Server Http

代码实现

头文件

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <ctype.h>
#include <strings.h>
#include <string.h>
#include <sys/stat.h>
//#include <pthread.h>
#include <sys/wait.h>
#include <stdlib.h>
```

宏

```
#define ISspace(x) isspace((int)(x))

#define SERVER_STRING "Server: jdbhttpd/0.1.0\r\n"
```

accept_request

```
/**********************************************************************/
/* A request has caused a call to accept() on the server port to
 * return.  Process the request appropriately. * Parameters: the socket connect
void accept_request(int client)
{
 char buf[1024];
 int numchars;
 char method[255];
 char url[255];
 char path[512];
```

```
 size_t i, j;
 struct stat st;
 int cgi = 0;      /* becomes true if server decides this is a CGI
                    * program */ char *query_string = NULL;

 //读http 请求的第一行数据（request line），把请求方法存进 method 中
 numchars = get_line(client, buf, sizeof(buf));
 i = 0; j = 0;
 while (!ISspace(buf[j]) && (i < sizeof(method) - 1))
 {
  method[i] = buf[j];
  i++; j++;
 }
 method[i] = '\0';

 //如果请求的方法不是 GET 或 POST 任意一个的话就直接发送 response 告诉客户端没实现该方法
 if (strcasecmp(method, "GET") && strcasecmp(method, "POST"))
 {
  unimplemented(client);
  return;
 }

 //如果是 POST 方法就将 cgi 标志变量置一(true)
 if (strcasecmp(method, "POST") == 0)
  cgi = 1;

 i = 0;
 //跳过所有的空白字符(空格)
 while (ISspace(buf[j]) && (j < sizeof(buf)))
j++;

 //然后把 URL 读出来放到 url 数组中
 while (!ISspace(buf[j]) && (i < sizeof(url) - 1) && (j < sizeof(buf)))
 {
  url[i] = buf[j];
  i++; j++;
 }
 url[i] = '\0';

 //如果这个请求是一个 GET 方法的话
 if (strcasecmp(method, "GET") == 0)
 {
  //用一个指针指向 url  query_string = url;

  //去遍历这个 url，跳过字符 ？前面的所有字符，如果遍历完毕也没找到字符 ？则退出循环
  while ((*query_string != '?') && (*query_string != '\0'))
   query_string++;
```

```
 //退出循环后检查当前的字符是 ? 还是字符串(url)的结尾
 if (*query_string == '?')
 {
  //如果是 ?  的话，证明这个请求需要调用 cgi，将 cgi 标志变量置一(true)
  cgi = 1;
  //从字符 ?  处把字符串 url 给分隔会两份
  *query_string = '\0';
  //使指针指向字符 ? 后面的那个字符
  query_string++;
 }
}

//将前面分隔两份的前面那份字符串，拼接在字符串htdocs的后面之后就输出存储到数组 path 中。相当
sprintf(path, "htdocs%s", url);

//如果 path 数组中的这个字符串的最后一个字符是以字符 / 结尾的话，就拼接上一个"index.html"
if (path[strlen(path) - 1] == '/')
 strcat(path, "index.html");

//在系统上去查询该文件是否存在
if (stat(path, &st) == -1) {
 //如果不存在，那把这次 http 的请求后续的内容(head 和 body)全部读完并忽略
 while ((numchars > 0) && strcmp("\n", buf))  /* read & discard headers */
  numchars = get_line(client, buf, sizeof(buf));
 //然后返回一个找不到文件的 response 给客户端
 not_found(client);
}
else
{
 //文件存在，那去跟常量S_IFMT相与，相与之后的值可以用来判断该文件是什么类型的
 //S_IFMT参读《TLPI》P281，与下面的三个常量一样是包含在<sys/stat.h>
 if ((st.st_mode & S_IFMT) == S_IFDIR)
//如果这个文件是个目录，那就需要再在 path 后面拼接一个"/index.html"的字符串
  strcat(path, "/index.html");

