

HTTP服务器实验

实现： 使用C语言实现最简单的HTTP服务器

要求： 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

3. 支持的状态码

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

实验流程：

1. 根据上述要求，实现HTTP服务器程序
2. 执行sudo python topo.py命令，生成包括两个端节点的网络拓扑
3. 在主机h1上运行HTTP服务器程序，同时监听80和443端口
h1 # ./http-server
4. 在主机h2上运行测试程序，验证程序正确性
h2 # python3 test/test.py
如果没有出现AssertionError或其他错误，则说明程序实现正确
5. 在主机h1上运行http-server，所在目录下有一个小视频（30秒左右）
6. 在主机h2上运行vlc（注意切换成普通用户），通过网络获取并播放该小视频
媒体 -> 打开网络串流 -> 网络 -> 请输入网络URL -> 播放

socket编程基础

名词解析：socket有“插座”的意思，在linux环境下socket是进程间网络通信的特殊文件类型。编程中一般使用文件描述符fd来引用套接字，fd是一个int类型的文件描述符。简单解释下socket如何使得两个进程相互通信。

socket端：

1. 使用sockaddr_in结构创建服务器地址server_addr，对地址进行初始化
2. 设置地址的协议族，IP号、端口号
3. 创建一个socket，并返回到sfd，sfd此时能引用这个server socket
4. 将server_addr和服务端文件描述符(sfd)绑定起来
5. 使用listen函数对sfd进行监听，也就是对指定的IP和端口号进行监听
6. 阻塞等待接收客户端的请求（客户端只有请求正确的IP和端口才能正常连接服务器，所以客户端会有一个和服务器端一样的server_addr）
7. 等到client拿自己的cfd(客户端文件描述符)去连接server，server会新建一个socket
，这个socket指向connfd, server可以read这个connfd的数据，然后再将信息write到这个connfd上，connfd和cfd实际上指向的是同一个socket，客户端能够从这个cfd上读取server写进去的数据。这样就完成了c/s的通信。
8. 关闭socket连接

client端：

1. 创建socket，返回到cfd
2. 使用sockaddr_in结构创建服务器地址server_addr，对地址进行初始化。并将成员值设置为和上面server_addr一样
3. 使用cfd和server_addr作为参数去连接指定的server端口。
4. 连接成功后，client能向cfd指向的socket写数据，server建立连接后会创建新的socket，这个socket实际和cfd指向的是socket是一致的。client能够向这个socket写入数据和读取数据。
5. 关闭socket连接

通过socket实现能传输简单信息的http服务器

server端

```
int main() {  
  
    struct sockaddr_in server_addr, client_addr;  
  
    int listenfd, connfd;    //监听套接字和连接套接字  
  
    char buffer[MAXLINE], first_line[MAXLINE], left_line[MAXLINE],  
method[MAXLINE], url[MAXLINE], version[MAXLINE];  
  
    char str[INET_ADDRSTRLEN];
```

```
socklen_t client_addr_len;

char filename[MAXLINE];

long n;

int i, pid;


listenfd = socket(AF_INET, SOCK_STREAM, 0);           //创建套接字

bzero(&server_addr, sizeof(server_addr));           //初始化server_addr

server_addr.sin_family = AF_INET;                   //设置协议族(IPV4)

server_addr.sin_addr.s_addr = htonl(INADDR_ANY);    //具体IP地址

server_addr.sin_port = htons(PORT1);                //设置端口号


printf("threadID: %d, server_addr: %s, port: %d", pthread_self(),
inet_ntop(AF_INET, &server_addr.sin_addr, str, sizeof(str)),
ntohs(server_addr.sin_port));


int bind_status = bind(listenfd, (struct sockaddr*)&server_addr,
sizeof(server_addr));    //绑定套接字

if(bind_status < 0)    printf("bind error!\n");


int listen_status = listen(listenfd, 20);    //监听套接字, 等待用户发
起请求

if(listen_status < 0) printf("listen error!\n");

printf("Accepting connections ...");

while(1) {

    //阻塞等待接受客户端的请求

    client_addr_len = sizeof(client_addr);
```

```

        connfd = accept(listenfd, (struct sockaddr*)&client_addr,
&client_addr_len);    //接受客户端的请求

        n = read(connfd, buffer, MAXLINE);

        if (n == 0) {

            printf("client has been closed");

            break;

        }

        printf("port %d Received from %s at PORT %d, message is %s\n",

            ntohs(server_addr.sin_port),

            inet_ntop(AF_INET, &client_addr.sin_addr, str,
sizeof(str)),

            ntohs(client_addr.sin_port), buffer);

        //解析请求报文

        for (int i = 0; i < n; i++) {

            buffer[i] = toupper(buffer[i]);

        }

        write(connfd, buffer, n);

        close(connfd);

    }

}

```

为了方便测试，在topo.py中将两个host改为3个host，并设置连接。修改如下

```

class MyTopo(Topo):

    def build(self):

        h1 = self.addHost('h1')

        h2 = self.addHost('h2')

        h3 = self.addHost('h3')

        s1 = self.addSwitch('s1')


self.addLink(h1, s1, bw=100, delay='10ms')

self.addLink(h2, s1)

self.addLink(h3, s1)

