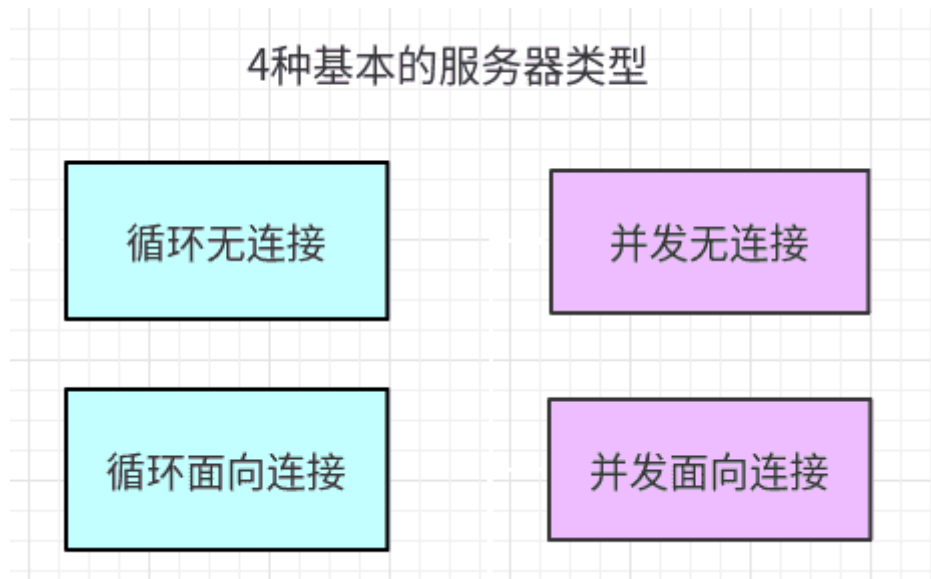


实验内容

开发一个处理一个HTTP请求的Web服务器。这个Web服务器应该接受并解析HTTP请求，然后从服务器的文件系统获取所请求的文件，按照HTTP响应消息创建一个响应消息。如果请求的文件不在服务器中，则服务器向客户端发送 404 Not Found 报文。