  //S_IXUSR, S_IXGRP, S_IXOTH三者可以参读《TLPI》P295
 if ((st.st_mode & S_IXUSR) ||
     (st.st_mode & S_IXGRP) ||
     (st.st_mode & S_IXOTH)    )
  //如果这个文件是一个可执行文件，不论是属于用户/组/其他这三者类型的，就将 cgi 标志变量置一
  cgi = 1;

 if (!cgi)
  //如果不需要 cgi 机制的话，
  serve_file(client, path);
 else
  //如果需要则调用
  execute_cgi(client, path, method, query_string);
}
```

```
 }

 close(client);
}
```

bad_request

```
/**********************************************************************/
/* Inform the client that a request it has made has a problem.
 * Parameters: client socket *//***********************************************
void bad_request(int client)
{
 char buf[1024];

 sprintf(buf, "HTTP/1.0 400 BAD REQUEST\r\n");
 send(client, buf, sizeof(buf), 0);
 sprintf(buf, "Content-type: text/html\r\n");
 send(client, buf, sizeof(buf), 0);
 sprintf(buf, "\r\n");
 send(client, buf, sizeof(buf), 0);
 sprintf(buf, "<P>Your browser sent a bad request, ");
 send(client, buf, sizeof(buf), 0);
 sprintf(buf, "such as a POST without a Content-Length.\r\n");
 send(client, buf, sizeof(buf), 0);
}
```

cat

```
/**********************************************************************/
/* Put the entire contents of a file out on a socket.  This function
 * is named after the UNIX "cat" command, because it might have been * easier ju
void cat(int client, FILE *resource)
{
 char buf[1024];

 //从文件文件描述符中读取指定内容
 fgets(buf, sizeof(buf), resource);
 while (!feof(resource))
 {
  send(client, buf, strlen(buf), 0);
  fgets(buf, sizeof(buf), resource);
 }
}
```

cannot_execute

```
/**********************************************************************/
/* Inform the client that a CGI script could not be executed.
 * Parameter: the client socket descriptor. *//*******************************
void cannot_execute(int client)
{
 char buf[1024];

 sprintf(buf, "HTTP/1.0 500 Internal Server Error\r\n");
 send(client, buf, strlen(buf), 0);
 sprintf(buf, "Content-type: text/html\r\n");
 send(client, buf, strlen(buf), 0);
 sprintf(buf, "\r\n");
 send(client, buf, strlen(buf), 0);
 sprintf(buf, "<P>Error prohibited CGI execution.\r\n");
 send(client, buf, strlen(buf), 0);
}
```

error_die

```
/**********************************************************************/
/* Print out an error message with perror() (for system errors; based
 * on value of errno, which indicates system call errors) and exit the * prograı
void error_die(const char *sc)
{
 //包含于<stdio.h>,基于当前的 errno 值，在标准错误上产生一条错误消息。参考《TLPI》P49
 perror(sc);
 exit(1);
}
```

execute_cgi

```
/**********************************************************************/
/* Execute a CGI script.  Will need to set environment variables as
 * appropriate. * Parameters: client socket descriptor *             path to th
void execute_cgi(int client, const char *path,
                 const char *method, const char *query_string)
{
 char buf[1024];
 int cgi_output[2];
```

```
int cgi_input[2];
pid_t pid;
int status;
int i;
char c;
int numchars = 1;
int content_length = -1;