```

在h1中运行./server，在h2和h3中分别运行./client1和./client2。client1和client2的区别是请求的端口不同，但主机IP是一样的，能够用来测试server是否能同时检测两个端口。结果展示如下

```

root@node2:/home/linux4/Desktop/webserver# ./server
root@node2:/home/linux4/Desktop/webserver# ./client
write to server : hello, I am client 1
20
Response from server: HELLO, I AM CLIENT 1V
HELLO, I AM CLIENT 1
root@node2:/home/linux4/Desktop/webserver#

root@node2:/home/linux4/Desktop/webserver# ./server_p
root@node2:/home/linux4/Desktop/webserver# gcc server_p.c -o server_p
root@node2:/home/linux4/Desktop/webserver# ./server_p
root@node2:/home/linux4/Desktop/webserver# gcc server_p.c -o server_p
root@node2:/home/linux4/Desktop/webserver# ./server_p
received from 10.0.0.2 at PORT 5770,message is hello world
root@node2:/home/linux4/Desktop/webserver# ./server_p
received from 10.0.0.2 at PORT 4502,message is hello world
received from 10.0.0.3 at PORT 4636,message is hello world
received from 10.0.0.2 at PORT 44616,message is hello world
root@node2:/home/linux4/Desktop/webserver# ./server_p
root@node2:/home/linux4/Desktop/webserver# ./server_p
received from 10.0.0.2 at PORT 47976,message is hello, I am client 1
received from 10.0.0.3 at PORT 58344,message is hello I am client 2
root@node2:/home/linux4/Desktop/webserver#

linux4@node2: ~/Desktop/webserver
linux4@node2: ~/Desktop/all-exps/05-http_server
linux4@node2:~$ cd Desktop/all-exps/05-http_server/
linux4@node2:~/Desktop/all-exps/05-http_server$ ls
client client.c client2.c server_p.c webserver webserver.c
linux4@node2:~/Desktop/all-exps/05-http_server$ dir httpd httpd.c httpserver index.html keys Makefile server test topo.py webserver
[sudo] password for linux4:
mininet> xterm h1 h2
mininet> xterm h3
mininet>

```

client1.c如下所示

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//read方法需要的头文件
#include <unistd.h>

```

```
//socket方法需要的头文件
#include <sys/socket.h>
#include <sys/types.h>
//htonl 方法需要的头文件
#include <netinet/in.h>
//inet_ntop方法需要的头文件
#include <arpa/inet.h>

#define MAXLINE 100
#define CLI_PORT 80
//webserver 主程序

int main(int argc, const char * argv[]) {
    struct sockaddr_in servaddr;
    char buf[MAXLINE];

    int clientfd;
    long n;
    //client socket连接
    clientfd = socket(AF_INET, SOCK_STREAM, 0);
    char *str = "hello world";

    //sockaddr_in结构体初始化
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(CLI_PORT);

    //connect()方法
    connect(clientfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    //write()方法是client 向 server 写数据
    write(clientfd, str, strlen(str));
    //最好用strlen, 否则server接收的数据会少
    printf("write to server : %s\n",str);

    //read()方法是从server接收数据
    n = read(clientfd, buf, sizeof(buf));
    //注意最好用sizeof而不是strlen。实验证明用strlen时接受会出错。bug调试很久才找到
    printf("%ld\n",n);
    if(n == 0) {
        printf("the other side has been close\n");
    }else {
```

```

        printf("Response from server: %s\n", buf);
        write(STDOUT_FILENO, buf, n);
        printf("\n");
    }
    close(clientfd);
}

```

现在客户端能和服务器进行简单的通信，下一步就是要支持稍微复杂些的通信。

支持并发的http服务器

1. 首先将socket函数封装为带有容错机制的函数，运行过程中出错的话可以更好的定位错误。

```

int Socket(int family, int type, int protocol) {
    int sockfd;
    if((sockfd = socket(family, type, protocol)) < 0)
    {
        perror("socket error");
        exit(1);
    }
    return sockfd;
}

void Bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen) {
    if(bind(sockfd, addr, addrlen) < 0)
    {
        perror("bind error");
        exit(1);
    }
}

void Listen(int sockfd, int backlog) {
    if(listen(sockfd, backlog) < 0)
    {
        perror("listen error");
        exit(1);
    }
}

int Accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen) {
    int connfd;
    if((connfd = accept(sockfd, addr, addrlen)) < 0)

```

```

    {
        perror("accept error");
        exit(1);
    }
    return connfd;
}

void Connect(int sockfd, const struct sockaddr *addr, socklen_t
addrlen) {
    if(connect(sockfd, addr, addrlen) < 0)
    {
        perror("connect error");
        exit(1);
    }
}

long Read(int fd, void *buf, size_t count) {
    long n;
    if((n = read(fd, buf, count)) < 0)
    {
        perror("read error");
        exit(1);
    }
    return n;
}

void Write(int fd, void *buf, size_t count) {
    if(write(fd, buf, count) < 0)
    {
        perror("write error");
        exit(1);
    }
}

