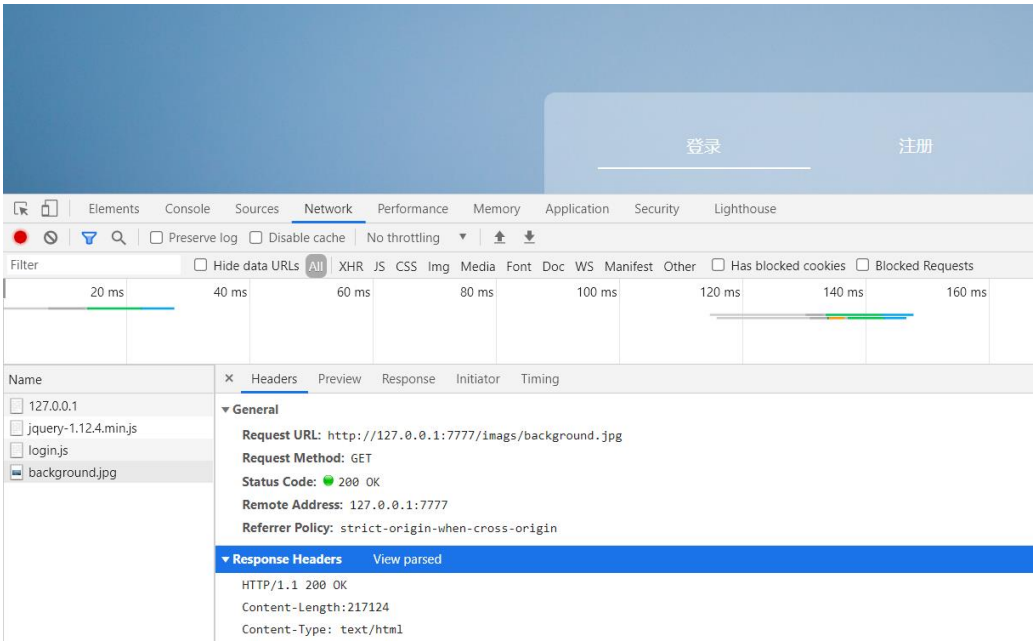
2.1 Content-Type 应该是比较重要的一个字段，浏览器就是依靠 Content-Type 来判断响应的内容类型。浏览器并不靠 URL 中的文件类型来判断响应的内容。所以看以看到我们在处理 html 资源时加上了 text/html，代表我们将要发送 html 文件。

```
set_response_header("200 OK", [('Content-Type', 'text/html;charset=utf8')])
```

那这里就出现了一个问题，为什么在短连接时没有加上 Content-Type 字段呢，我们可以发现一些很有趣的事情，在处理静态资源请求时添加上 Content-Type 字段，所有静态资源我们都设置为 image/jpeg，即以图片格式来读取，那理论上来说，像 js css 这种文件就是无法正确读取的。我们可以看到在 chrome 浏览器下看到 js 文件格式确实是以 image/jpeg 接收到的，但仍然可以正确显示。将 jpeg 文件的 Content-Type 设置为 text/html 也仍然可以正确显示。