实验过程

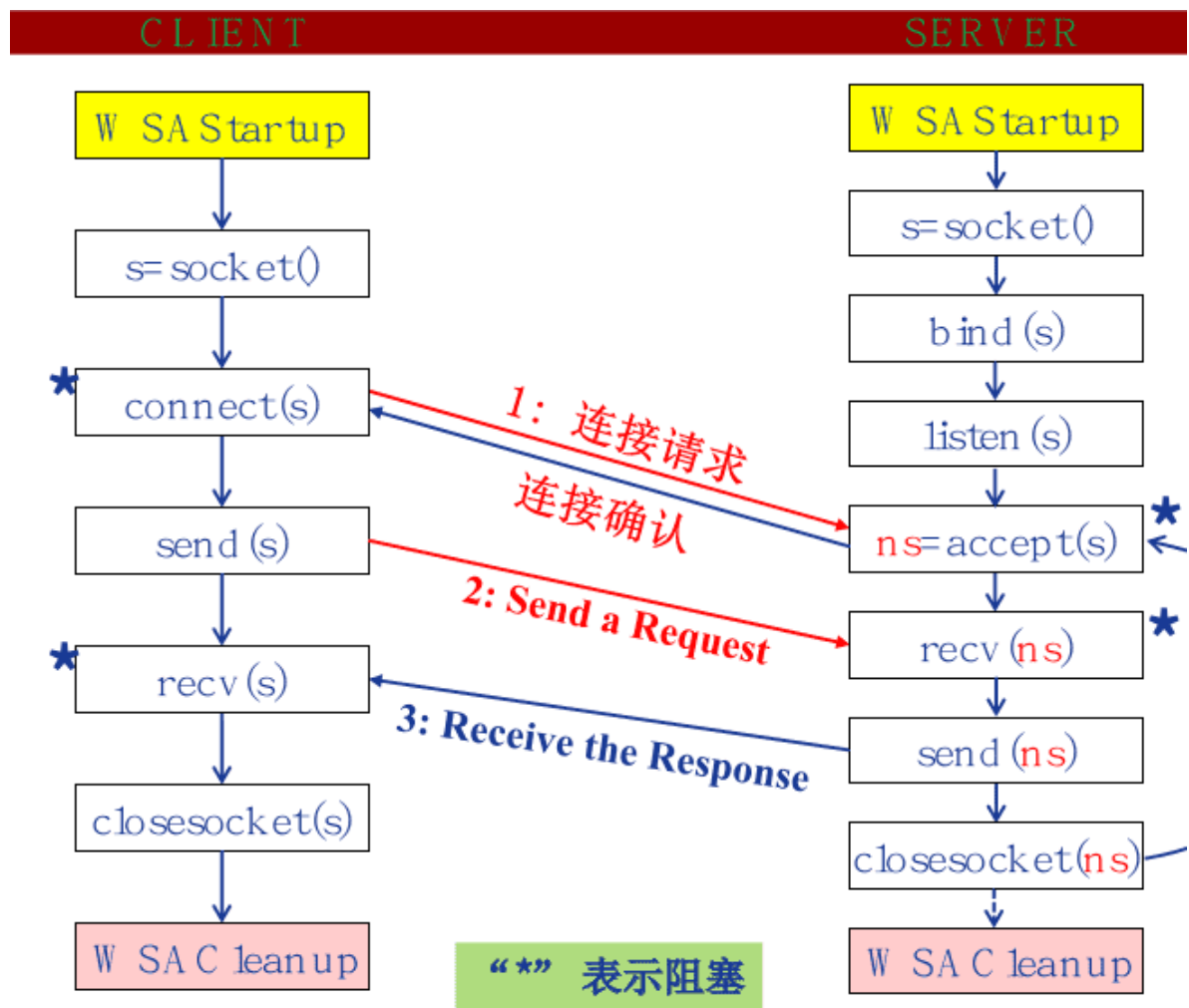
基本的服务器类型有四种：



其实循环跟并发区别在于是否引进了线程，无连接跟面向连接区别在于传输层用的是 TCP 还是 UDP。

API调用基本流程

下面的PPT是 Windows 下 Socket 一些基于 TCP 的 API 调用基本流程



因为现在还没有学习到线程编程的知识，所以这次写的只是一个循环的面向连接的 Web 服务器，就当是熟悉熟悉一下 Socket 编程吧。

server.c

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <signal.h>
#include <string.h>

/*
@变量声明
*/
int server_sockfd, client_sockfd;
int server_len, client_len;
struct sockaddr_in server_address;
struct sockaddr_in client_address;

/*
@HTTP响应消息格式
*/
```

```

*/
char *http_res_tmpl = "HTTP/1.1 200 OK\r\n"
    "Server: my localhost 127.0.0.1\r\n"
    "Accept-Ranges: bytes\r\n"
    "Content-Length: %d\r\n"
    "Connection: close\r\n"
    "Content-Type: %s\r\n\r\n";

/*
@宏定义
*/
#define bufmaxlen 1024 //1KB
#define urlilen 50

/*
@brief:生成服务器端的socket描述符
@para:void
@return:int
*/
int get_sever_sockfd(void)
{
    return socket(AF_INET,SOCK_STREAM,0);
}

/*
@brief:设置服务端的sockaddr_in结构，包括协议族，本机地址，端口号，地址长度
@para:void
@return:none
*/
void name_server_socker_addr(void)
{
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
    server_address.sin_port = htons(8080);
    server_len = sizeof(server_address);
}

/*
@brief:HTTP发送文件
@para:客户端套接字 发送的内容(文件等)
@return:none
*/
void http_send(int client_sockfd,char *content)
{
    char header[bufmaxlen],text[bufmaxlen];
    int len = strlen(content);
    sprintf(header,http_res_tmpl,len,"text/html");
    len = sprintf(text,"%s%s",header,content);
    send(client_sockfd,text,len,0);
}

/*
@brief:信号异常处理

```

```

@para:sign
@return:none
*/
void handle_signal(int sign){
    close(server_sockfd);
    fputs("\nsignal interrupt,safe exit\n",stdout);
    exit(0);
}

/*
@brief:将要发送的文件放进缓冲区
@para:文件名 发送缓冲区 发送缓冲区大小
@return:0成功转换 -1无法打开指定文件
*/
int convert_whatfileToSend_buf(char *filename,char *buf,int size)
{
    FILE *t;
    t=fopen(filename,"rb");
    if (!t)
    {
        printf("open file fail\n");
        return -1;
    }
    memset(buf,0,size);
    while (fread(buf,1,size,t));//一个循环一个字节一个字节的读
    fread(buf,size,1,t);
    fclose(t);
    return 0;
}

int main()
{
    signal(SIGINT,handle_signal);//按下Ctrl+C的时候的处理

    /*获取服务器的套接字描述符*/
    server_sockfd=get_sever_sockfd();

    name_server_socker_addr();