void Close(int fd) {
    if(close(fd) < 0)
    {
        perror("close error");
        exit(1);
    }
}

```


2. 使服务器带有并发能力，也就是http server接收到一个请求就会fork()一个进程去处理请求。

下面是部分逻辑代码：

当server Accept一个请求时，fork一个进程，如果是子进程，则对该请求处理。如果没有正常fork，pid还是原来大于0的父进程，则直接关闭socket。

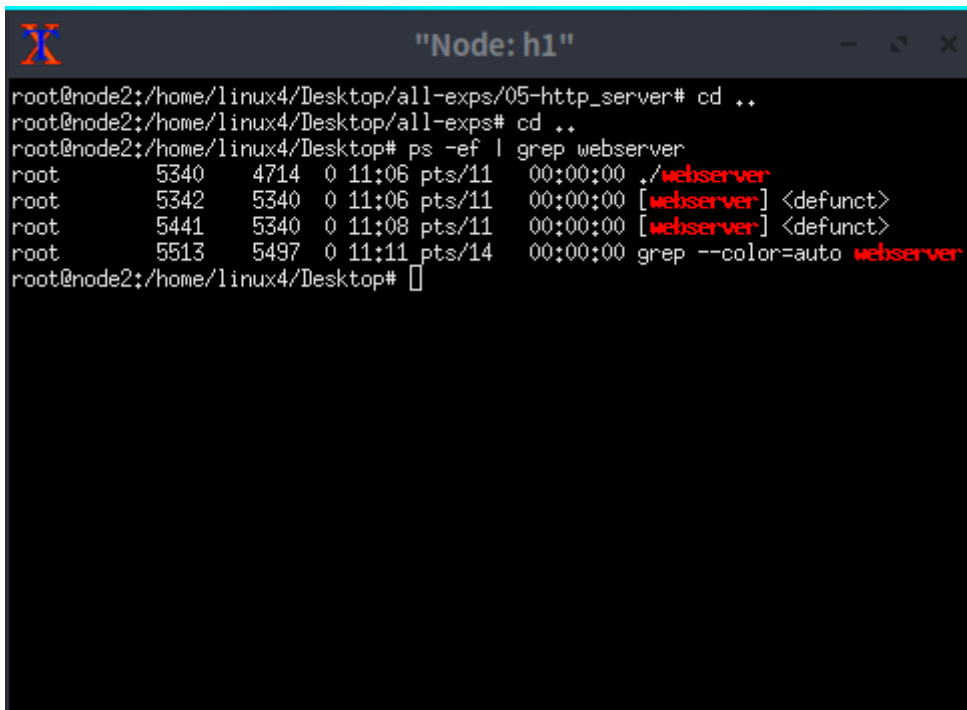
```
//死循环中进行accept()
while (1) {
    cliaddr_len = sizeof(cliaddr);

    //accept()函数返回一个connfd描述符
    connfd = Accept(listenfd, (struct sockaddr *)&cliaddr,
                    &cliaddr_len);

    pid = fork();
    if(pid < 0) {
        printf("fork error");
        return 1;
    } else if(pid == 0) {    //pid=0表示子进程
        while (1) {
            n = Read(connfd, buf, MAXLINE);
            if (n == 0) {
                printf("the other side has been closed.\n");
                break;
            }
            printf("received from %s at PORT %d,message is %s,\n",
                    inet_ntop(AF_INET, &cliaddr.sin_addr, str,
                               sizeof(str)),
                    ntohs(cliaddr.sin_port),buf,
                    n);

            for (int i = 0; i < n; i++)
                buf[i] = toupper(buf[i]);
            Write(connfd, buf, n);
        }
        Close(connfd);
        exit(0);
    } else {    //pid>0表示父进程
        Close(connfd);
    }
}
```

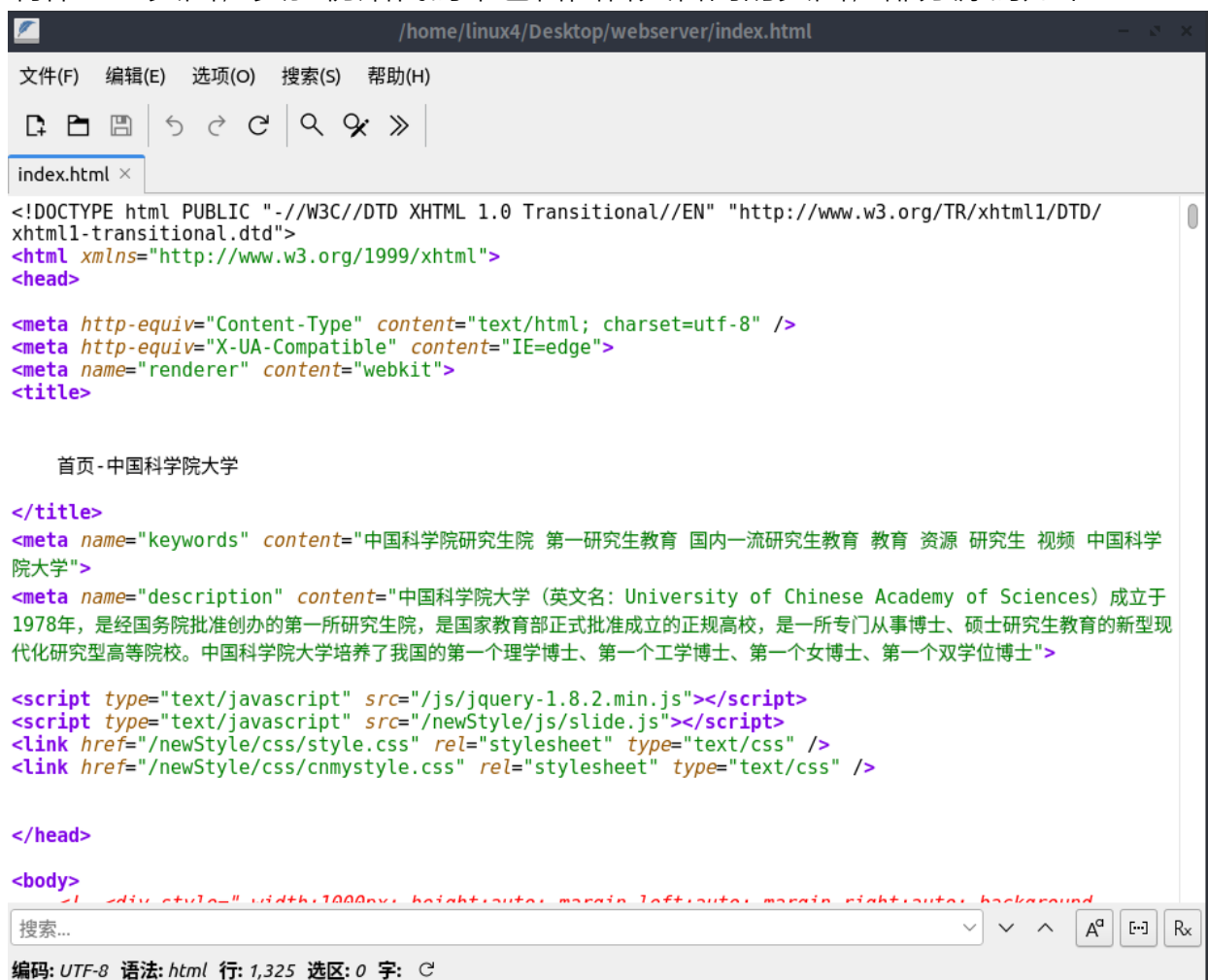
从下图可以看出，使用了多进程并发机制后，当有多个请求时，会fork子进程，服务器端的进程对应也在增加。