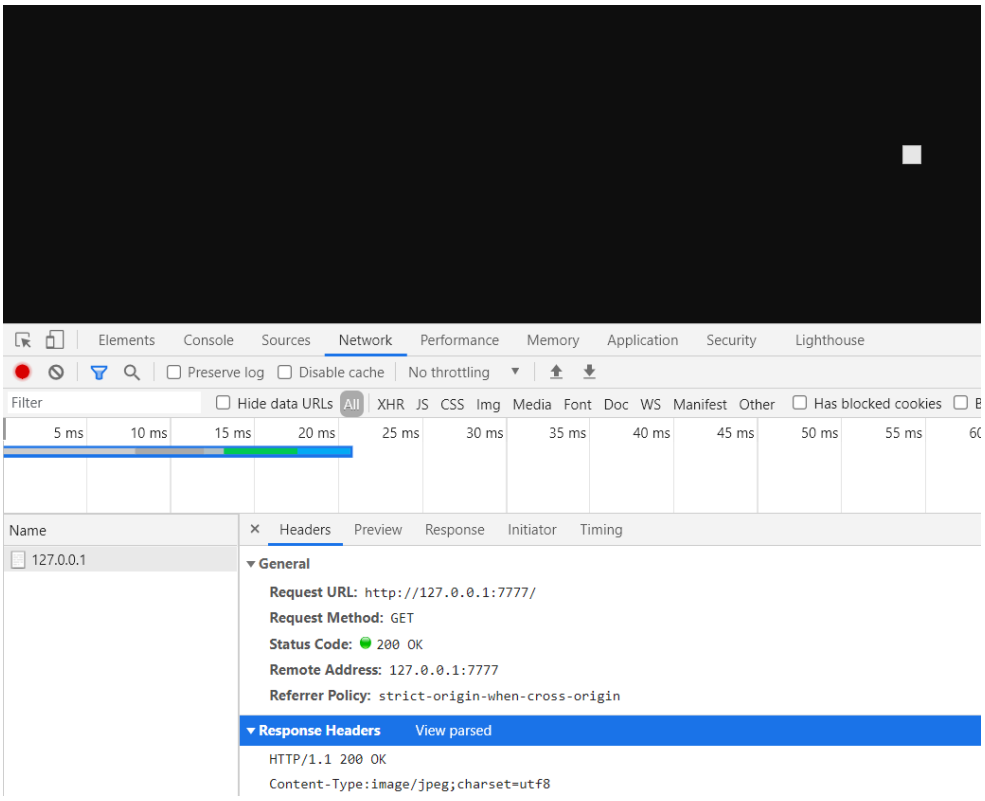
```
# 回应请求 浏览器请求回应里面回成用\r\n表示
response = "HTTP/1.1 200 OK\r\n"
# 读出服务器上存储的静态资源内容
html_content = f.read()
response += "Content-Length:%d\r\n" % len(html_content)
response += 'Content-Type: image/jpeg\r\n'
# response += "Transfer-Encoding: gzip, chunked\r\n"
# response += "Keep-Alive: max=5, timeout=1\r\n"
# 回应里面加上一个空行，隔开头部和内容
response += "\r\n"
```





而当我们再用修改html的content-type时却出现了错误，浏览器无法正确显示格式，并且是在解析到 html 文件后就不再发起请求了。

```
set_response_header("200 OK", [('Content-Type', 'image/jpeg;charset=utf8')])
```



再进行尝试发送 html 时不加上 content-type 字段，发现各个界面也都可以正确显示

由此我们合理猜测如果不加上 content-type 字段，就由浏览器自动解析是什么类型的文件，再显示出来。但对于 html 类型的文件如果设置错误，就会浏览器就会按照设置的 content-type 来解析，出现读取错误的情况。但对于其他一些静态资源比如 jpg css 等，似乎设置错误浏览器仍然可以正确读取。

因此如果不设置 content-type 头部，都交给浏览器来处理似乎是可行的。但这边的设置涉及到编码问题，故程序在解析 html 文件时选择了传输 content-type 字段，我们在问题的第三大点中将会讨论编码问题

2.2 Content-Length 与 keep-Alive

这两个字段其实都是与长连接相关的，首先 keep-Alive 在 HTTP /1.1 中是默认的，没有设置为 connection:close 默认就都是 keep-Alive，不需要我们在 header 中显式指出。但是头部中关于最大连接数和超时的设置似乎没有起作用，这里的解释似乎只能是浏览进行了一些处理，并不允许 server 随意的设置。

```
response += "Keep-Alive: max=5, timeout=1\r\n"
```

对于长连接其实就是服务端在发送完数据后不立刻关闭，而是等待客户端发起挥手报文，服务端收到 FIN 报文后再关闭。Content-Length 字段起的作用就是客户端知道自己接收到多少字节的数据后就可以开始进行四次挥手。对于短连接，这个字段其实是可有可无的，因为服务端在发送完数据后关闭了连接，那么客户端也接受完数据后也就可以相应的关闭连接。不需要 Content-Length 字段来标识结束（实际上是通过最后的 EOF 标识）。

而对于长连接，由客户端首先发起 FIN，就需要知道自己何时已经接收完成，就需要 content-length 字段。服务端接收到 FIN 报文，即 request 为空，就可以跳出长连接的 while 循环。对于单独的图片测试后发现长连接确实可以正常使用。

