****

**计算机网络实验报告**

Socket编程进阶



**学 院 国际工程学院**

**年 级 2018**

**专 业 计算机技术**

**学 号 2018229038**

**姓 名 李亚蓉**

**2018年 12月12 日**

1. **实验目的**

使用Socket编程技术实现一个简单的HTTP Web服务器。该服务器能够为客户端实现html网页下载、客户端登陆信息的接收等用来展示基本的HTTP协议功能。

1. **实验原理**
2. **Get和Post方法**

**（1）GET方法：**如果GET方法只是简单的请求一份资源,而不传递参数的话则由服务器直接将资源返回即可。如果GET方法的url中带有参数的话，则就要使用CGI模式进行处理。

**（2）POST方法：**POST方法要使用CGI模式进行处理，POST的参数在消息中文中出现。

使用GET方法使用的是带参数的url，传递的参数会使用?连接在资源后面POST方法使用的是不带参数的url 它的参数是通过http请求正文传递给服务器的，http的请求和响应模式

**2、HTTP服务器实现框架**

**（1）面向链接：**

http协议是基于TCP通信协议，因此实现web服务器的第一步至少要能实现两个主机不同进程之间的TCP通信，并且需要解决高并发问题所以这里推荐使用多线程服务器来构建，每次创建出来一个新线程出来的时候将线程分离，然后让这个新线程去处理这个请求.

**（2）分析出请求行:**

当服务器接收到请求后，首先知道的是HTTP服务器版本号，和请求方法。web服务器是要支持cgi模式: 请求的方法不同，cgi可能也不同，我们实现的知识比较简单单的处理GET和POST方法

**（3）判断cgi模式：**

     当我们判断出来是GET请求时候，并且url中没有参数的话，就用非CGI模式，非CGI模式处理起来比较简单，首先解析出来请求路径，判断是不是合法资源，如果是就直接返回这个资源。

    当是CGI模式处理请求的时候，我们要fork一个子进程，对子进程exec替换CGI程序，这个过程中使用pipe进行父子进程之间的通信。所有需要的参数在exec之前，都将这些参数导出为环境变量，就算exec的话，子进程还是能够通过环境变量获取所需的参数。

**（4）响应客户端：**

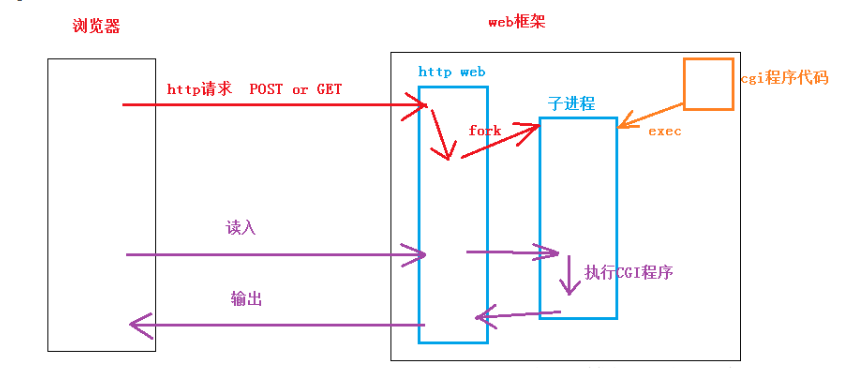
此时我们已经知道了方法以及是否为cgi模式，然后开始读取URL，这里有一个细节非cgi模式 请求参数会跟在url当中，如果cgi模式的话，参数在消息正文中，然后我们读取到路径，判断路径当中资源是否存在，如果存在判断这个资源是一个目录，普通文件还是一个可执行程序，这里分情况分析

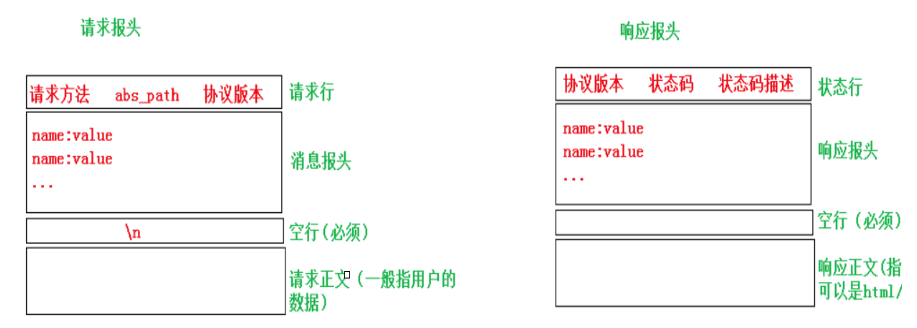
    a.如果是cgi模式，直接进入cgi内部运行;只要是POST方法就需要支持cgi,直接进入cgi函数内部运行.

    b.如果是非cgi模式时一定是GET方法并且没有参数，此时进入wwwroot()函数内部即可，该函数会将所请求的资源以html的格式返回给浏览器.

