# 实验六 HTTP Client/HTTP Server 实验

### 一、实验目的

初步掌握 ESP8266 的 Wi-Fi 相关的库。学习 ESP8266 作为 HTTP Client 和 HTTP Server

的基本技能。 **二、背景知识**

1. HTTP

1.1 HTTP 简介

HTTP 协议是 Hyper Text Transfer Protocol 的缩写，简称超文本传输协议，用于从 WWW

服务器传输文本到本地浏览器的传送协议。

HTTP 是一个基于 TCP/IP 通信协议来传递数据，浏览器作为 HTTP 客户端通过 URL 向 HTTP 服务端即 WEB 服务器发送所有请求。WEB 服务器根据接收到的请求后，向客户端发 送响应信息。



HTTP 协议作为 TCP/IP 模型中应用层的协议，承载于 TCP 协议之上，有时也承载于 TLS

或者 SSL 协议层之上，这个时候就是 HTTPS。

HTTP 是一个应用层协议，由请求和响应构成，是一个标准的客户端服务器模型。HTTP 默认的端口号是 80，HTTPS 的端口号是 443。

浏览网页是 HTTP 主要应用，但不代表只用于网页浏览。HTTP 只是一种协议，只要通 信双方遵守这个协议，HTTP 就能用。

### HTTP 特点

1) 简单快速：客户端向服务端请求服务时，只需要传送请求方法和路径。HTTP 协议简 单，使得 HTTP 服务器的程序规模小，因而通信速度快；

2) 灵活：HTTP 允许传输任意类型的数据对象，正在传输的类型由 Content-Type 加以标 记。

3) 连接问题

 HTTP0.9 和 1.0 使用非持续连接：限制每次连接都只处理一个请求，服务端处理完客 户的请求，并收到客户的应答后，即断开连接；

 HTTP1.1 使用持续连接：不必为每个 web 对象创建一个新连接，一个连接可以传送多 个对象，节省传输时间；

4) 无状态：HTTP 协议是无状态。对于事务处理没有记忆能力，如果需要处理前面信 息，则必须重传，这样可能导致每次连接传送的数据量增大。

### HTTP 工作流程

一次 HTTP 操作称为一个事务，工作流程可分为 4 步：

1) 首先客户端 client 与服务端 server 建立连接。

2) 建立连接后，客户端发送一个请求给服务端，请求方法的格式：统一资源标识符（URL）、

HTTP 协议版本号、请求头、请求内容等；

3) 服务端接收到请求后，给予相应的响应信息，其格式为：状态行（包括协议版本、成功或 者失败代码）、服务器信息、实体信息等；

4) 客户端接收到服务端返回的信息，通过浏览器显示在用户的显示屏上，然后客户端与服 务端断开连接； 以上四步骤，只要其中一步出现错误，那么就会产生错误信息返回给客户端。

### HTTP 请求

客户端发送一个 HTTP 请求到服务器，请求信息包括以下格式：

* + - **请求行**（request line）

 **请求头部**（header）

* + - **空行** (empty line)
    - **请求数据** (request body)



请求行以一个方法符号开头，以空格分开，后面跟着请求的 URI 和协议的版本。

### 1.4.1 Get 请求

请求例子，使用 Charles 抓取的 request：

GET /562f25980001b1b106000338.jpg HTTP/1.1

Host img.mukewang.com

User-Agent Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.106 Safari/537.36

Accept image/webp,image/\*,\*/\*;q=0.8 Referer [htt](http://www.imooc.com/)p[://www.imooc.com/](http://www.imooc.com/) Accept-Encoding gzip, deflate, sdch Accept-Language zh-CN,zh;q=0.8

1) 第一部分：请求行，用来说明请求类型,要访问的资源以及所使用的 HTTP 版本.

GET 说明请求类型为 GET,[/562f25980001b1b106000338.jpg]为要访问的资源，该 行的最后一部分说明使用的是 HTTP1.1 版本。

2) 第二部分：请求头部，紧接着请求行（即第一行）之后的部分，用来说明服务器 要使用的附加信息:

从第二行起为请求头部，HOST 将指出请求的目的地.User-Agent,服务器端和客户 端脚本都能访问它,它是浏览器类型检测逻辑的重要基础.