但是项目中使用长连接时遇到了一些问题，也就是客户端始终在转圈加载，也就是始终不发送 FIN 报文，连接一直保持建立。

```
895 GET /images/trolltech-logo.png HTTP/1.1
 44 cbt(7777) → 57456 [ACK] Seq=724 Ack=1696 Win=2618880 Len=0
 45 [TCP Keep-Alive] 57456 → cbt(7777) [ACK] Seq=1695 Ack=724 Win=
 56 [TCP Keep-Alive ACK] cbt(7777) → 57456 [ACK] Seq=724 Ack=1696
 45 [TCP Keep-Alive] 57456 → cbt(7777) [ACK] Seq=1695 Ack=724 Win=
 56 [TCP Keep-Alive ACK] cbt(7777) → 57456 [ACK] Seq=724 Ack=1696
 45 [TCP Keep-Alive] 57456 → cbt(7777) [ACK] Seq=1695 Ack=724 Win=
```

那很自然的猜想就是 content-length 字段计算多了，导致客户端以为没有接收完成。查阅相关资料后发现，如果 content-length 字段出现错误，可以采用 chunked 这种分块传输策略。不依赖头部的长度信息，也能知道实体的边界也即分块编码。

```
response += "Transfer-Encoding: gzip, chunked\r\n"
```

但是程序加上这个首部后反而 html 文件连页面内的 js css jpg 文件都无法解析了。查阅了很多资料后仍然是单独的图片或 html 文件可以进行长连接，若 html 中包含了其余资源请求就无法成功。所以暂时搁置了这个长连接的问题，先采用的短连接的方式。在完成报告写道关于 content-tyoe 字段的过程中，我有了一个猜想到可能是浏览器解析文件类型的原因，程序中对静态资源的文件其实没有详细分解处理，而是都交由浏览器自行处理，这中间可能使得计算文件长度时出现了一些问题。但限于期末的时间限制，还没有测试完全，所以上传的项目仍然采用的是短连接的版本。

3、关于编码问题

Windows 下如果不进行一些设置其实默认采用的是 GBK 编码，所以一开始想当然的以为所有地方都采用 GBK 编码就可以了，当然这肯定是错误的。

Windows 下的 web 页面默认采用 GBK 编码的意思是如果 Html 页面没有进行 charset 的设置，默认采用的 GBK 编码。这就对应到了上文 content-type 字段的解释。

```
set_response_header("200 OK", [('Content-Type', 'text/html;charset=utf8')])
```

如果设置了 utf-8 编码，那么服务端在发送时就要 encode 为 utf8 发送。这里不采用 GBK 的原因也是因为 utf-8 编码覆盖的字符其实更多，有时采用 GBK 会出现以下错误，用一个更宽的字符集就可以解决问题。

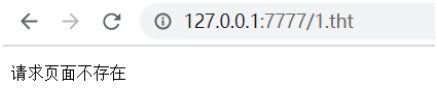
```
'gbk' codec can't decode byte 0x8e in position 4231: illegal multibyte sequence
```

关于编码可以找一下对应关系，在 deal_dynaminc 中，我们因为对 html 页面进行了 charset 设置，故发送时编码为 utf-8

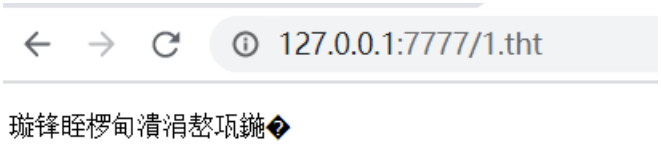
```
connection_socket.send(response.encode("utf-8"))
```

在 deal_static 中由于未对 html 页面进行设置，所以默认为 gbk 编码，发送时编码同样需要采用 GBK

```
response_body = "请求页面不存在".encode("GBK")
```



若这里采用 utf-8，那么浏览器中文就会出现乱码



但其实对于英文字符，使用 GBK 和 utf-8 编码的差别其实不大，只是在于 utf-8 能够处理更多的字符。总的来说只要注意服务器发送内容的编码和浏览器 charset 设置的编码相对应即可。

四、 后续可优化

这次项目基本大致疏通了一遍了从 webserver 与前端交互的过程，也完成了一个很简易的框架来处理前端的请求，但其中也有很多不足的地方，比如前端的页面登陆权限的设置问题，某些功能使用弹窗更好，用户信息提交应该使用 post 等等。后端也只处理了 get 请求，对 post 请求的处理还没有实现。另外对于 http 头部字段的设置其实还有很多信息可

以挖掘，每个 header 字段其实都有其作用，长连接、分块编码、字段长度、最大连接数、超时时间等等都值得细细研究。另外还有一点就是关于 socket 连接的处理，这里就和操作系统连接很紧密，线程、进程、epoll 有很多种方案可以考虑。总的来说确实还有很多可以完善的地方。