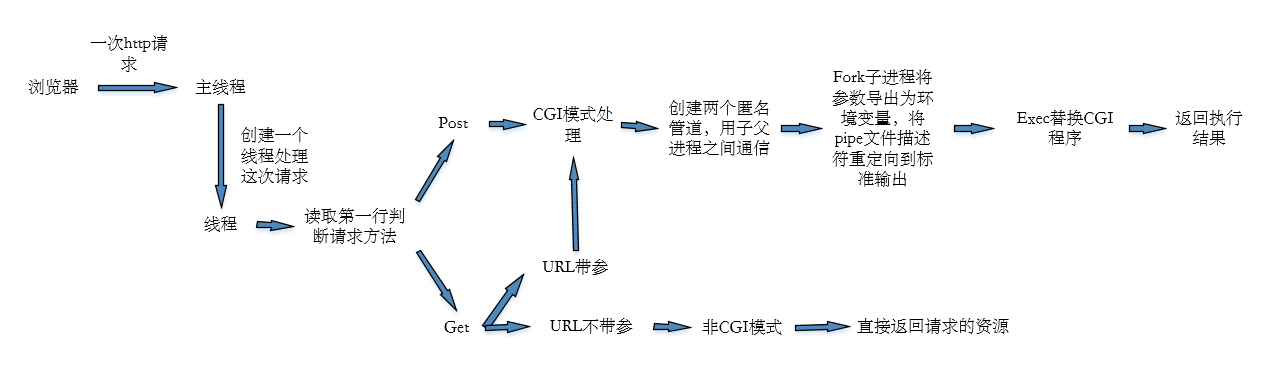
**（5）解释运行cgi模式(如图一)：**

首先服务器要从浏览器读取参数，然后创建出来一个子进程去执行cgi部分的可执行资源，父进程通过环境变量的方式传递给子进程，子进程运行完成之后呢，将结果交给父进程，父进程再将数据输出给浏览器. 所以父进程在这个例子当中就向是一个中介，只进行参数和结果的转交实际上并不会执行任何资源，因此将子进程的输入输出文件描述符重定向，就可以让子进程直接与浏览器"联系".