    /*绑定*/
    int bindsta=bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
    if (bindsta== -1)
    {
        printf("bind fail\n");
        return -1;
    }

    /*监听*/
    int listensta=listen(server_sockfd,1);
    if(listensta== -1)
    {
        printf("listen error\n");
        return -3;
    }

    while (1)

```

```

{
    client_len=sizeof(client_address);
    client_sockfd=accept(server_sockfd,(struct sockaddr*)&client_address,&client_len);//这里天坑
    if(client_sockfd<0)
    {
        printf("accept error\n");
        return -4;
    }

    char buf[bufmaxlen];
    memset(buf,0,sizeof(buf));

    /*读取客户端的消息并打印出来*/
    read(client_sockfd, buf, bufmaxlen);
    printf("%s\n",buf);

    /*从接受到的客户端的请求报文中截取URL，暂时都是GET请求*/
    char* urlbegpos=strstr(buf,"GET /")+sizeof("GET /")-1;
    char* urlendpos=strstr(buf,"HTTP")-1;
    char urlbuf[urllen];
    memset(urlbuf,'0',urllen);
    for (int i=0; urlbegpos!= urlendpos; urlbegpos++,i++)
    {
        urlbuf[i]=*urlbegpos;
    }

    /*匹配URL*/
    if (strstr(urlbuf,"404.html"))
    {
        int filesta=convert_whatfileToSend_buf("404.html",buf,bufmaxlen);
        if(filesta== -1)return -6;
    }
    else
    {
        int filesta=convert_whatfileToSend_buf("HelloWorld.html",buf,bufmaxlen);
        if(filesta== -1)return -6;
    }

    /*根据不同请求发送不同的内容*/
    http_send(client_sockfd,buf);

    /*关闭客户端的套接字*/
    close(client_sockfd);
}

printf("abnormal exit\n");

/*关闭服务器的套接字*/
close(server_sockfd);

return 0;
}

```

HelloWorld.html

```
<html>
<head><title>default page</title></head>
<body>
  <p>Hello World!</p>
</body>
</html>
```

404.html

```
<html>
<head><title>404</title></head>
<body>
  <p>404 Not Found</p>
</body>
</html>
```

server.c里面用到的一些API函数原型

sockaddr_in 结构

❖ 已定义结构 `sockaddr_in`:

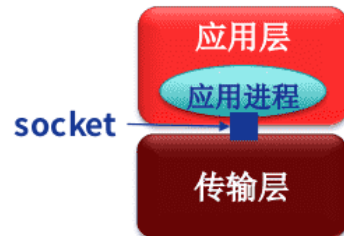
```
struct sockaddr_in
{
    u_char sin_len;           /* 地址长度 */
    u_char sin_family;        /* 地址族 (TCP/IP: AF_INET) */
    u_short sin_port;         /* 端口号 */
    struct in_addr sin_addr;   /* IP 地址 */
    char sin_zero[8];         /* 未用 (置0) */
}
```

❖ 使用 **TCP/IP** 协议簇的网络应用程序声明端点地址变量时，使用结构 `sockaddr_in`

socket 函数

socket

```
sd = socket(protofamily,type,proto);
```



- ❖ 创建套接字
- ❖ 操作系统返回套接字描述符 (sd)
- ❖ 第一个参数(协议族): `protofamily = PF_INET` (TCP/IP)
- ❖ 第二个参数(套接字类型):
 - `type = SOCK_STREAM, SOCK_DGRAM` or `SOCK_RAW` (TCP/IP)
- ❖ 第三个参数(协议号): 0 为默认
- ❖ 例: 创建一个流套接字的代码段

```
struct protoent *p;  
p=getprotobyname("tcp");  
SOCKET sd=socket(PF_INET,SOCK_STREAM,p->p_proto);
```

bind 函数

bind