```
"Node: h1"
root@node2:/home/linux4/Desktop/all-exps/05-http_server# cd ..
root@node2:/home/linux4/Desktop/all-exps# cd ..
root@node2:/home/linux4/Desktop# ps -ef | grep webserver
root      5340      4714  0 11:06 pts/11    00:00:00 ./webserver
root      5342      5340  0 11:06 pts/11    00:00:00 [webserver] <defunct>
root      5441      5340  0 11:08 pts/11    00:00:00 [webserver] <defunct>
root      5513      5497  0 11:11 pts/14    00:00:00 grep --color=auto webserver
root@node2:/home/linux4/Desktop#
```

支持html页面的http server

1. 制作html页面，实验初始代码中包含国科大官网的页面，部分源码如下：



```
/home/linux4/Desktop/webserver/index.html
文件(F) 编辑(E) 选项(O) 搜索(S) 帮助(H)
index.html x
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="renderer" content="webkit">
<title>

    首页 - 中国科学院大学

</title>
<meta name="keywords" content="中国科学院研究生院 第一研究生教育 国内一流研究生教育 教育 资源 研究生 视频 中国科学
院大学">
<meta name="description" content="中国科学院大学（英文名：University of Chinese Academy of Sciences）成立于
1978年，是经国务院批准创办的第一所研究生院，是国家教育部正式批准成立的正规高校，是一所专门从事博士、硕士研究生教育的新型现
代化研究型高等院校。中国科学院大学培养了我国的第一个理学博士、第一个工学博士、第一个女博士、第一个双学位博士">

<script type="text/javascript" src="/js/jquery-1.8.2.min.js"></script>
<script type="text/javascript" src="/newStyle/js/slide.js"></script>
<link href="/newStyle/css/style.css" rel="stylesheet" type="text/css" />
<link href="/newStyle/css/cnmystyle.css" rel="stylesheet" type="text/css" />

</head>
<body>
<div style="width:100%; height:auto; margin-left:auto; margin-right:auto; background
```

2. 打开服务器，使用firefox输入 `http://localhost:80/index.html` ,查看web发到服务器中的内容。

整个头部信息如下所示，在代码中被存在buf中。**注意我们在read() socket时，是要将内容放到buf中，可以设置buf的大小，如果buf设置太小，则需要循环read，直到connfd中的内容被读空

```
o linux4@node2:~/Desktop/webserver$ sudo ./webserver
[sudo] password for linux4:
GET /index.html HTTP/1.1
Host: 192.168.254.135
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/105.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8
Cookie: Hm_lvt_01a7c3bef12e65d63268f93ef38b1dbe=1665326418

h;q=0.9,en;q=0.8
Cookie: Hm_lvt_01a7c3bef12e65d63268f93ef38b1dbe=1665326418the other side has been closed.
□
```

使用下面代码来解析请求

```
sscanf(buf, "%s %s %s", method, uri, version); //对buf进行解析
printf("method:%s\n", method);
printf("uri:%s\n", uri);
printf("version:%s\n", version);
```

3. 根据请求中的url来查找服务器本地文件。如果在文件夹下找到文件名，则获取文件类型，拼接response返回给服务器终端。另外需要读取文件信息并将其输入给浏览器上，主要的函数是mmap(0, sbuf.st_size, PROT_READ, MAP_PRIVATE, fd, 0);

响应函数response如下所示：

```
void response(int connfd, char *filename) {
    struct stat sbuf; //文件状态结构体
    int fd;
    char *srcp;
    char response[MAXLINE], filetype[20];

    if (stat(filename, &sbuf) < 0) {
        //文件不存在
        sprintf(response, "HTTP/1.1 404 Not Found\r");
        exit(1);
    }

    else {
        get_filetype(filename, filetype); //获取文件类型
```

```

//Open File
fd = open(filename, O_RDONLY);