图一 CGI模式的过程

**3、HTTP的请求与响应格式（如图二）**：

图二 HTTP的请求与响应格式

1. 一次完整的HTTP请求的流程（如图三）：

图三 完整的HTTP请求的流程

1. **实验过程**

以下是完整的HTTP过渡到TCP实现客户端与服务器的交互过程：

1、当客户端执行网络请求的时候，从url中解析出url的主机名，并将主机地址转换成ip

2、从url解析出服务器的所用端口号

3、客户端用TCP连接服务器

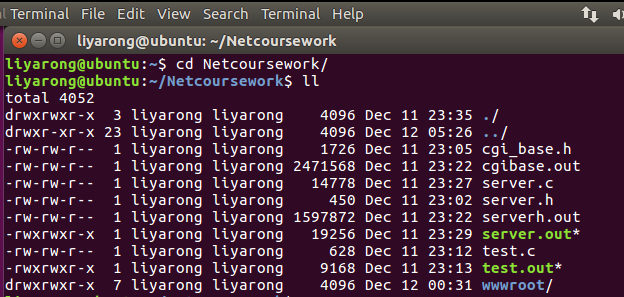
4、连接成功后 获取输出流，将数据以报文的形式传递给服务器

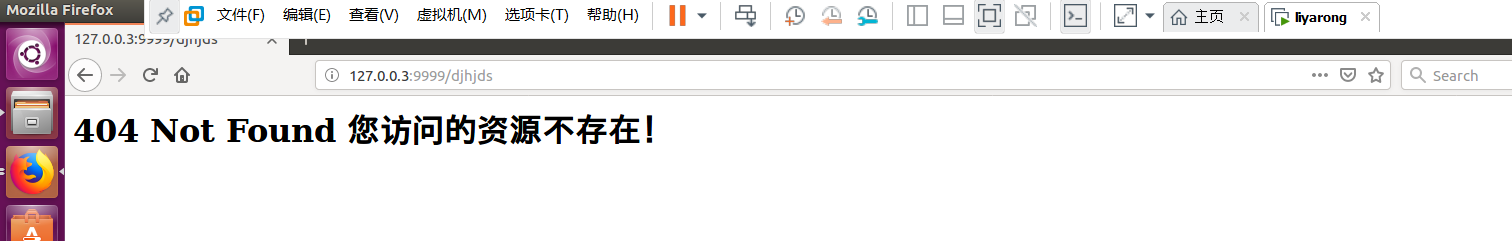
5、当服务器接收到数据之后，进行判断和解析码，并回应一条响应报文

6、客户端从输入流中获取报文，然后进行解析

7、关闭网络连接

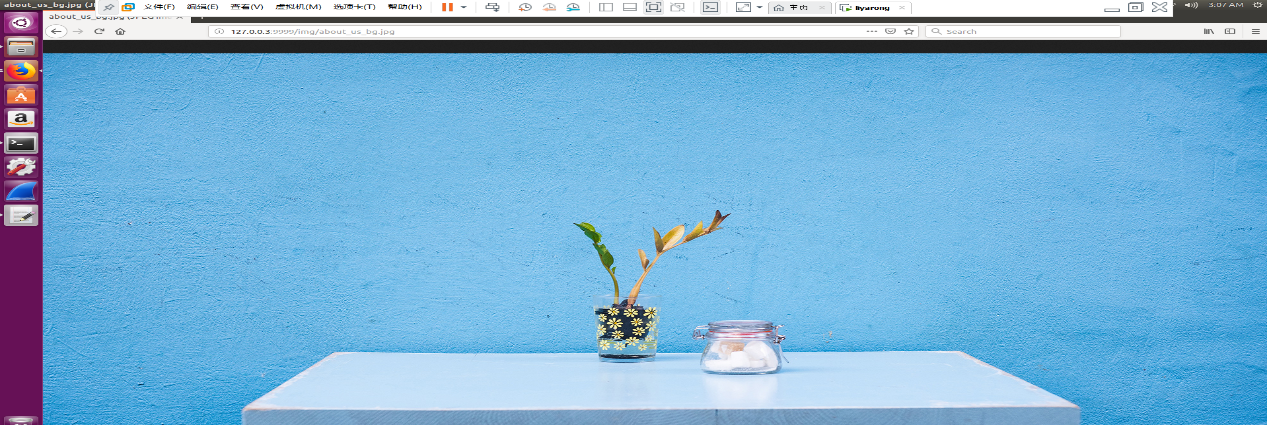
1. **实验结果**

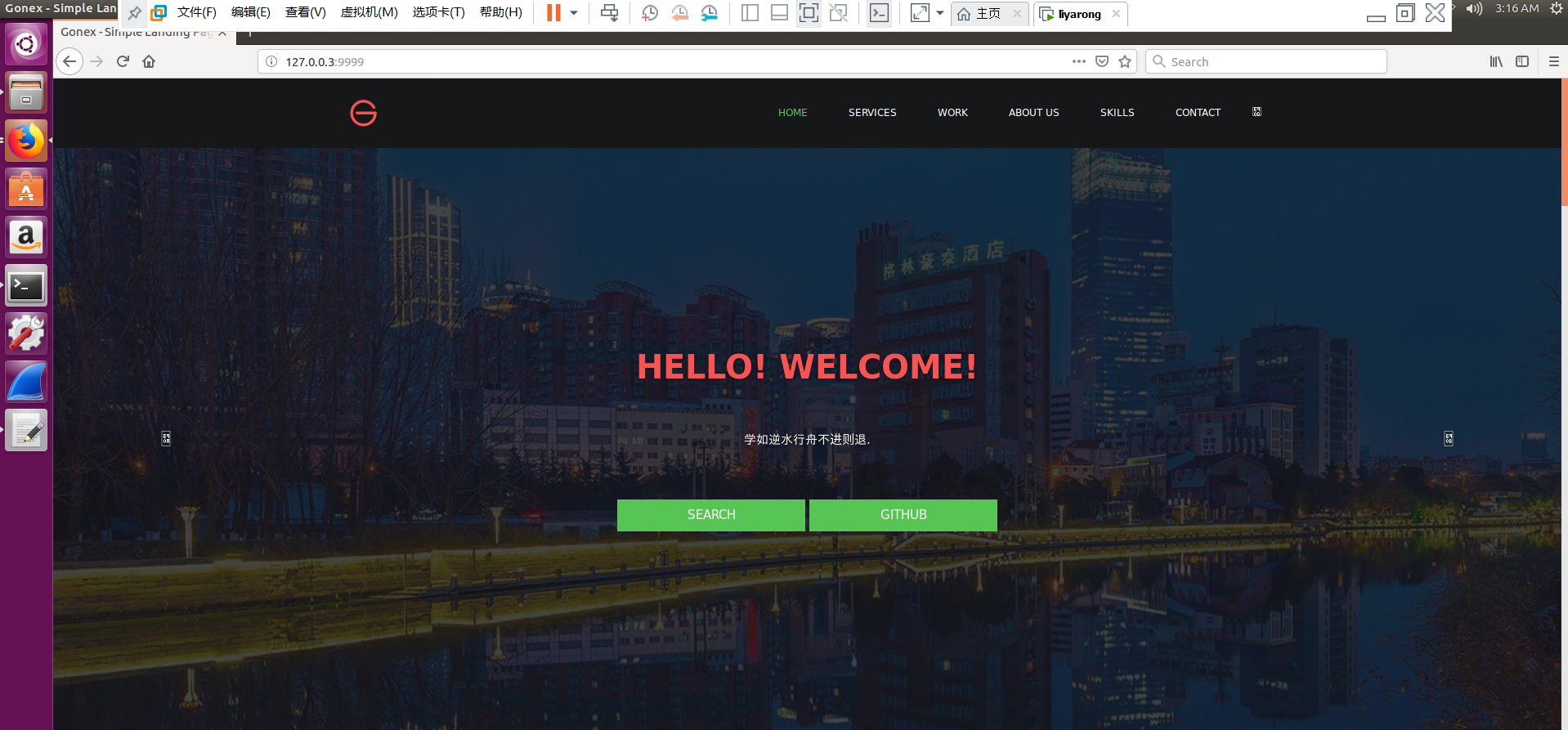


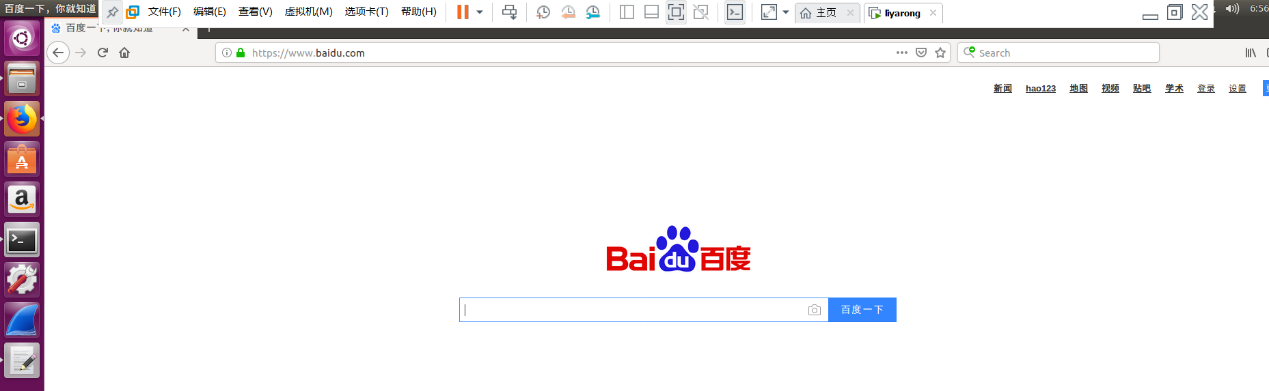
图四 目录文件‘’