该信息由你的浏览器来定义,并且在每个请求中自动发送等等 3) 第三部分：空行，请求头部后面的空行是必须的:

即使第四部分的请求数据为空，也必须有空行。

4) 第四部分：请求数据也叫主体，可以添加任意的其他数据。

这个例子的请求数据为空。 **1.4.2 POST 请求**

请求例子，使用 Charles 抓取的 request：

POST / HTTP1.1

Host:[www.wrox.com](http://www.wrox.com/)

User-Agent:Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022)

Content-Type:application/x-www-form-urlencoded Content-Length:40

Connection: Keep-Alive

name=Professional%20Ajax&publisher=Wiley

1. 第一部分：请求行，第一行明了是 post 请求，以及 http1.1 版本。

2. 第二部分：请求头部，第二行至第六行。

3. 第三部分：空行，第七行的空行。

4. 第四部分：请求数据，第八行。 **1.5 HTTP Response 响应信息**

一般情况下，服务端接收并处理客户端发过来的请求会返回一个 HTTP 的响应信息。HTTP 响应也由四个部分组成，分别是:

 **状态行**

 **消息报头**

 **空行**

 **响应正文**

1) 第一部分：状态行，由 HTTP 协议版本号， 状态码， 状态消息 三部分组成。 第一行为状态行，（HTTP/1.1）表明 HTTP 版本为 1.1 版本，状态码为 200，状态消息为（ok） 2) 第二部分：消息报头，用来说明客户端要使用的一些附加信息： 第二行和第三行为消息报头

Date:生成响应的日期和时间；Content-Type:指定了 MIME 类型的 HTML(text/html),编码类型 是 UTF-8

3) 第三部分：空行，消息报头后面的空行是必须的 4. 第四部分：响应正文，服务器返回给 客户端的文本信息。

空行后面的 html 部分为响应正文。

**1.6 HTTP 状态码** 状态代码有三位数字组成，第一个数字定义了响应的类别，共分五种类别:

1xx：指示信息–表示请求已接收，继续处理

2xx：成功–表示请求已被成功接收、理解、接受

3xx：重定向–要完成请求必须进行更进一步的操作

4xx：客户端错误–请求有语法错误或请求无法实现

5xx：服务器端错误–服务器未能实现合法的请求

常见状态码：

200 OK //客户端请求成功

400 Bad Request //客户端请求有语法错误，不能被服务器所理解

401 Unauthorized //请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使 用

403 Forbidden //服务器收到请求，但是拒绝提供服务

404 Not Found //请求资源不存在，eg：输入了错误的 URL 500 Internal Server Error //服务器发生不可预期的错误

503 Server Unavailable //服务器当前不能处理客户端的请求，一段时间后可能恢复正常

### 1.7 实例

下面实例是一点典型的使用 GET 来传递数据的实例： 客户端请求：

GET /hello.txt HTTP/1.1

User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3 [Host: www.](http://www.example.com/)exam[ple.com](http://www.example.com/)

Accept-Language: en, mi

服务端响应:

HTTP/1.1 200 OK

Date: Mon, 27 Jul 2009 12:28:53 GMT

Server: Apache

Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT ETag: "34aa387-d-1568eb00"

Accept-Ranges: bytes Content-Length: 51 Vary: Accept-Encoding Content-Type: text/plain

输出结果：

Hello World! My payload includes a trailing CRLF.

# HTTP Client

### 2.1 HTTP 请求方法

|  |  |  |
| --- | --- | --- |
| **序号** | **方法** | **描述** |
| 1 | GET | 请求指定的页面信息，并返回实体主体。 |
| 2 | HEAD | 类似于 GET 请求，只不过返回的响应中没有具体的内容，用于获取报头 |
| 3 | POST | 向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。  POST 请求可能会导致新的资源的建立和/或已有资源的修改。 |
| 4 | PUT | 从客户端向服务器传送的数据取代指定的文档的内容。 |
| 5 | DELETE | 请求服务器删除指定的页面。 |
| 6 | CONNECT | HTTP/1.1 协议中预留给能够将连接改为管道方式的代理服务器。 |
| 7 | OPTIONS | 允许客户端查看服务器的性能。 |

|  |  |  |
| --- | --- | --- |
| 8 | TRACE | 回显服务器收到的请求，主要用于测试或诊断。 |
| 9 | PATCH | 是对 PUT 方法的补充，用来对已知资源进行局部更新 。 |