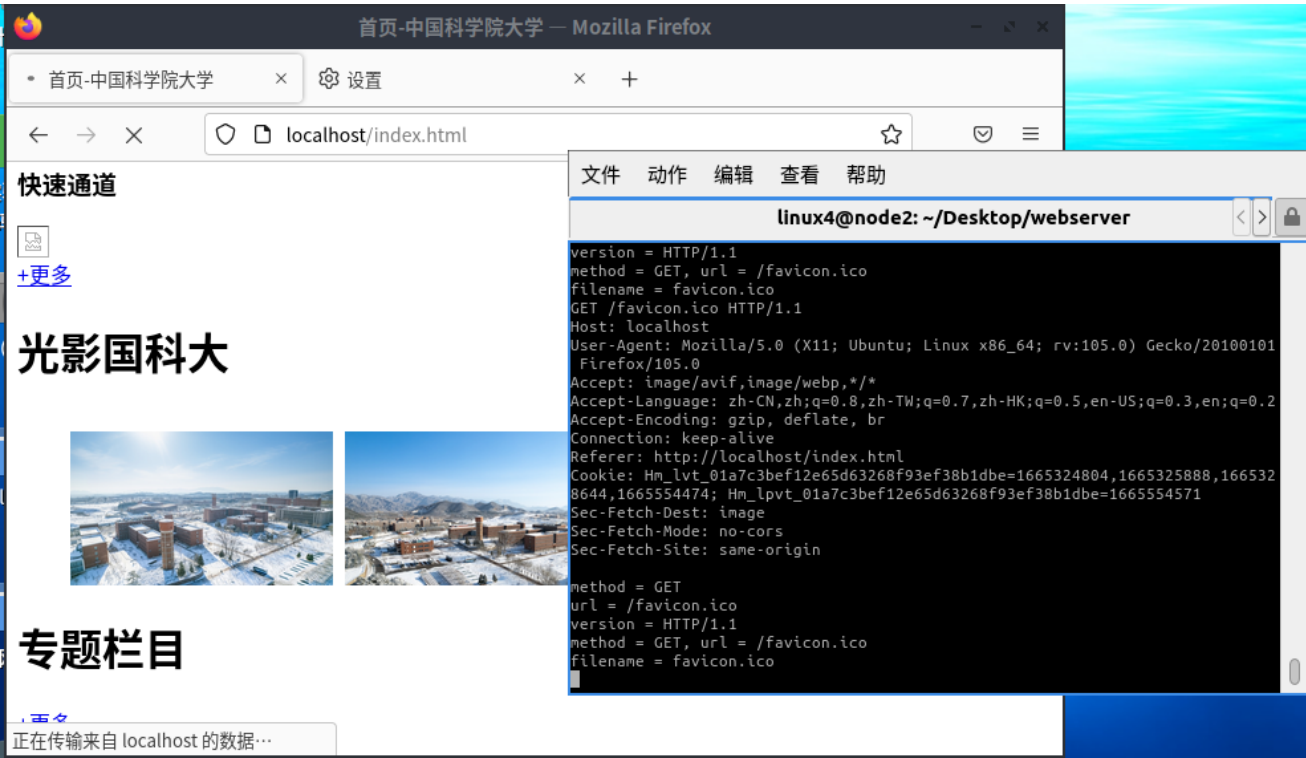
//Send response 这是是在进行拼接，一定要返回头部，
//客户端会先识别头部然后对数据部分进行个性化解析
strcat(response, "HTTP/1.1 200 OK\r\n");
strcat(response, "Server: LongXing's Tiny Web Server\r\n");
sprintf(response, "Content-length: %ld\r\n", sbuf.st_size);
sprintf(response, "Content-type: %s\r\n\r\n", filetype);
Write(connfd, response, strlen(response));
printf("Response headers:\n");
printf("%s", response);
//mmap()读取filename 的内容写给浏览器
srcp = mmap(0, sbuf.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
//内存映射，直接将fd指向的文件映射到srcp上，而不先将磁盘上的文件读取
到内核缓冲区，
//再从内核缓冲区将文件进程虚拟地址空间。它映射完了就可以直接使用，只
有一次读取。
Close(fd);
Write(connfd, srcp, sbuf.st_size);
munmap(srcp, sbuf.st_size);
}
}

```