//往 buf 中填东西以保证能进入下面的 while buf[0] = 'A'; buf[1] = '\0';
//如果是 http 请求是 GET 方法的话读取并忽略请求剩下的内容
if (strcasecmp(method, "GET") == 0)
 while ((numchars > 0) && strcmp("\n", buf))  /* read & discard headers */
  numchars = get_line(client, buf, sizeof(buf));
else    /* POST */
{
 //只有 POST 方法才继续读内容
 numchars = get_line(client, buf, sizeof(buf));
 //这个循环的目的是读出指示 body 长度大小的参数，并记录 body 的长度大小。其余的 header 里
 //注意这里只读完 header 的内容，body 的内容没有读
 while ((numchars > 0) && strcmp("\n", buf))
 {
  buf[15] = '\0';
  if (strcasecmp(buf, "Content-Length:") == 0)
   content_length = atoi(&(buf[16])); //记录 body 的长度大小
  numchars = get_line(client, buf, sizeof(buf));
 }
 //如果 http 请求的 header 没有指示 body 长度大小的参数，则报错返回
 if (content_length == -1) {
  bad_request(client);
  return;
 }
}

sprintf(buf, "HTTP/1.0 200 OK\r\n");
send(client, buf, strlen(buf), 0);

//下面这里创建两个管道，用于两个进程间通信
if (pipe(cgi_output) < 0) {
 cannot_execute(client);
 return;
}
if (pipe(cgi_input) < 0) {
 cannot_execute(client);
 return;
}

//创建一个子进程
if ( (pid = fork()) < 0 ) {
```

```
 cannot_execute(client);
 return;
}
//子进程用来执行 cgi 脚本
if (pid == 0)  /* child: CGI script */
{
 char meth_env[255];
 char query_env[255];
 char length_env[255];

 //dup2()包含<unistd.h>中，参读《TLPI》P97
 //将子进程的输出由标准输出重定向到 cgi_ouput 的管道写端上
 dup2(cgi_output[1], 1);
 //将子进程的输出由标准输入重定向到 cgi_ouput 的管道读端上
 dup2(cgi_input[0], 0);
 //关闭 cgi_ouput 管道的读端与cgi_input 管道的写端
 close(cgi_output[0]);
 close(cgi_input[1]);

 //构造一个环境变量
 sprintf(meth_env, "REQUEST_METHOD=%s", method);
 //putenv()包含于<stdlib.h>中，参读《TLPI》P128
 //将这个环境变量加进子进程的运行环境中
 putenv(meth_env);

 //根据http 请求的不同方法，构造并存储不同的环境变量
 if (strcasecmp(method, "GET") == 0) {
  sprintf(query_env, "QUERY_STRING=%s", query_string);
  putenv(query_env);
 }
 else {   /* POST */
  sprintf(length_env, "CONTENT_LENGTH=%d", content_length);
  putenv(length_env);
 }
 //execl()包含于<unistd.h>中，参读《TLPI》P567
 //最后将子进程替换成另一个进程并执行 cgi 脚本
 execl(path, path, NULL);
 exit(0);

} else {    /* parent */
 //父进程则关闭了 cgi_output管道的写端和 cgi_input 管道的读端
 close(cgi_output[1]);
 close(cgi_input[0]);

 //如果是 POST 方法的话就继续读 body 的内容，并写到 cgi_input 管道里让子进程去读
 if (strcasecmp(method, "POST") == 0)
  for (i = 0; i < content_length; i++) {
   recv(client, &c, 1, 0);
```

```
      write(cgi_input[1], &c, 1);
     }
     //然后从 cgi_output 管道中读子进程的输出，并发送到客户端去
    while (read(cgi_output[0], &c, 1) > 0)
     send(client, &c, 1, 0);

    //关闭管道
    close(cgi_output[0]);
    close(cgi_input[1]);
    //等待子进程的退出
    waitpid(pid, &status, 0);
   }
  }
```

get_line

```
/**********************************************************************/
/* Get a line from a socket, whether the line ends in a newline,
 * carriage return, or a CRLF combination.  Terminates the string read * with a
int get_line(int sock, char *buf, int size)
{
 int i = 0;
 char c = '\0';
 int n;

 while ((i < size - 1) && (c != '\n'))
 {
  //recv()包含于<sys/socket.h>,参读《TLPI》P1259,
  //读一个字节的数据存放在 c 中
  n = recv(sock, &c, 1, 0);
  /* DEBUG printf("%02X\n", c); */
  if (n > 0)
  {
   if (c == '\r')
   {
    //
    n = recv(sock, &c, 1, MSG_PEEK);
    /* DEBUG printf("%02X\n", c); */
    if ((n > 0) && (c == '\n'))
     recv(sock, &c, 1, 0);
    else
     c = '\n';
   }
   buf[i] = c;
   i++;