根据 HTTP 标准，HTTP 请求可以使用多种请求方法。 HTTP1.0 定义了三种请求方法： GET, POST 和 HEAD 方法。

HTTP1.1 新增了六种请求方法：OPTIONS、PUT、PATCH、DELETE、TRACE 和 CONNECT 方 法。

**2.2 实例**

通过 TCP client 包装 HTTP 请求协议去调用心知天气接口获取天气信息。

#include <ESP8266WiFi.h>

#include <ArduinoJson.h>

//以下三个定义为调试定义

#define DebugBegin(baud\_rate) Serial.begin(baud\_rate)

#define DebugPrintln(message) Serial.println(message)

#define DebugPrint(message) Serial.print(message)

const char\* ssid

ssid

= "GreenIoT";

// XXXXXX -- 使用时请修改为当前你的 wifi

const char\* password = " \*\*\*\* "; // XXXXXX -- 使用时请修改为当前你的 wifi 密

码

const char\* host = "api.seniverse.com";

const char\* APIKEY = "wcmquevztdy1jpca"; //API KEY const char\* city = "guangzhou";

const char\* language = "zh-Hans";//zh-Hans 简体中文 会显示乱码

const unsigned long BAUD\_RATE = 115200;

speed

const unsigned long HTTP\_TIMEOUT = 5000; server

const size\_t MAX\_CONTENT\_SIZE = 1000; response

// serial connection

// max respone time from

// max size of the HTTP

// 从此网页中提取的数据的类型

struct WeatherData {

char city[16];//城市名称

char weather[32];//天气介绍（多云...）

char temp[16];//温度

char udate[32];//更新时间

};

WiFiClient client;

char response[MAX\_CONTENT\_SIZE]; char endOfHeaders[] = "\r\n\r\n";

void setup() {

// put your setup code here, to run once: WiFi.mode(WIFI\_STA); //设置 esp8266 工作模式 DebugBegin(BAUD\_RATE);

DebugPrint("Connecting to ");//提示 DebugPrintln(ssid);

WiFi.begin(ssid, password); //连接 wifi WiFi.setAutoConnect(true);

while (WiFi.status() != WL\_CONNECTED) {

//这个函数是 wifi 连接状态，返回 wifi 链接状态

delay(500); DebugPrint(".");

}

DebugPrintln(""); DebugPrintln("WiFi connected"); delay(500);

DebugPrintln("IP address: "); DebugPrintln(WiFi.localIP());//WiFi.localIP()返回 8266 获得的 ip 地址 client.setTimeout(HTTP\_TIMEOUT);

}

void loop() {

// put your main code here, to run repeatedly:

//判断 tcp client 是否处于连接状态，不是就建立连接

while (!client.connected()){

if (!client.connect(host, 80)){ DebugPrintln("connection...."); delay(500);

}

}

//发送 http 请求 并且跳过响应头 直接获取响应 body

if (sendRequest(host, city, APIKEY) && skipResponseHeaders()) {

//清除缓冲 clrEsp8266ResponseBuffer();

//读取响应数据

readReponseContent(response, sizeof(response));

WeatherData weatherData;

if (parseUserData(response, &weatherData)) { printUserData(&weatherData);

}

}

delay(5000);//每 5s 调用一次

}

/\*\*

\* @发送 http 请求指令

\*/

bool sendRequest(const char\* host, const char\* cityid, const char\* apiKey) {

// We now create a URI for the request

//心知天气 发送 http 请求

String GetUrl = "/v3/weather/now.json?key="; GetUrl += apiKey;

GetUrl += "&location="; GetUrl += city;

GetUrl += "&language="; GetUrl += language;

// This will send the request to the server client.print(String("GET ") + GetUrl + " HTTP/1.1\r\n" +

"Host: " + host + "\r\n" + "Connection: close\r\n\r\n");

DebugPrintln("create a request:"); DebugPrintln(String("GET ") + GetUrl + " HTTP/1.1\r\n" +

"Host: " + host + "\r\n" + "Connection: close\r\n");

delay(1000); return true;