现在可以在本地浏览器中输入请求并得到响应，由于html中的一些图标文件在服务器本地是缺失的，所以有些文件不会显示。

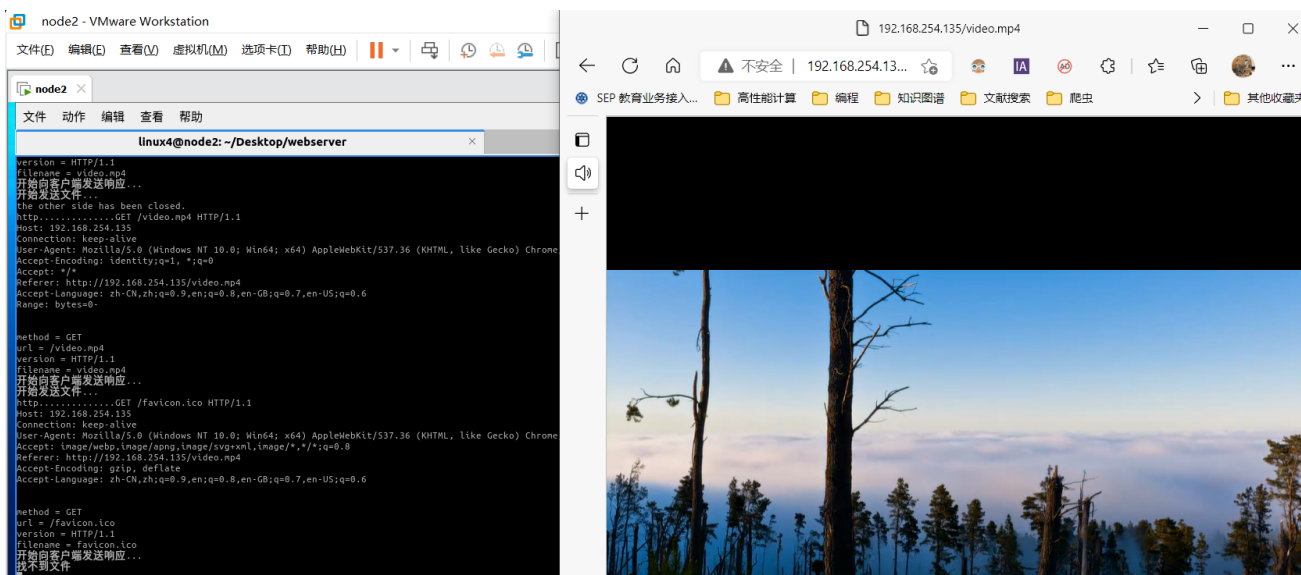
HTTP请求和响应展示

linux虚拟机中HTTP请求index.html



虚拟机外部的主机浏览器HTTP请求index.html





支持HTTPS的服务器

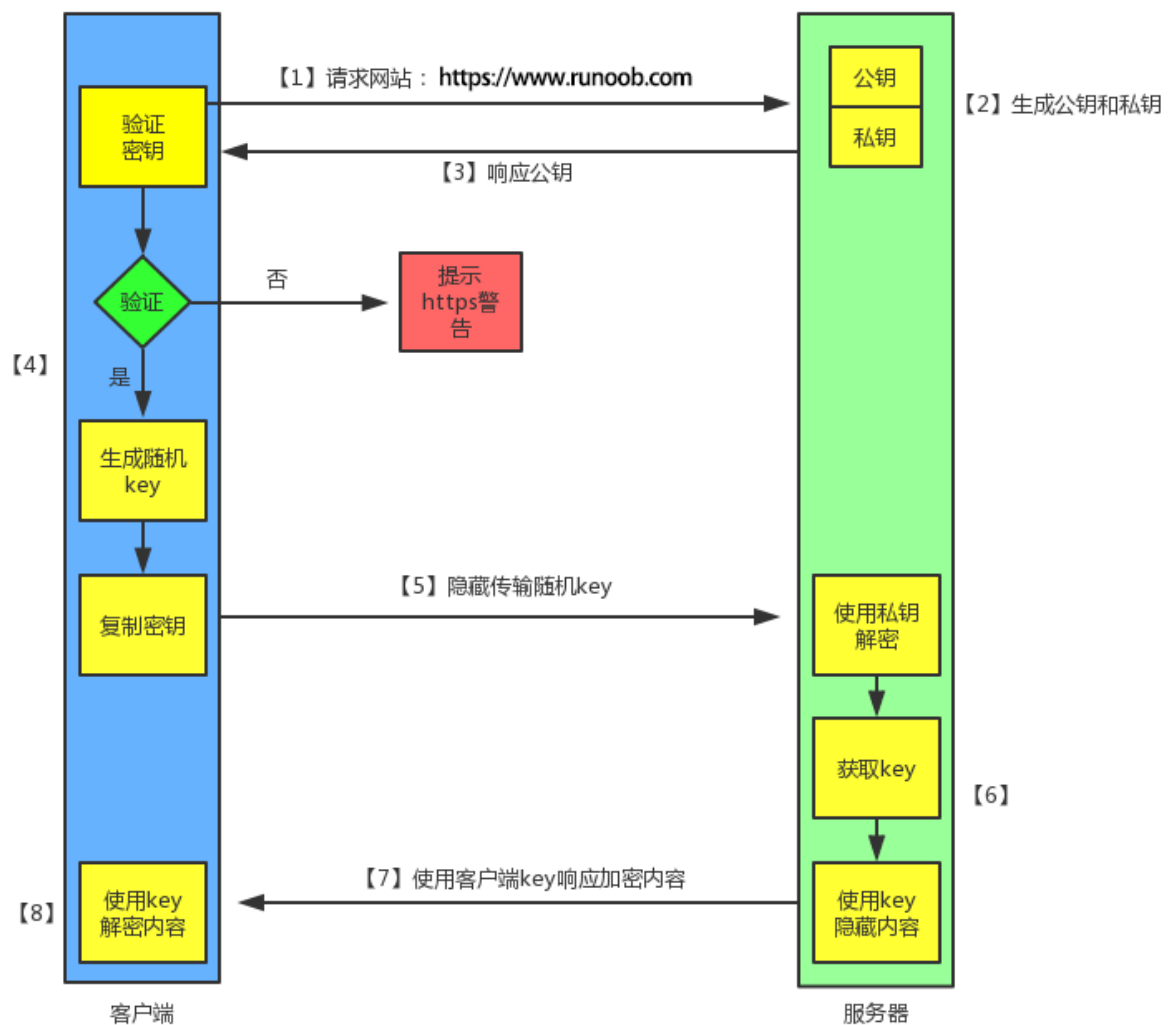
HTTPS基础

目前大部分网站都支持https，这是一种安全的http协议。不同于HTTP的明文传输，HTTPS使用SSL/TSL来加密数据包，一般是使用协商的对称加密算法和密钥加密，保证数据机密性。如果不使用HTTPS，当攻击者截取了Web浏览器和网站服务器之间的传输报文，就可以直接读懂其中的信息，这对大多用户来说都是不愿面对的。

使用 HTTPS 协议需要到 CA（Certificate Authority，数字证书认证机构）申请证书，然后在网页服务器代码中进行SSL编程，如果不使用A证书，只使用自己的用户证书（.cert）和用户私钥（.prokey），安全证书不会通过，导致浏览器连接时需额外确认是否需要不安全连接，在curl时需要加上--insecure参数才能进行请求。

HTTPS建立过程和HTTP不同，后者只需要三次握手就能建立，而HTTPS需要额外九次SSL握手，所以一共是12个包。因此，HTTP的响应速度是要比HTTPS快。另外，HTTP使用80端口，而HTTPS使用443端口。

HTTPS的请求建立过程



HTTPS中SSL部分代码

//封装的部分SSL代码