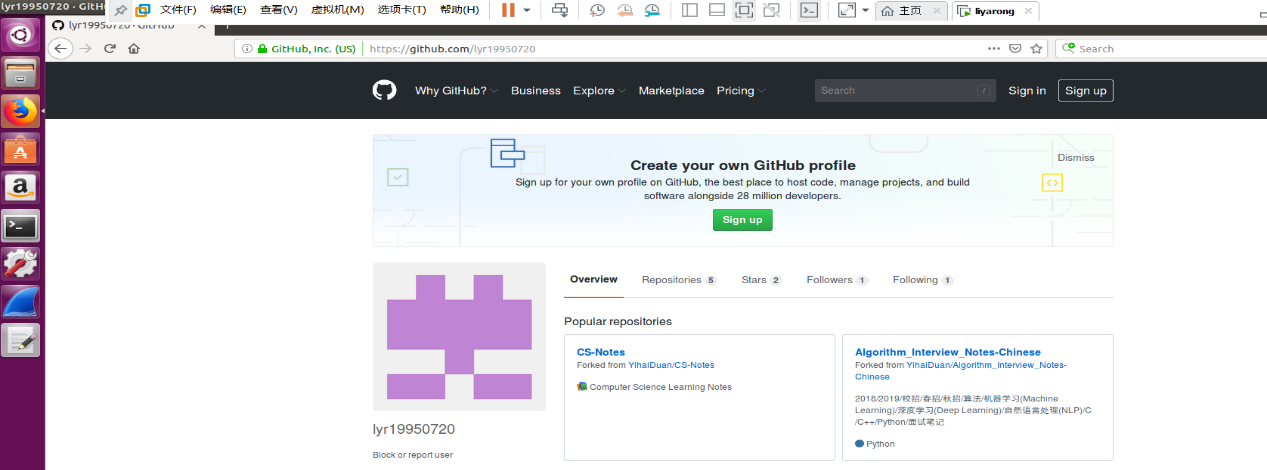
图五 访问的资源不存在

图六 访问的是文本资源

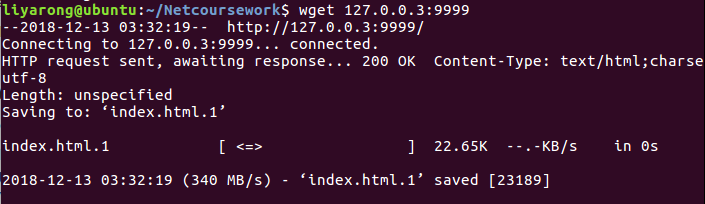
图七 访问的资源是图片



图八 访问某个网页

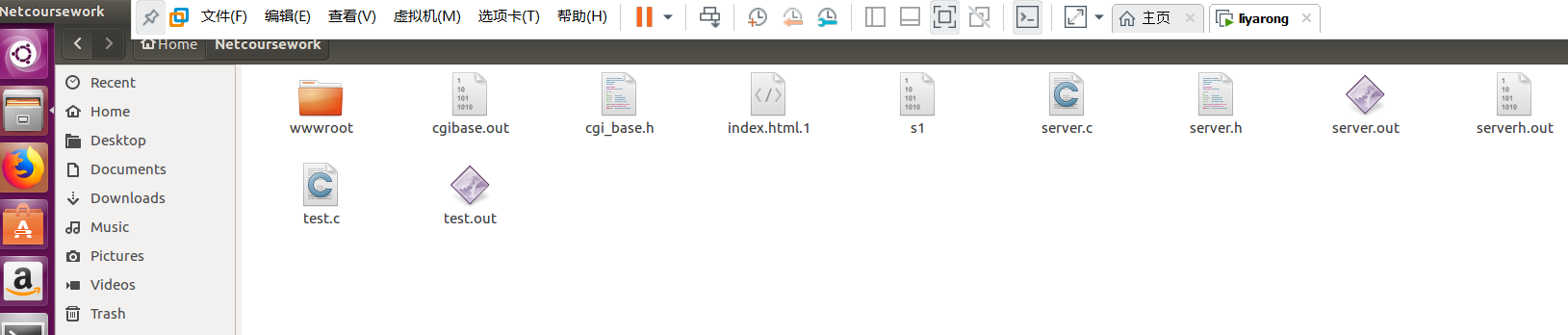
图九 点击Search进入百度搜索界面

图十 点击Github进入个人Github界面

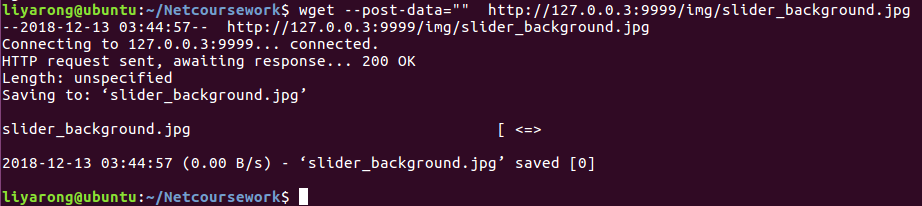


图十一 get请求

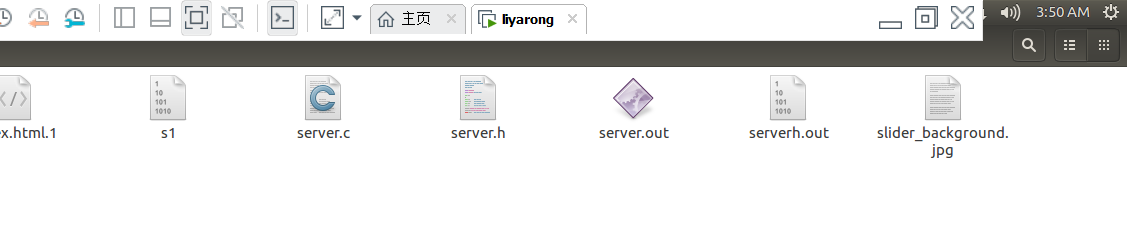
采用（图十一）get请求并且带有query\_string，触发 CGI程序，动态生成页面并下载到本地保存为index.html.1（图十二）