  }
```

```
    }
    else
     c = '\n';
   }
   buf[i] = '\0';

   return(i);
  }
```

headers

```
/**********************************************************************/
/* Return the informational HTTP headers about a file. */
/* Parameters: the socket to print the headers on
 *             the name of the file *//*************************************
void headers(int client, const char *filename)
{
 char buf[1024];
 (void)filename;  /* could use filename to determine file type */

 strcpy(buf, "HTTP/1.0 200 OK\r\n");
 send(client, buf, strlen(buf), 0);
 strcpy(buf, SERVER_STRING);
 send(client, buf, strlen(buf), 0);
 sprintf(buf, "Content-Type: text/html\r\n");
 send(client, buf, strlen(buf), 0);
 strcpy(buf, "\r\n");
 send(client, buf, strlen(buf), 0);
}
```

not_found

```
/**********************************************************************/
/* Give a client a 404 not found status message. */
/**********************************************************************/
void not_found(int client)
{
 char buf[1024];

 sprintf(buf, "HTTP/1.0 404 NOT FOUND\r\n");
 send(client, buf, strlen(buf), 0);
 sprintf(buf, SERVER_STRING);
 send(client, buf, strlen(buf), 0);
```

```
 sprintf(buf, "Content-Type: text/html\r\n");
 send(client, buf, strlen(buf), 0);
 sprintf(buf, "\r\n");
 send(client, buf, strlen(buf), 0);
 sprintf(buf, "<HTML><TITLE>Not Found</TITLE>\r\n");
 send(client, buf, strlen(buf), 0);
 sprintf(buf, "<BODY><P>The server could not fulfill\r\n");
 send(client, buf, strlen(buf), 0);
 sprintf(buf, "your request because the resource specified\r\n");
 send(client, buf, strlen(buf), 0);
 sprintf(buf, "is unavailable or nonexistent.\r\n");
 send(client, buf, strlen(buf), 0);
 sprintf(buf, "</BODY></HTML>\r\n");
 send(client, buf, strlen(buf), 0);
}
```

serve_file

```
/**********************************************************************/
/* Send a regular file to the client.  Use headers, and report
 * errors to client if they occur. * Parameters: a pointer to a file structure |
void serve_file(int client, const char *filename)
{
 FILE *resource = NULL;
 int numchars = 1;
 char buf[1024];

 //确保 buf 里面有东西，能进入下面的 while 循环
 buf[0] = 'A'; buf[1] = '\0';
 //循环作用是读取并忽略掉这个 http 请求后面的所有内容
 while ((numchars > 0) && strcmp("\n", buf))  /* read & discard headers */
  numchars = get_line(client, buf, sizeof(buf));

 //打开这个传进来的这个路径所指的文件
 resource = fopen(filename, "r");
 if (resource == NULL)
  not_found(client);
 else
 {
  //打开成功后，将这个文件的基本信息封装成 response 的头部(header)并返回
  headers(client, filename);
  //接着把这个文件的内容读出来作为 response 的 body 发送到客户端
  cat(client, resource);
 }
 fclose(resource);
}
```

startup

```
/**********************************************************************/
/* This function starts the process of listening for web connections
 * on a specified port.  If the port is 0, then dynamically allocate a * port ar
int startup(u_short *port)
{
 int httpd = 0;
 //sockaddr_in 是 IPV4的套接字地址结构。定义在<netinet/in.h>,参读《TLPI》P1202
 struct sockaddr_in name;