```
int bind(sd,localaddr,addrlen) ;
```

- ❖ 绑定套接字的本地端点地址
 - IP地址+端口号
- ❖ 参数:
 - 套接字描述符 (sd)
 - 端点地址 (localaddr)
 - 结构 `sockaddr_in`
- ❖ 客户程序一般不必调用bind函数
- ❖ 服务器端?
 - 熟知端口号
 - IP地址?



INADDR_ANY 地址通配符

Socket API函数

❖ 考虑如下情形：



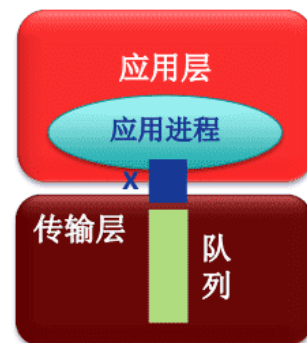
- ❖ 服务器应该绑定哪个地址？
- ❖ 问题？
- ❖ 解决方案
 - 地址通配符： **INADDR_ANY**

listen 函数

listen

```
int listen(sd, queue size);
```

- ❖ 置服务器端的流套接字处于监听状态
 - 仅服务器端调用
 - 仅用于面向连接的流套接字
- ❖ 设置连接请求队列大小（queue size）
- ❖ 返回值：
 - 0：成功
 - SOCKET_ERROR：失败

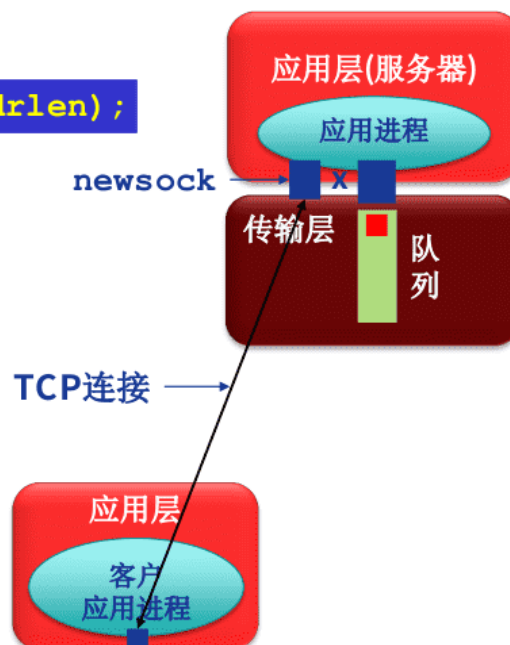


accept 函数

accept

```
newsock = accept(sd, caddr, caddrlen);
```

- ❖ 服务程序调用accept函数从处于监听状态的流套接字sd的客户连接请求队列中取出排在最前的一个客户请求，并且创建一个新的套接字来与客户套接字创建连接通道
 - 仅用于TCP套接字
 - 仅用于服务器
- ❖ 利用新创建的套接字（newsock）与客户通信



send 函数

send, sendto

```
send(sd, *buf, len, flags);
```

```
sendto(sd, *buf, len, flags, destaddr, addrlen);
```

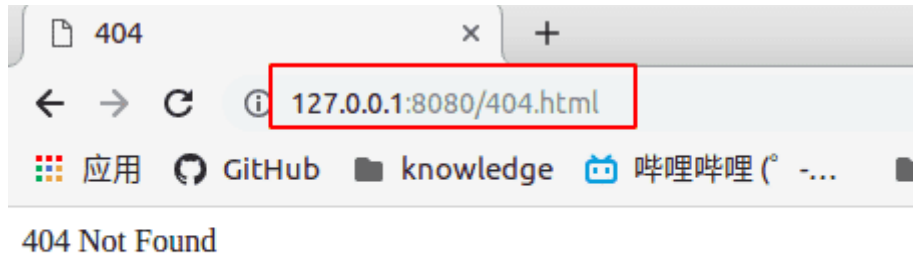
- ❖ send函数TCP套接字（客户与服务器）或调用了connect函数的UDP客户端套接字
- ❖ sendto函数用于UDP服务器端套接字与未调用connect函数的UDP客户端套接字

实验结果截图

终端运行服务器



打开浏览器，随便输入哪张网卡的ip地址都可以，然后注意是到8080端口并且请求 404.html

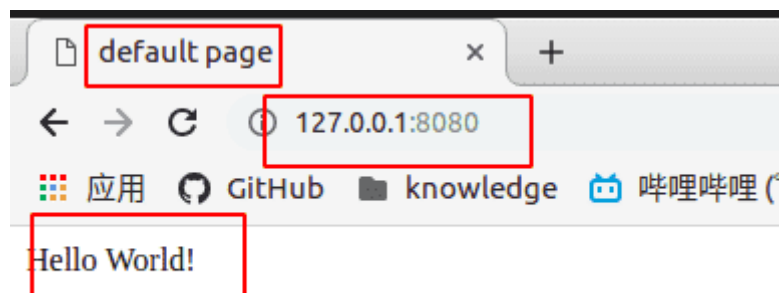


在服务器端打印出来的URL