图十二 访问并下载的页面



图十三 post请求



图十四 访问下载的图片

采用（图十三）post请求并且带有query\_string，触发 CGI程序，动态生成页面并下载到本地保存为slider\_background.jpg（图十四）

1. **体会**

通过本次实验基本实现了以下功能：

（1）、实现最基本的HTTP/1.0版本的web服务器，客户端能够使用GET、POST方法请求资源 ；

（2）、服务器将客户请求的资源以html页面的形式呈现，能够返回一个静态页面，并能够进行差错处理（如：客户请求的资源不存在时，服务器能够返回一个404的页面）

（3）、服务器能进行简单的cgi程序·，能够根据 query\_string或者 body中的内容动态生成页面。比如当客户在表单中输入数据后，服务器能够将运行结果返回给客户 。但是在实验过程中遇到了以下几个问题：

（1）页面可以显示出来，但是乱码：

      解决方法：在响应报文生成时 content\_type ,设置utf8编码风格。成功解决了上述乱码问题。  const char\* type\_line="\" Content-Type:\"\" text/html;charset=utf-8\"\n";

（2）服务器经常显示端口被占用，发现服务器有大量Time\_wait状态。表示服务器端在等

最后一次ACK,而没有彻底断开与客户端的连接。

    解决方法：设置，使得可重用Time\_wait状态，成功解决上述问题。

     int opt=1;

     setsockopt(listen\_sock,SOL\_SOCKET,SO\_REUSEADDR,&opt,sizeof(opt));

（3） 刚开始，服务器只能处理一个客户端连接的情况，原因是，服务器是单线程，阻塞到了recv,没法循环去accept,导致无法处理另一个客户端连接的情况。

      解决方法：使用多线程的技术来处理，一个客户端连接进来，create一个线程去单独处理一个客户端的情况，主线程进行线程分离，返回执行accept,去处理下一个客户端的请求。

（4）关掉终端，就无法访问服务器。

       解决方法：利用守护进程的方式来运行。

1. **源码**
2. server.h

typedef struct Request{

char first\_line[SIZE];

char \*method;

char\* url;

char\* url\_path;//重点关注的内容1

char \*query\_string;//重点关注的内容2（参数）

// char\* version;

// 接下来是 header部分，如果要完整的解析下来，

// 此处需要使用二叉搜索树或者Hash表

// 这里我们偷个懒，其它header都不要了，只保留一个

// content-length

int content\_length;

}Request;

1. cgi\_base.h

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<unistd.h>

//分get post 两种情况来读取计算的参数

//1. GET 从 query\_string读取

//2. post 从body中读取

//读取结果就放在buf这个输出缓冲区中

static int GetQueryString(char buf[]){

//1. 从环境变量中获取到方法是什么

char \* method=getenv("REQUEST\_METHOD");

if(method==NULL){

//由于当前的CGI程序对应的标准输出已经被重定向

//到官道上了，而这部分数据又会被返回给客户端

//避免让程序内部的错误暴露给用户，通过 stderr来

//作为输入日志的手段

fprintf(stderr,"method==NULL\n");

return -1;

}

// 2. 判断方法是 GET 还是 POST。

// 如果是GET，就从环境变量里面读取 QUERY\_STRING

// 如果是 POST，就需要从环境变量里读取 CONTENT\_LENGTH

if(strcasecmp(method,"GET")==0){

char \*query\_string=getenv("QUERY\_STRING");

if(query\_string==NULL){

fprintf(stderr,"query\_string is NULL\n");

return -1;

}

//拷贝完成后，buf里边的内容形如 a=10&b=20

strcpy(buf,query\_string);

}else{

char \*content\_length\_str=getenv("CONTENT\_LENGTH");

if(content\_length\_str==NULL){

fprintf(stderr,"content\_length is NULL\n");

return -1;

}

int content\_length=atoi(content\_length\_str);

int i=0;

for(;i<content\_length;i++){

//此处由于父进程把body 已经写入管道

//子进程又把0号文件描述符重定向到了管道

//此时从标准输入来读数据，也就读到了管道中的数据

read(0,&buf[i],1);

}

buf[i]='\0';

//此循环读完后，buf里面的内容就形如

//a=10&b=20

}

return 0; }

1. server.c

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<arpa/inet.h>

#include<pthread.h>

#include"server.h"

#include <sys/stat.h>

#include<fcntl.h>

#include<sys/sendfile.h>

#include<sys/wait.h>

typedef struct sockaddr sockaddr;

typedef struct sockaddr\_in sockaddr\_in;

//一次从socket中读取一行数据,把数据放到buf缓冲区中,如果读取失败，返回-1,\n \r \r\n

int ReadLine(int sock,char buf[],ssize\_t size){

//从socket中一次读取一个字符

char c='\0';

ssize\_t i=0;//当前读了多少字符

//结束条件：

//a) 读的长度太长，达到了缓冲区长度的上限

//b) 读到了\n(此处我们要兼容\r \r\n的情况，如果遇到\r\n或者\r 想办法换成\n

while(i<size-1&&c!='\n'){

ssize\_t read\_size=recv(sock,&c,1,0);

if(read\_size<=0){

//预期是读到\n这样的换行符，结果还没读到呢，

//就先读到了EOF,这种情况我们也暂时认为是失败

return -1;