```
long SSL_Read(SSL *ssl, void *buf, size_t count) {
    long n;
    if((n = SSL_read(ssl, buf, count)) < 0)
    {
        perror("read error");
        exit(1);
    }
    return n;
}
```

```
void SSL_Write(SSL *ssl, void *buf, size_t count) {
    if(SSL_write(ssl, buf, count) < 0)
    {
        perror("write error");
    }
}
```

```
        exit(1);
    }
}
```

//SSL编程流程

//再收到客户端来的https请求，先accept返回connfd，之后再

//1. 初始化SSL

```
SSL_library_init();
OpenSSL_add_all_algorithms();
SSL_load_error_strings();
```

//2. 加载用户证书和私钥以及对其进行检查

SSL_CTX *ctx = SSL_CTX_new(SSLv23_server_method()); //创建服务端SSL会话环境

```
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
```

```
if(SSL_CTX_use_certificate_file(ctx, "cnlab.cert", SSL_FILETYPE_PEM) <= 0) {
```

```
    //加载公钥证书
    printf("load public key error");
    exit(1);
}
```

```
printf("加载私钥...\n");
```

```
if(SSL_CTX_use_PrivateKey_file(ctx, "cnlab.prikey", SSL_FILETYPE_PEM) <= 0) {
```

```
    //加载私钥
    printf("load private key error");
    exit(1);
}
```

```
printf("验证私钥...\n");
```

```
if(SSL_CTX_check_private_key(ctx) <= 0) {
    //检查私钥
    printf("check private key error");
    exit(1);
}
```

//3. 根据ctx创建一个ssl


```

SSL *ssl = SSL_new(ctx);

//4. 将fd与ssl绑定
SSL_set_fd(ssl, fd)

//5.再次进行Accept, 使用SSL_accept(ssl)获取客户端的请求
if(SSL_accept(ssl) == -1) {
    ERR_print_errors_fp(stderr);
}

//进行响应, 使用SSL_Read进行读, 使用SSL_Write进行写操作

```

下面是针对HTTPS的响应函数

```

void https_response(SSL *ssl, int connfd, char *filename) {
    struct stat sbuf;    //文件状态结构体
    int fd;
    char *srcp;
    char response[MAXLINE], filetype[20];

    if (stat(filename, &sbuf) < 0) {
        //文件不存在
        sprintf(response, "HTTP/1.1 404 Not Found\r");
        printf("找不到文件\n");
        exit(1);
    }

    else {
        get_filetype(filename, filetype);    //获取文件类型

        //Open File
        fd = open(filename, O_RDONLY);
        //Send response 这是是在进行拼接
        strcat(response, "HTTP/1.0 200 OK\r\n");
        SSL_Write(ssl, response, strlen(response));

