Socket 应用编程实验

一、 实验内容

实现：使用 C 语言实现最简单的 HTTP 服务器

1、 同时支持 HTTP（80 端口）和 HTTPS（443 端口），使用两个线程分别监听各自端口

2、只需支持 GET 方法，解析请求报文，返回相应应答及内容。其中应答具体要求如下。

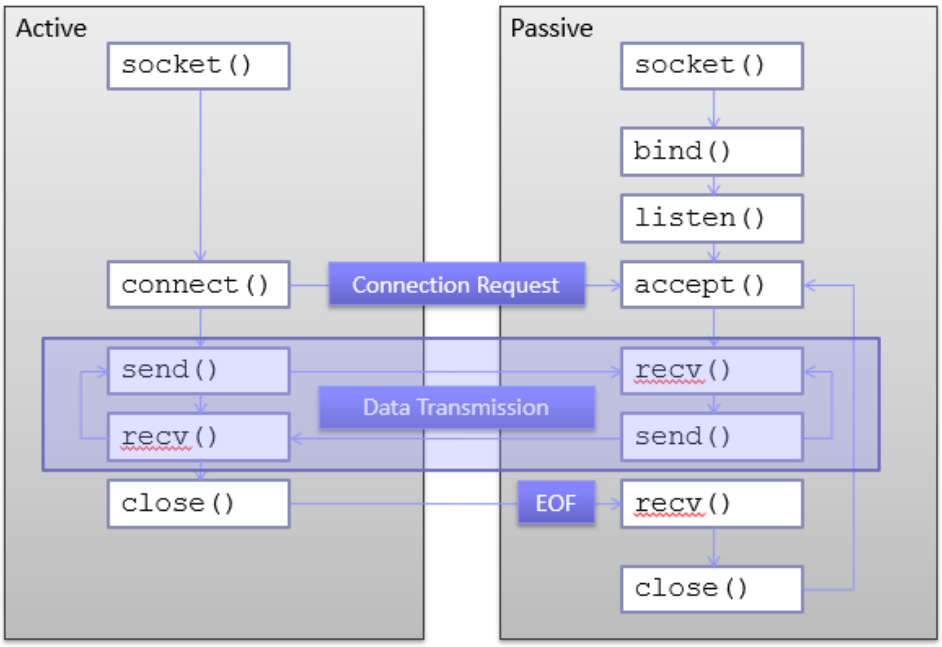
需支持的状态码	场景
200 OK	对于443端口接收的请求，如果程序所在文件夹存在所请求的文件，返回该状态码，以及所请求的文件
301 Moved Permanently	对于80端口接收的请求，返回该状态码，在应答中使用Location字段表达相应的https URL
206 Partial Content	对于443端口接收的请求，如果所请求的为部分内容（请求中有Range字段），返回该状态码，以及相应的部分内容
404 Not Found	对于443端口接收的请求，如果程序所在文件夹没有所请求的文件，返回该状态码

图表 1：状态码

二、 设计思路

1、 提供的实验代码实现了一个基本的 HTTPS 服务器，该代码实现 443 端口和部分 200 状态码的功能实现。

2、 基于实验内容，要在现有代码的框架实现双线程，增加 80 端口和返回 301 状态码的代码，在 443 端口增加支持 206 和 404 状态码返回的代码。



图表 2：客户端应答流程

整体 HTTP 和 HTTPS 服务器设计思路如上图所示，我们只需完成接收和返

回部分。

3、具体实现

（1）双线程

在 main 函数中创建两个线程，其中创建 thread_1 进程用于监听 HTTP 端口，主线程监听 HTTPS 端口。两个线程都使用 listen_port 函数进行监听。最后 pthread_detach 函数在第一个线程结束后自动释放资源，