}

/\*\*

\* @Desc 跳过 HTTP 头，响应正文的开头

\*/

bool skipResponseHeaders() {

// HTTP headers end with an empty line bool ok = client.find(endOfHeaders);

if (!ok) {

DebugPrintln("No response or invalid response!");

}

return ok;

}

/\*\*

\* @Desc 从 HTTP 服务器响应中读取正文

\*/

void readReponseContent(char\* content, size\_t maxSize) { size\_t length = client.readBytes(content, maxSize); delay(100);

DebugPrintln("Get the data from Internet!"); content[length] = 0;

DebugPrintln(content); DebugPrintln("Read data Over!"); client.flush();//清除一下缓冲

}

/\*\*

* @Desc 解析数据 Json 解析

\* 数据格式如下：

\* {

* "results": [

\* {

* "location": {
* "id": "WX4FBXXFKE4F",

\* "name": "北京",

* "country": "CN",

\* "path": "北京,北京,中国",

* "timezone": "Asia/Shanghai",
* "timezone\_offset": "+08:00"

\* },

\* "now": {

\* "text": "多云",

\* "code": "4",

* "temperature": "23"

\* },

\* "last\_update": "2017-09-13T09:51:00+08:00"

\* }

\* ]

\*}

\*/

bool parseUserData(char\* content, struct WeatherData\* weatherData) {

// -- 根据需要解析的数据来计算 JSON 缓冲区最佳大小

// 如果你使用 StaticJsonBuffer 时才需要

// const size\_t BUFFER\_SIZE = 1024;

// 在堆栈上分配一个临时内存池

// StaticJsonBuffer<BUFFER\_SIZE> jsonBuffer;

// -- 如果堆栈的内存池太大，使用 DynamicJsonBuffer jsonBuffer 代替

DynamicJsonBuffer jsonBuffer;

JsonObject& root = jsonBuffer.parseObject(content);

if (!root.success()) {

DebugPrintln("JSON parsing failed!"); return false;

}

//复制感兴趣的字符串

strcpy(weatherData->city, root["results"][0]["location"]["name"]); strcpy(weatherData->weather, root["results"][0]["now"]["text"]);

//获取温度、最近更新时间

// -- 这不是强制复制，你可以使用指针，因为他们是指向“内容”缓冲区内，所以你需

要确保

// 当你读取字符串时它仍在内存中

return true;

}

// 打印从 JSON 中提取的数据

void printUserData(const struct WeatherData\* weatherData) { DebugPrintln("Print parsed data :");

DebugPrint("City : "); DebugPrint(weatherData->city);

//打印天气、气温、最近更新时间

}

// 关闭与 HTTP 服务器连接

void stopConnect() { DebugPrintln("Disconnect"); client.stop();

}

void clrEsp8266ResponseBuffer(void){

memset(response, 0, MAX\_CONTENT\_SIZE);