 //socket()用于创建一个用于 socket 的描述符，函数包含于<sys/socket.h>。参读《TLPI》P115
 //这里的PF_INET其实是与 AF_INET同义，具体可以参读《TLPI》P946
 httpd = socket(PF_INET, SOCK_STREAM, 0);
 if (httpd == -1)
  error_die("socket");

 memset(&name, 0, sizeof(name));
 name.sin_family = AF_INET;
 //htons(), ntohs() 和 htonl()包含于<arpa/inet.h>, 参读《TLPI》P1199
 //将*port 转换成以网络字节序表示的16位整数
 name.sin_port = htons(*port);
 //INADDR_ANY是一个 IPV4通配地址的常量，包含于<netinet/in.h>
 //大多实现都将其定义成了0.0.0.0 参读《TLPI》P1187
 name.sin_addr.s_addr = htonl(INADDR_ANY);

 //bind()用于绑定地址与 socket。参读《TLPI》P1153
 //如果传进去的sockaddr结构中的 sin_port 指定为0，这时系统会选择一个临时的端口号
 if (bind(httpd, (struct sockaddr *)&name, sizeof(name)) < 0)
  error_die("bind");

 //如果调用 bind 后端口号仍然是0，则手动调用getsockname()获取端口号
 if (*port == 0)  /* if dynamically allocating a port */
 {
  socklen_t namelen = sizeof(name);
  //getsockname()包含于<sys/socker.h>中，参读《TLPI》P1263
  //调用getsockname()获取系统给 httpd 这个 socket 随机分配的端口号
  if (getsockname(httpd, (struct sockaddr *)&name, &namelen) == -1)
   error_die("getsockname");
  *port = ntohs(name.sin_port);
 }
 //最初的 BSD socket 实现中，backlog 的上限是5.参读《TLPI》P1156
 if (listen(httpd, 5) < 0)
error_die("listen");
```

```
 return(httpd);
}
```

unimplemented

```
/**********************************************************************/
/* Inform the client that the requested web method has not been
 * implemented. * Parameter: the client socket *//*****************************
void unimplemented(int client)
{
 char buf[1024];

 sprintf(buf, "HTTP/1.0 501 Method Not Implemented\r\n");
 send(client, buf, strlen(buf), 0);
 sprintf(buf, SERVER_STRING);
 send(client, buf, strlen(buf), 0);
 sprintf(buf, "Content-Type: text/html\r\n");
 send(client, buf, strlen(buf), 0);
 sprintf(buf, "\r\n");
 send(client, buf, strlen(buf), 0);
 sprintf(buf, "<HTML><HEAD><TITLE>Method Not Implemented\r\n");
 send(client, buf, strlen(buf), 0);
 sprintf(buf, "</TITLE></HEAD>\r\n");
 send(client, buf, strlen(buf), 0);
 sprintf(buf, "<BODY><P>HTTP request method not supported.\r\n");
 send(client, buf, strlen(buf), 0);
 sprintf(buf, "</BODY></HTML>\r\n");
 send(client, buf, strlen(buf), 0);
}
```

main

```
int main(void)
{
 int server_sock = -1;
 u_short port = 0;
 int client_sock = -1;
 //sockaddr_in 是 IPV4的套接字地址结构。定义在<netinet/in.h>,参读《TLPI》P1202
 struct sockaddr_in client_name;
 socklen_t client_name_len = sizeof(client_name);
 //pthread_t newthread;

 server_sock = startup(&port);
```

```
 printf("httpd running on port %d\n", port);

 while (1)
 {
  //阻塞等待客户端的连接，参读《TLPI》P1157
  client_sock = accept(server_sock,
                       (struct sockaddr *)&client_name,
                       &client_name_len);
  if (client_sock == -1)
   error_die("accept");
  accept_request(client_sock);
 /*if (pthread_create(&newthread , NULL, accept_request, client_sock) != 0)
   perror("pthread_create");*/ }

 close(server_sock);

 return(0);
}
```