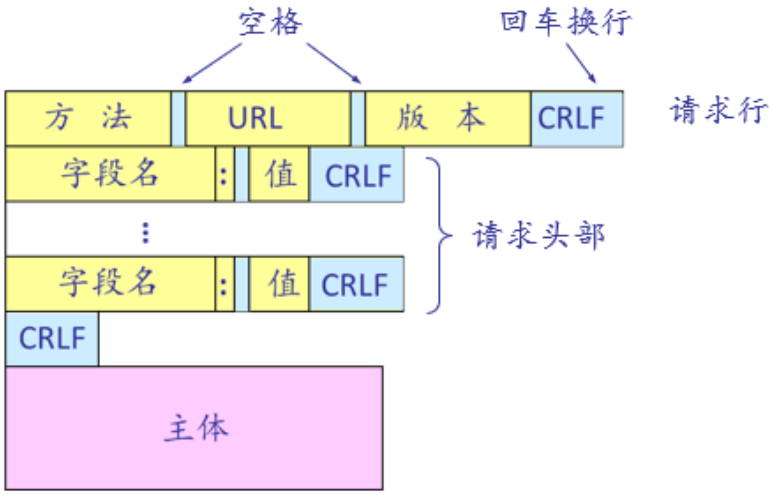
```
int main(){
    pthread_t thread_1;
    int http_port = HTTP;
    int https_port = HTTPS;

    if(pthread_create(&thread_1,NULL,listen_port,&http_port) != 0){
        perror("Create thread failed");
        exit(1);
    }
    listen_port(&https_port);
    pthread_detach(thread_1);
}
```

（2）配置服务器监听的地址和端口

这部分与提供代码 main 函数部分大致相同，不做赘述。

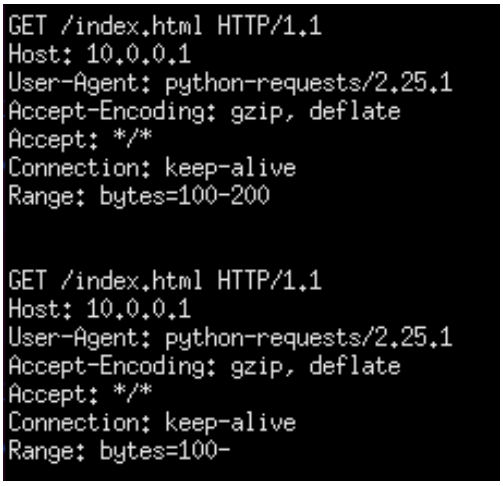
（3）request 解析

这一部分使用 reques_decode 函数对 request 进行解析，将函数分为 method，url，version，头部和主体几个部分，这样在后续处理中针对状态码可以直接应用，比较方便。函数中主要使用指针作为标志位，标志处理的位置。request 大致如下图所示。



图表 3：请求样式

在实际中，请求如下图所示



图表 4：实际请求样式

```
struct Request* request_decode(const char* raw_request)
```

raw_request 指向 request 中未被处理的第一位。

首先，定义一个指向 request 的结构体指针 req，在该函数中会将该结构体中所有变量赋值。

```
struct Request *req = NULL;
typedef struct Request {
    enum Method method;
    char *url;
    char *version;
    struct Header *headers;
    char *body;
} Request;
```

其次，判断是否为 GET 方法，使用 strcspn 判断长度，memcpy 函数进行类型判断

```
size_t method_len = strcspn(raw_request, " ");
    if (memcmp(raw_request, "GET", strlen("GET")) == 0) {
        req -> method = GET;
    }
    else {
        req -> method = UNSUPPORTED;
    }
    raw_request += (method_len + 1);
```

url 和 version 部分同样使用 strcspn 函数，同时使用 memcpy 函数将需要的部分放进 req 结构体中，然后移动 raw_request 指针。

```
 size_t url_len = strcspn(raw_request, " ");
    req -> url = malloc(url_len + 1);
    memcpy(req -> url, raw_request, url_len);
    req -> url[url_len] = '\0';
    raw_request += url_len + 1; // move past <SP>
```

而后处理 header 部分，这其中对于状态码有用的部分是 host 和 range 字段，但在这里我以“：“为标志位，将所有 name 和后面的内容拆分出来用链表连接。如下

图为一个链表，识别和赋值方法与 url 类似。

```
typedef struct Header {
    char *name;
    char *value;
    struct Header *next;
} Header;
```

之后处理 body 部分，将所有剩余字符赋值给 req->body。

```
size_t body_len = strlen(raw_request);
    req -> body = malloc(body_len + 1);
    memcpy(req -> body, raw_request, body_len);
    req -> body[body_len] = '\0';
```