}

if(c=='\r'){

//当前遇到了\r,但是还需要确定下一个字符是不是\n

//MSG\_PEEK 选项从内核的缓冲区中读数据

//但是读到的数据不会从缓冲区中删除掉

recv(sock,&c,1,MSG\_PEEK);

if(c=='\n'){

recv(sock,&c,1,0);

}else{

c='\n';

}

}

buf[i++]=c;

}

buf[i]='\0';

return i;//实际读到的字符个数

//如果读到\n就返回

}

int split(char input[],char\*split\_char,char \*output[],int output\_size){

// strtok不是一个线程安全的函数。因为根据其定义，它必须使用

// 内部静态变量来记录字符串中下一个需要解析的标记的当前位置。

// 但是，由于指示这个位置的变量只有一个，那么，在同一个程序中出

// 现多个解析不同字符串的strtok调用时，各自的字符串的解析就会互相干扰。所以使用strtok\_r

//strtok\_r会破坏待分解字符串的完整性，但是将剩余字符串保存在手动传入的缓冲区中。保证了安全性。

int i=0;

char \*pch;

char\*tmp=NULL;//strtok\_r需要手动传入缓冲区

pch=strtok\_r(input,split\_char,&tmp);

while(pch!=NULL){

if(i>=output\_size){

return i;

}

output[i++]=pch;

pch=strtok\_r(NULL,split\_char,&tmp);

}

return i;

}

int ParseFirstLine(char first\_line[],char\*\*p\_url,char\*\*p\_method){

//把首行按照空格进行字符串切分

char \*tok[10];

//切分得到的每一个部分，就放到tok数组之中

//返回值就是tok数组中包含几个元素

//最后一个参数10表示tok数组中最多能放几个元素

int tok\_size=split(first\_line," ",tok,10);

if(tok\_size!=3){//切分出来的是 方法 url 版本号

printf("Split failed! tok\_size=%d\n",tok\_size);

return -1;

}

\*p\_method=tok[0];

\*p\_url=tok[1];

return 0;

}

int ParseQueryString(char\*url,char\*\*p\_url\_path,char\*\*p\_query\_string){

\*p\_url\_path=url;

char\*p=url;

for(;\*p!='\0';++p){

if(\*p=='?'){

\*p='\0';

\*p\_query\_string=p+1;

return 0;

}

}

//循环结束都没找到？，说明这个请求不带query\_string

\*p\_query\_string=NULL;

return 0;

}

int ParseHeader(int sock,int \*content\_length){

char buf[SIZE]={0};

while(1){

//1循环从socket中读取一行

size\_t read\_size=ReadLine(sock,buf,sizeof(buf));

//处理读失败的情况

if(read\_size<=0){

return -1;

}

//处理读完的情况，读到空行

if(strcmp(buf,"\n")==0){

return 0;

}

//2 判定当前行是不是Content-Length

//如果是，就直接把value读出来

//如果不是就丢弃

//strncmp比较字符串前n个字符是否相等

const char\*content\_length\_str="Content-Length: ";

if(content\_length!=NULL&&strncmp(buf,content\_length\_str,strlen(content\_length\_str))==0){

\*content\_length=atoi(buf+strlen(content\_length\_str));//只把value的值赋给content\_length

}

}

return 0;

}

void Handler404(int sock){

//构造一个完整的HTTP响应

//状态码就是404

//body部分应该也是一个404页面

const char \* first\_line="HTTP/1.1 404 NOT Found\n";

const char\* blank\_line="\n";

const char\* type\_line="\" Content-Type:\"\" text/html;charset=utf-8\"\n";//在headler部分也加上，增强代码的健壮性，保证

//任何浏览器都不乱码

const char\* html="<head><meta http-equiv=\"Content-Type\" content=\"text/html;charset=utf-8\"> </head>"

"<h1>404 Not Found 您访问的资源不存在！</h1>";

send(sock,first\_line,strlen(first\_line),0);

send(sock,type\_line,strlen(type\_line),0);

send(sock,blank\_line,strlen(blank\_line),0);

send(sock,html,strlen(html),0);

return;

}

int IsDir(const char\* file\_path){//判断是文件还是目录

struct stat st;

int ret=stat(file\_path,&st);//stat可以获取到file\_path的详细信息，存放在结构体st中

if(ret<0){//文件不存在，表示不是目录

return 0;

}

if(S\_ISDIR(st.st\_mode)){//通过S\_ISDIR这个宏来判定file\_path是否是一个目录，st.mode保存的是文件类型

return 1;

}

return 0;

}

void HandlerFilePath(const char\*url\_path,char file\_path[]){

//HTTP服务器的根目录：用户指定的任意一个存在的目录

//url\_path =>/index.html

//file\_path=>./wwwroot/index.html

//a)给 url\_path 加上前缀

sprintf(file\_path,"./wwwroot%s",url\_path);//字符串拼接

//b)例如：url\_path => / 请求的是一个目录，想办法给目录中追加上一个 index.html文件（取一个默认文件）

if(file\_path[strlen(file\_path)-1]=='/'){//判断file\_path最后一个字符是否是‘\’

strcat(file\_path,"index.html");

}

//c) 例如 url\_path => /image无法确定是文件还是目录，需要判定

if(IsDir(file\_path)){

strcat(file\_path,"/index.html");

}

return;

}

ssize\_t GetFileSize(const char\*file\_path){

struct stat st;

int ret=stat(file\_path,&st);

if(ret<0){

//打开文件失败，很可能文件不存在

//此时返回的文件长度为0

return 0;