}

//清空

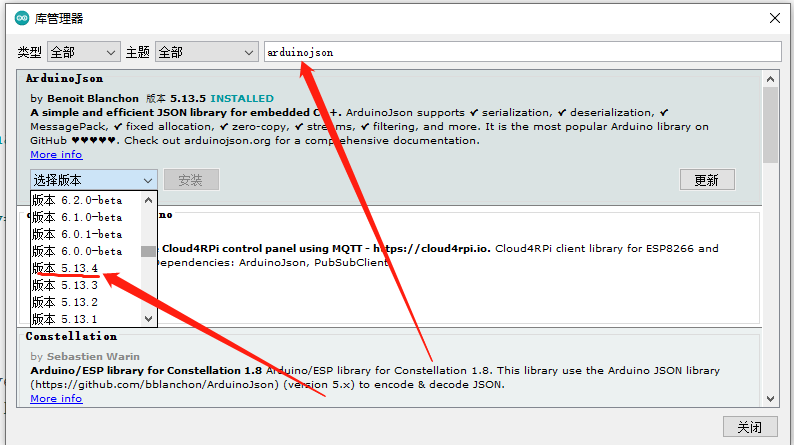
### 

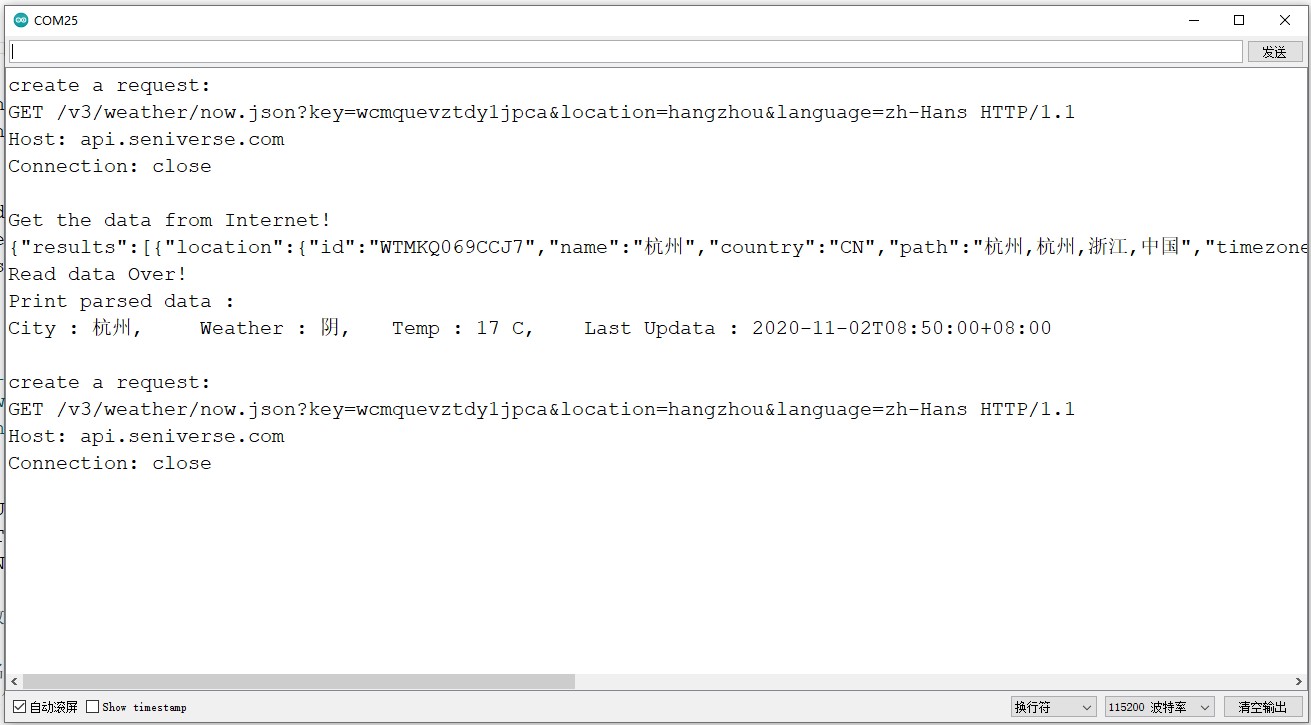
本例用到了 ArduinoJson 库，需要使用 [ArduinoJson](https://github.com/bblanchon/ArduinoJson)；尽量使用 ArduinoJson 5.x 版本，

6.x 版本改变很大，很多方法有改动。

源码解析：

Setup 中配置好 URL，串口参数，以及 HttpClient，并设置了 Client 的请求头。Loop 中， 每隔 1s 去请求一次 get 服务,将获取回来的天气信息通过 JSON 库解释为对应的数值。





# HTTP Server/Web Server

通过 Tcp server 处理 http 请求，需要自己解析请求协议以及判断各种数据，容易出

错。ESP8266WebServer 库是专门将 ESP8266 用作 WebServer 操作。ESP8266WebServer 库不 属于 ESP8266WiFi 库的一部分，需要单独引入.