（4）HTTP 端口处理

```
     struct Request* req = request_decode(request_buf);

    // 301
    code = MOVED_PERMANTLY;
    char new_url[100];
    strcpy(new_url, "https://10.0.0.1");
    strcat(new_url, req->url);
    response_len += sprintf(response_buf, "%s %d MOVED PERMANENTLY\r\n",
req->version, code);
    response_len += sprintf(response_buf + response_len, "Location: %s\r\n",
new_url);
```

从 request_decode 函数中获得分解后的请求结构体后，将状态码 301 需要的部分放进 response_buf 即可。返回内容如下图所示。

```
HTTP/1.1 301 MOVED PERMANENTLY
Location: https://10.0.0.1/index.html
```

图表 5：应当输出

（5）HTTPS 端口处理

由于 HTTPS 有三个状态码需要处理，因此需要进行判断。判断依次为是否打开文件成功和是否返回部分文件。

如果状态码为 404 的话，打开文件失败，将版本和状态码放入应答缓冲区即可。

```
if ((fp = fopen(refined_url, "r")) == NULL) {
        code = NOT_FOUND;
        response_len += sprintf(response_buf, "%s %d NOT FOUND\r\n", req ->
version, code);
        response_len += sprintf(response_buf + response_len, "\r\n");
    }
```

而后进行状态码 206 和状态码 200 的判断，在头部查找 range 字段，如果找到则设置 partial 为一，即状态码为 206。使用 sscanf 函数对 start 和 end 进行赋值，如果 range 字段形如“100-”则 end 会赋值为 file_len-1，即文件末尾

```
            int start = 0;
        int end = file_len - 1;
        char str_range[15];
        strcpy(str_range, "Range");
        for (h = req -> headers; h; h = h -> next) {
            if (strcmp(h -> name, str_range) == 0) {
                partial = 1;
                sscanf(h -> value, "bytes=%d-%d", &start, &end);
                end += 1;
                break;
            }
        }
```

而之后对于状态码 206 和 200 的处理类似，这里只叙述我是如何处理 206 的。先使用 fseek 将文件指针移动到请求开始的位置，而后在应答缓冲区中放入版本、状态码和内容长度。使用 fread 读取需要的文件内容，放入缓冲区中即可。

```
                code = PARTIAL_CONTENT;
                int read_len = end - start;
                fseek(fp, start, SEEK_SET);
                response_len += sprintf(response_buf, "%s %d PARTIAL
CONTENT\r\n", req -> version, code);
                response_len += sprintf(response_buf + response_len, "Content-
length: %d\r\n", read_len);
                response_len += sprintf(response_buf + response_len, "\r\n");
                fread(response_buf + response_len, sizeof(char), read_len, fp);
                fseek(fp, 0, SEEK_SET);
                response_len += read_len;
```

图表 6：只输出前一百个字符

三、 遇到的问题

1、 request_decode 中解析请求

这个函数使用 raw_request 指针进行请求的处理，但是如何移动使其准确指向未被处理的第一个字符是一个比较大的问题。

h2 窗口中出现了报错。

图表 6：h2 窗口报错

在 h1 窗口中查看所有请求信息，下图状态码的应带中 url 出现问题且状态码出现错位，考虑请求的请求行出现问题，进行检查后，发现 raw_request 处理版本移位出现错误，由于版本后接“\r\n”，因此不能像 url 部分一样长度

+1（raw_request += url_len + 1），而是应该 raw_request += version_len + 2，改正后输出正常。

```
HTTP/1.1 301 MOVED PERMANENTLY
Location: https://10.0.0.1/notfound.html

HTTP/1.1 404 NOT FOUND


This code finish
```

图表 7：h1 窗口错误输出

四、 测试

```
root@leona-virtual-machine:/home/leona/CN/03-socket# ./http-server
bind success port443
bind success port80
connect success port80
socket request receive
socket responese send
connect success port443
ssl request receive
ssl response sent
connect success port80
socket request receive
socket responese send
connect success port443
ssl request receive
ssl response sent
connect success port80
socket request receive
socket responese send
connect success port443
ssl request receive
ssl response sent
connect success port80
socket request receive
socket responese send
connect success port443
ssl request receive
ssl response sent
connect success port80
socket request receive
socket responese send
connect success port443
ssl request receive
ssl response sent
connect success port80
socket request receive
socket responese send
connect success port443
ssl request receive
ssl response sent
```

图表 7：将请求和应答均打印出来的结果

图表 8：将 80 端口和 443 端口应答均打印