}

return st.st\_size;

}

int WriteStaticFile(int sock,const char\*file\_path){

//1 打开文件

//什么情况下会打开文件失败 a) 文件描述符不够用 b) 文件不存在

int fd=open(file\_path,O\_RDONLY);

if(fd<0){

perror("open");

return 404;

}

//2 把构造出的HTTP响应写入到socket之中

//a)写入首行

const char\* first\_line="HTTP/1.1 200 OK\n";

send(sock,first\_line,strlen(first\_line),0);

//b)写入header

//可根据扩展名来显示类型，但现在绝大多数浏览器都能自动匹配类型，根据扩展名自动匹配类型

const char\* type\_line=" Content-Type: text/html;charset=utf-8\n";//文本类型

//const char\* type\_line="\" Content-Type:\"\" image/jpg;charset=utf-8\"\n";//图片类型

send(sock,type\_line,strlen(type\_line),0);

//c)写入空行

const char\* blank\_line="\n";

send(sock,blank\_line,strlen(blank\_line),0);

//d)写入body(文件内容）

/\*不够高效

//把文件中所有的文件一个字节一个字节的读取出来再一个字节一个字节写入sock

ssize\_t file\_size=GetFileSize(file\_path);//获取到文件的总字节数

ssize\_t i=0;

for(;i<file\_size;i++){

char c;

read(fd,&c,1);//系统调用不够高效，设计从内核向用户缓存拷贝数据

send(sock,&c,1,0);//涉及从用户缓存向网卡中拷贝数据

}

\*/

sendfile(sock,fd,NULL,GetFileSize(file\_path));//直接从内核到内核的拷贝

//避免了内核和用户之间得来回拷贝，从fd拷贝到网卡上，NULL表示从起始

//位置开始拷贝

//3 关闭文件

close(fd);

return 200;

}

int HandlerStaticFile(int sock,Request\*req){//静态生成页面

//1 根据 url\_path 获取到文件在服务器上的真实路径

char file\_path[SIZE]={0};//用来保存文件的真实路径

HandlerFilePath(req->url\_path,file\_path);//利用url来找到文件的真实路径

//2 读取文件，把文件的内容直接写回到socket之中

int err\_code=WriteStaticFile(sock,file\_path);

return err\_code;

}

int HandlerCGIFather(int new\_sock,int father\_read,int father\_write,int child\_pid,Request\*req){

//如果是post请求，就把body写入到管道

if(strcasecmp(req->method,"POST")==0){

int i=0;

char c='\0';

for(;i<req->content\_length;i++){

read(new\_sock,&c,1);

write(father\_write,&c,1);

}

}

//2 构造HTTP响应

const char\*first\_line="HTTP/1.1 200 OK\n";

send(new\_sock,first\_line,strlen(first\_line),0);

const char\* type\_line="\" Content-Type:\"\" text/html;charset=utf-8\"\n";

send(new\_sock,type\_line,strlen(type\_line),0);

const char\* blank\_line="\n";

send(new\_sock,blank\_line,strlen(blank\_line),0);

//循环从管道中读取数据，并写入到socket

char c='\0';

while(read(father\_read,&c,1)>0){//读管道，读到EOF返回0，什么

//时候读到EOF,写端全部关闭，再尝试继续读，会读到EOF，管道读端

//关闭，再尝试写，会产生SIDPIPE信号，导致进程异常终止

send(new\_sock,&c,1,0);

}

//4.回收子进程的资源

waitpid(child\_pid,NULL,0);//回收对应的子进程

return 200;

}

int HandlerCGIChild(int child\_read,int child\_write,Request\*req){

//1.设置环境变量

char method\_env[SIZE]={0};

sprintf(method\_env,"REQUEST\_METHOD=%s",req->method);

putenv(method\_env);

//还需要设置 QUERY\_STRING 或者 CONTENT\_LENGTH

if(strcasecmp(req->method,"GET")==0){

char query\_string\_env[SIZE]={0};

sprintf(query\_string\_env,"QUERY\_STRING=%s",req->query\_string);

putenv(query\_string\_env);

}else{

char content\_length\_env[SIZE]={0};

sprintf(content\_length\_env,"CONTENT\_LENGTH=%d",req->content\_length);

putenv(content\_length\_env);

}

//2.把标准输入，标准输出重定向到管道中

dup2(child\_read,0);

dup2(child\_write,1);

//3.对子进程进行程序替换

// url\_path: /cgi-bin/test

// file\_path: ./wwwroot/cgi-bin/test

char file\_path[SIZE]={0};

HandlerFilePath(req->url\_path,file\_path);

//l

//lp

//le

//v

//vp

//ve

execl(file\_path,file\_path,NULL);

return 200;

}

int HandlerCGI(int new\_sock,Request\*req){//动态生成页面

//1.创建一对匿名管道

int fd1[2],fd2[2];

int ret=pipe(fd1);

if(ret<0){

return 404;

}

ret=pipe(fd2);

if(ret<0){

//第二个管道创建失败，记得要关闭第一个管道

//的读端和写端

close(fd1[0]);

close(fd2[1]);

return 404;

}

// fd1 fd2 这样的变量名描述性太差，后边直接使用的话

// 是非常容易弄混淆的，所以直接在此处定义几个更加明确

// 的变量来描述该文件描述符的用途。

int father\_read=fd1[0];

int child\_write=fd1[1];

int father\_write=fd2[1];

int child\_read=fd2[0];

//2.创建子进程

ret=fork();