```
copyright@copyright-Vostro-3559:~/桌面/作业1-Web服务器/自己的作业$ ./server
GET /404.html HTTP/1.1
Host: 127.0.0.1:8080
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/73.0.3683.86 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8

GET /favicon.ico HTTP/1.1
Host: 127.0.0.1:8080
Connection: keep-alive
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/73.0.3683.86 Safari/537.36
Accept: image/webp,image/apng,image/*,*/*;q=0.8
Referer: http://127.0.0.1:8080/404.html
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8
```

接着输入下面这个网址



服务器打印出来的URL是 /

```
GET / HTTP/1.1
Host: 127.0.0.1:8080
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/73.0.3683.86 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8
```

在服务器端按下 `Ctrl+C` 会看到信号处理函数的输出内容

```
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML
o) Chrome/73.0.3683.86 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/
png,*/*;q=0.8,application/signed-exchange;v=b3
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8

GET /favicon.ico HTTP/1.1
Host: 127.0.0.1:8080
Connection: keep-alive
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML
o) Chrome/73.0.3683.86 Safari/537.36
Accept: image/webp,image/apng,image/*,*/*;q=0.8
Referer: http://127.0.0.1:8080/
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8

^C
signal interrupt,safe exit
```

然后马上利用 `netstat -nt | grep 8080` 命令查看端口的情况

```
signal interrupt,safe exit
copyright@copyright-Vostro-3559:~/桌面/作业1-Web服务器/自己的作业$ netstat -nt | g
rep 8080
tcp        1      0 127.0.0.1:40646          127.0.0.1:8080          CLOSE_WAIT
tcp        0      0 127.0.0.1:8080          127.0.0.1:40646        FIN_WAIT2
```

再隔一段时间再看

```
copyright@copyright-Vostro-3559:~/桌面/作业1-Web服务器/自己的作业$ netstat -nt | g
rep 8080
tcp        1      0 127.0.0.1:40646          127.0.0.1:8080          CLOSE_WAIT
tcp        0      0 127.0.0.1:8080          127.0.0.1:40646        FIN_WAIT2
copyright@copyright-Vostro-3559:~/桌面/作业1-Web服务器/自己的作业$ netstat -nt | g
rep 8080
tcp        1      0 127.0.0.1:40646          127.0.0.1:8080          CLOSE_WAIT
copyright@copyright-Vostro-3559:~/桌面/作业1-Web服务器/自己的作业$ netstat -nt | g
rep 8080
copyright@copyright-Vostro-3559:~/桌面/作业1-Web服务器/自己的作业$
```

注意的点

两个命令查看端口占用

```
netstat -nt | grep 8080
sudo lsof -i:8080      #需要root权限
```

memset函数

函数原型，注意到中间是 `int` 类型的参数。

```
void *memset(void *s, int ch, size_t n);
```

一直我都把这个函数当成是将某个数组置0用的办法，但是如果用于字符数组的话，记得中间参数，要换成 '0'。一开始没注意的时候，直接将 ch=0，然后在调试程序的时候发现输出的并不是我想要的“置零”的效果。

指针做参数

在函数 `int convert_whatfileToSend_buf(char *filename, char *buf, int size)` 中在用 `memset` 函数对 `buf` 进行处理的时候，使用了 `sizeof(buf)` 去当做 `memset` 函数的第三个参数。但是在这里是对一个指针求大小，所以只有4个字节的大小。

fopen

用 `fopen` 打开 `HelloWorld.html` 的时候，一开始参数设置的是 `r+`，发送过去的消息是乱码，后来再加一个以二进制方式打开文件的选项，就不会了，也即 `rb`。

accept失败