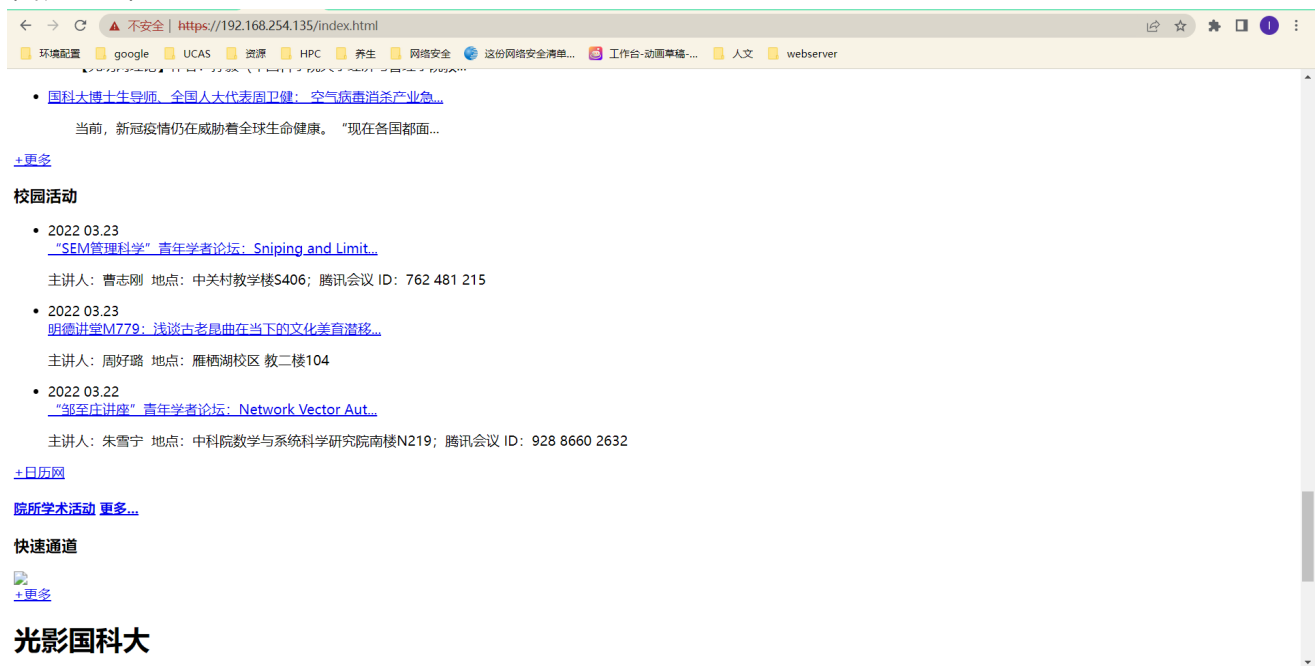
        strcat(response, "Server: LongXing's Tiny Web Server\r\n");
        SSL_Write(ssl, response, strlen(response));
        sprintf(response, "Content-length: %ld\r\n", sbuf.st_size);
        SSL_Write(ssl, response, strlen(response));
        sprintf(response, "Content-type: %s\r\n\r\n", filetype);
        SSL_Write(ssl, response, strlen(response));
        printf("Response headers:\n");
    }
}

```

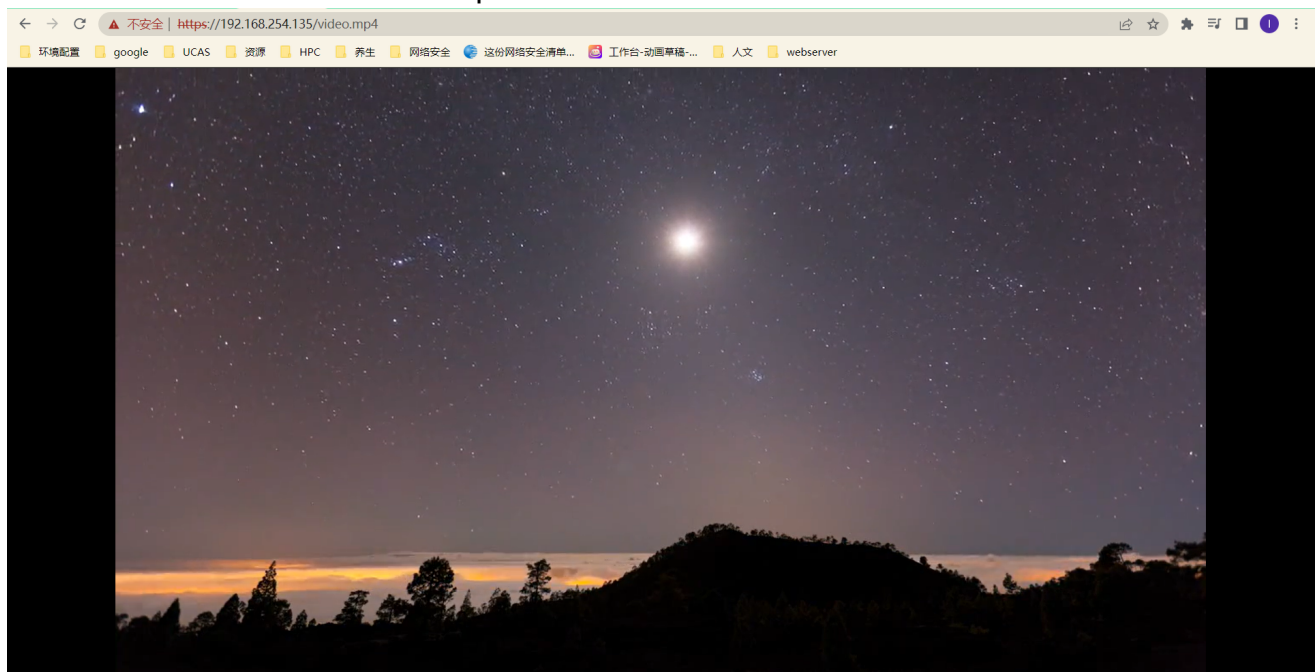
```
printf("%s", response);
//mmap()读取filename 的内容写给浏览器
srcp = mmap(0, sbuf.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
SSL_Write(ssl, srcp, sbuf.st_size);
munmap(srcp, sbuf.st_size);
Close(fd);
}
}
```

HTTPS请求和响应展示

HTTPS在443端口请求index.html，注意有不安全提示是因为使用了自签证书而不是官方证书



HTTPS在443端口请求video.mp4



使用telnet模拟浏览器对index.html请求

```
命令提示符
HTTP/1.0 200 OK
HTTP/1.0 200 OK
Server: LongXing's Tiny Web Server
Content-length: 53464
Content-type: text/html

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="renderer" content="webkit">
<title>
```

同时支持HTTP和HTTPS的服务器

思路：将支持HTTPS的代码从新建socket、绑定、监听、Accept、响应等操作全部封装到https_server函数中。将支持HTTP请求的所有代码封装到http_server函数中。在主函数main中，建立两个子进程，子进程1运行http_server函数，子进程2运行https_server函数。这样在同一个服务器中能有两个socket，一个监听支持HTTP请求的80端口，另外一个监听支持HTTPS请求的443端口。mian函数如下：

```
int main(int argc, char * argv[]) {
    pid_t pid1, pid2;
    pid1 = fork();
    if(pid1 < 0) printf("fork1 error\n");
    if(pid1 == 0) http_server();

    pid2 = fork();
    if(pid2 < 0) printf("fork2 error\n");
    if(pid2 == 0) https_server();

    int st1, st2;
    waitpid(pid1, &st1, 0);
    waitpid(pid2, &st2, 0);
    return 0;
}
```

通过多进程，当客户端进行HTTP请求，进程1会监听到并进行响应，当客户端进行HTTPS请求，进程2会监听到并进行响应。实现了同时支持HTTP和HTTPS的服务器，并且能够传输正常HTML页面以及视频流。

遇到的问题

1. 不了解socket编程

2. 不了解https的工作机制
3. 不了解response中报文头部的构造及重要性

在查阅资料 and 不断实验后，对上述问题有了初步的了解并加以解决了。

总结

构造一个简单的http服务器，虽然实验项目不大，但却使自己对计算机网络教材中的概念更加清晰。实践是检验真理的唯一标准，只有通过实际操作才能检验自己对讲义上的知识是否真的了了解。另外，这次的实验也锻炼自己发现和解决问题的能力。总而言之，获益匪浅。