//3.父子进程各自执行不同的逻辑

if(ret>0){

//farther

//注意关掉父进程不用的文件描述符

//此处父进程优先关闭这俩个管道的文件描述符

//是为了后续父进程从子进程这里读数据的时候

//能够读到EOF,对于管道来说，所有的写端关闭

//，继续读，才有EOF,而此处所有的写端，一方面

//是父进程需要关闭，另一方面子进程也需要关闭

//所以此处父进程先关闭不必要的写端置之后，后

//续子进程用完了，直接关闭，父进程也就直接读到了

//EOF

close(child\_read);

close(child\_write);

int err\_code=HandlerCGIFather(new\_sock,father\_read,father\_write,ret,req);

}else if(ret==0){

//child

//注意关掉子进程不用的文件描述符

close(father\_read);

close(father\_write);

int err\_code=HandlerCGIChild(child\_read,child\_write,req);

}else{

perror("fork");

close(fd1[0]);

close(fd1[1]);

close(fd2[0]);

close(fd2[1]);

return 404;

}

//4.收尾工作和错误处理

}

void HandlerRequest(int new\_sock){

int err\_code=200;

//1.读取并解析请求

Request req;

memset(&req,0,sizeof(req));

//a)从socket中读取首行

if(ReadLine(new\_sock,req.first\_line,sizeof(req.first\_line))<=0){

//todo失败处理

err\_code=404;

goto END;

}

//b）从首行中解析出url和method

if(ParseFirstLine(req.first\_line,&req.url,&req.method)<0){

//todo失败处理

err\_code=404;

goto END;

}

//c)解析url,从url中解析出url\_path,query\_string

if(ParseQueryString(req.url,&req.url\_path,&req.query\_string)<0){

//todo失败处理

err\_code=404;

goto END;

}

//处理header,丢弃了大部分header,只处理content-length

if(ParseHeader(new\_sock,&req.content\_length)<0){

//todo失败处理

err\_code=404;

goto END;

}

printf("method: %s\n",req.method);

printf("url\_path: %s\n",req.url\_path);

printf("query\_string: %s\n",req.query\_string);

printf("content\_length: %d\n",req.content\_length);

//2.静态、动态生成页面，把生成页面写回客户端

//假如浏览器发送的请求方法叫“get”/"GET"，不区分大小写的比较

//如果请求是get请求并且没有query\_string，那么就返回静态页面

if(strcasecmp(req.method,"GET")==0

&&req.query\_string==NULL){

err\_code=HandlerStaticFile(new\_sock,&req);

}

//如果是get请求，有query\_string,那么就返回动态页面

else if(strcasecmp(req.method,"GET")==0

&&req.query\_string!=NULL){

err\_code=HandlerCGI(new\_sock,&req);

}

//如果是post请求（一定是带参数的，参数是通过body来传给服务器的），那么也返回动态页面

else if(strcasecmp(req.method,"POST")==0){

err\_code=HandlerCGI(new\_sock,&req);

}else{//收到其他方法

//todo错误处理

err\_code=404;

goto END;

}

//错误处理：直接返回一个404的HTTP响应

END:

if(err\_code!=200){

Handler404(new\_sock);

}

close(new\_sock);

return;

}

void \* ThreadEntry(void \* arg){

int64\_t new\_sock=(int64\_t)arg;

//此处HandlerRequest作用是为了解耦合，一旦需要把服务器改成多进程或者IO多路复用

//整体代码的改动都是比较小的

HandlerRequest(new\_sock);

return NULL;

}

//服务器启动

void HttpServerStart(const char\*ip,short port){

int listen\_sock=socket(AF\_INET,SOCK\_STREAM,0);

if(listen\_sock<0){

perror("socket");

return ;

}

//设置，使得可重用Time-wait状态

int opt=1;

setsockopt(listen\_sock,SOL\_SOCKET,SO\_REUSEADDR,&opt,sizeof(opt));

sockaddr\_in addr;

addr.sin\_family=AF\_INET;

addr.sin\_addr.s\_addr=inet\_addr(ip);//将点分十进制的ip地址转换成32位的

addr.sin\_port=htons(port);//主机序的字节序转换为网络序的字节序

int ret=bind(listen\_sock,(sockaddr\*)&addr,sizeof(addr));

if(ret<0){

perror("bind");

return ;

}

ret=listen(listen\_sock,5);

if(ret<0){

perror("listern");

return ;

}

printf("Server Init Ok\n");

while(1){

sockaddr\_in peer;//对端ip,端口号

socklen\_t len=sizeof(peer);

int64\_t new\_sock=accept(listen\_sock,(sockaddr\*)&peer,&len);

if(new\_sock<0){

perror("accept");

continue;

}

//使用多线程来实现Tcp服务器

pthread\_t tid;//定义一个线程号

pthread\_create(&tid,NULL,ThreadEntry,(void \*)new\_sock);

pthread\_detach(tid);

}

}

int main(int argc,char\*argv[]){

if(argc!=3){

printf("Usage ./server [ip] [port]\n");

return 1;

}

HttpServerStart(argv[1],atoi(argv[2]));

}

1. test.c

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<unistd.h>

#include"cgi\_base.h"

int main(){

char buf[1024\*10]={0};

int ret= GetQueryString(buf);

if(ret<0){

fprintf(stderr,"[CGI] Getquerystring failed!\n");

}

int a,b;

sscanf(buf,"a=%d&b=%d",&a,&b);

int sum=a+b;

//printf 输出的结果就会被返回到客户端上

//作为 HTTP服务器，每次给客户端返回的字符串必须

//符合HTTP协议的格式

//由于父进程已经把 首行 header 空行 都已经写回给客户端

//因此此时 CGI 程序只返回 body 部分即可，就是html格式的数据

printf("<h1>sum=%d</h1>",sum);

return 0;

}