```
client_sockfd=accept(server_sockfd,(struct sockaddr*)&client_address,&client_len);
```

失败的原因就是第三个参数。。。真的是没注意到一个取地址符号。。。

参考

1. `bind` 函数调用失败原因

- [链接1](#)
- [链接2](#)

`bind` 普遍遭遇的问题是试图绑定一个已经在使用的端口。该陷阱是也许没有活动的套接字存在，但仍然禁止绑定端口（`bind` 返回 `EADDRINUSE`），它由 TCP 套接字状态 `TIME_WAIT` 引起。该状态在套接字关闭后约保留 2 到 4 分钟。在 `TIME_WAIT` 状态退出之后，套接字被删除，该地址才能被重新绑定而不出问题。可以给套接字应用 `SO_REUSEADDR` 套接字选项，以便端口可以马上重用。

2. `lsof` 用法

- [链接](#)

3. 关于 `close` 跟 `closesocket` 两个函数的区别

- [链接](#)

`close()` is a *nix function. It will work on any file descriptor, and sockets in *nix are an example of a file descriptor, so it would correctly close sockets as well.

`closesocket()` is a Windows-specific function, which works specifically with sockets. Sockets on Windows do not use *nix-style file descriptors, `socket()` returns a handle to a kernel object instead, so it must be closed with `closesocket()`.

大意就是 `close` 是 unix 下用的而 `closesocket` 是 windows 下用的。

其中的一句 `sockets in *nix are an example of a file descriptor`，感觉讲的挺不错的。

4.strstr函数的用法

- [链接](#)

```
//原型: const char * strstr ( const char * str1, const char * str2);  
char * strstr ( char * str1, const char * str2);  
  
//参数: str1, 待查找的字符串指针;  
//      str2, 要查找的字符串指针。  
  
//说明: 在str1中查找匹配str2的子串, 并返回指向首次匹配时的第一个元素指针。如果没有找到, 返回NULL  
指针。
```

5.Socket编程

- [链接](#)

6.C++实现简单Web服务器

- [链接](#)

这个博客讲一些原理挺详细的，下面是截取一段对HTTP请求消息格式的讲解（客户端）

第一行就是请求行，请求行的格式是这样的：请求方法+空格+URL+空格+协议版本+\r+\n 这里的请求方法是GET，URL是/(在这里，URL就相当于资源的路径，若在网址栏输入的是<http://localhost:8888/hello.html>的话，这里浏览器发送过来的URL就是/hello.html)，协议版本是HTTP/1.1（现在多数协议版本都是这个）。

第二行到最后一行都是请求头部，请求头部的格式是这样的：头部字段：+空格+数值+\r+\n 然后多个头部子段组织起来就是请求头部，在最后的头部字段的格式中需要有两个换行符号，最后一行的格式是：头部字段：+空格+数值+\r+\n+\r+\n 因为在后面还要跟着请求数据，为了区分请求数据和请求头的结束，就多了一个换行符。

7.sockaddr_in 跟 sockaddr_un 的区别

- [链接](#)

`sockaddr_in` 主要用于不同主机之间的socket编程

`sockaddr_un` 主要用于同一个主机中的本地Local socket

实验体会

- 这次需要调用到很多个 API 函数，有些只能用于客户端，有些只能用于服务器的；有的变量是一些默认的值，一个宏定义就填上去。如果先从小部分的这些函数入手其实很细，会比较零散，应该是先熟悉一个服务器调用 API 的总体路线，然后再去找对应的函数。
- 记得检查一些函数的返回值是否正确，比如是否打开文件成功，是否成功绑定上了端口号等等。

可选练习

- 目前这个 Web 服务器一次只处理一个 HTTP 请求。请实现一个能够同时处理多个请求的多线程服务器。使用线程，首先创建一个主线程，在固定端口监听客户端请求。当从客户端收到 TCP 连接请求时，它将通过另一个端口建立 TCP 连接，并在另外的单独线程中为客户端请求提供服务。这样在每个请求/响应对应的独立线程中将有一个独立的 TCP 连接。
- 不使用浏览器，编写自己的 HTTP 客户端来测试你的服务器。你的客户端将使用一个 TCP 连接用于连接到服务器，向服务器发送 HTTP 请求，并将服务器响应显示出来。你可以假定发送的 HTTP 请求将使用 GET 方法。客户端应使用命令行参数指定服务器 IP 地址或主机名，服务器正在监听的端口，以及被请求对象在服务器上的路径。一下是运行客户端的输入命令格式。

```
client.py server_host server_port filename
```