#include <ESP8266WebServer.h>

## WebServer 方法

管理 webserver 方法； 处理 client 请求方法； 响应 client 请求方法；

### WebServer 管理方法

**ESP8266WebServer() —— 创建 web server**

/\*\*

* 创建 webserver
* @param addr IPAddress (IP 地址)
* @param port int （端口号，默认是 80）

\*/

ESP8266WebServer(IPAddress addr, int port = 80);

/\*\*

* 创建 webserver（使用默认的 IP 地址）
* @param port int （端口号，默认是 80）

\*/

ESP8266WebServer(int port = 80);

### begin() —— 启动 web server

/\*\*

\* 启动 webserver

\*/

void begin();

/\*\*

* 启动 webserver
* @param port uint16\_t 端口号

\*/

void begin(uint16\_t port);

注意点：尽量在配置好各个请求处理之后再调用 begin 方法；

### close() —— 关闭 webserver

/\*\*

\* 关闭 webserver，关闭 TCP 连接

\*/

void close();

### stop() —— 关闭 webserver

/\*\*

\* 关闭 webserver

\* 底层就是调用 close();

\*/

void stop();

WebServer 处理 Http 请求的逻辑：

首先，获取有效的 Http 请求：\_currentClient.available()； 然后，开始解析 Http 请求：\_parseRequest(\_currentClient)； 解析 HTTP requestUri、Http requestMethod、HttpVersion； 寻找可以处理该请求的 requestHandler；

对于 GET 请求，解析请求头、请求参数（requestArguments）、请求主机名；

对于 POST、PUT 等非 GET 请求，也会解析请求头、请求参数、请求主机名；，然后根据 Content\_Type 的类型 去 匹配不同 的读取 数据 方法。如 果 Content\_Type 是 multipart/form-data,那么会处理表单数据，需用到 boundaryStr（特别地，如果涉及 到文件上传功能，这会在文件上传处理过程中回调注册的文件上传处理回调函数，请读 者自行往上翻阅）；如果 Content\_Type 属于其他的，则直接读取处理； 最后，匹配可以处理该请求的方法：\_handleRequest()；在该方法中会回调在第 2 步找

到的 requestHandler，requestHandler 会回调注册进去的对应请求的回调函数； 至此，整体的 Http 请求解析完成；

### 响应 client 请求方法

经过 handleClient()解析完 http 请求之后，就可以在 requestHandler 设置的请求处 理回调函数里面获得 http 请求的具体信息，然后根据具体信息给到对应的响应信息。

### WebServer 示例

3.3.1 WebServer

/\*\*

* Demo：
* 演示 web Server 功能
* 打开 PC 浏览器 输入 IP 地址。请求 web server

\*/

#include <ESP8266WiFi.h>

const char\* ssid = "GreenIoT";//wifi 账号 这里需要修改

const char\* password = "\*\*\*\*\*\*\*";//wifi 密码 这里需要修改

//创建 tcp server 端口号是 80 WiFiServer server(80);

void setup(){ Serial.begin(115200); Serial.println();

Serial.printf("Connecting to %s ", ssid); WiFi.mode(WIFI\_STA); WiFi.begin(ssid, password);

while (WiFi.status() != WL\_CONNECTED){ delay(500);

Serial.print(".");

}

Serial.println(" connected");

//启动 TCP 连接

server.begin();

//打印 TCP server IP 地址

Serial.printf("Web server started, open %s in a web browser\n", WiFi.localIP().toString().c\_str());

}

/\*\*

\* 模拟 web server 返回 http web 响应内容

\* 此处手动拼接 HTTP 响应内容

\*/

String prepareHtmlPage(){ String htmlPage =

String("HTTP/1.1 200 OK\r\n") + "Content-Type: text/html\r\n" +

"Connection: close\r\n" + // the connection will be closed after completion of the response

"Refresh: 5\r\n" + // refresh the page automatically every 5 sec "\r\n" +

"<!DOCTYPE HTML>" +

"<html>" +

"Analog input: " + String(analogRead(A0)) + "</html>" +

"\r\n"; return htmlPage;

}

void loop(){

WiFiClient client = server.available();

// wait for a client (web browser) to connect if (client){

Serial.println("\n[Client connected]"); while (client.connected()){

// 不断读取请求内容

if (client.available()){

String line = client.readStringUntil('\r'); Serial.print(line);

// wait for end of client's request, that is marked with an empty line if (line.length() == 1 && line[0] == '\n'){

//返回响应内容 client.println(prepareHtmlPage()); break;

}

}

//由于设置了 Connection: close 当响应数据之后就会自动断开连接

}

delay(100); // give the web browser time to receive the data

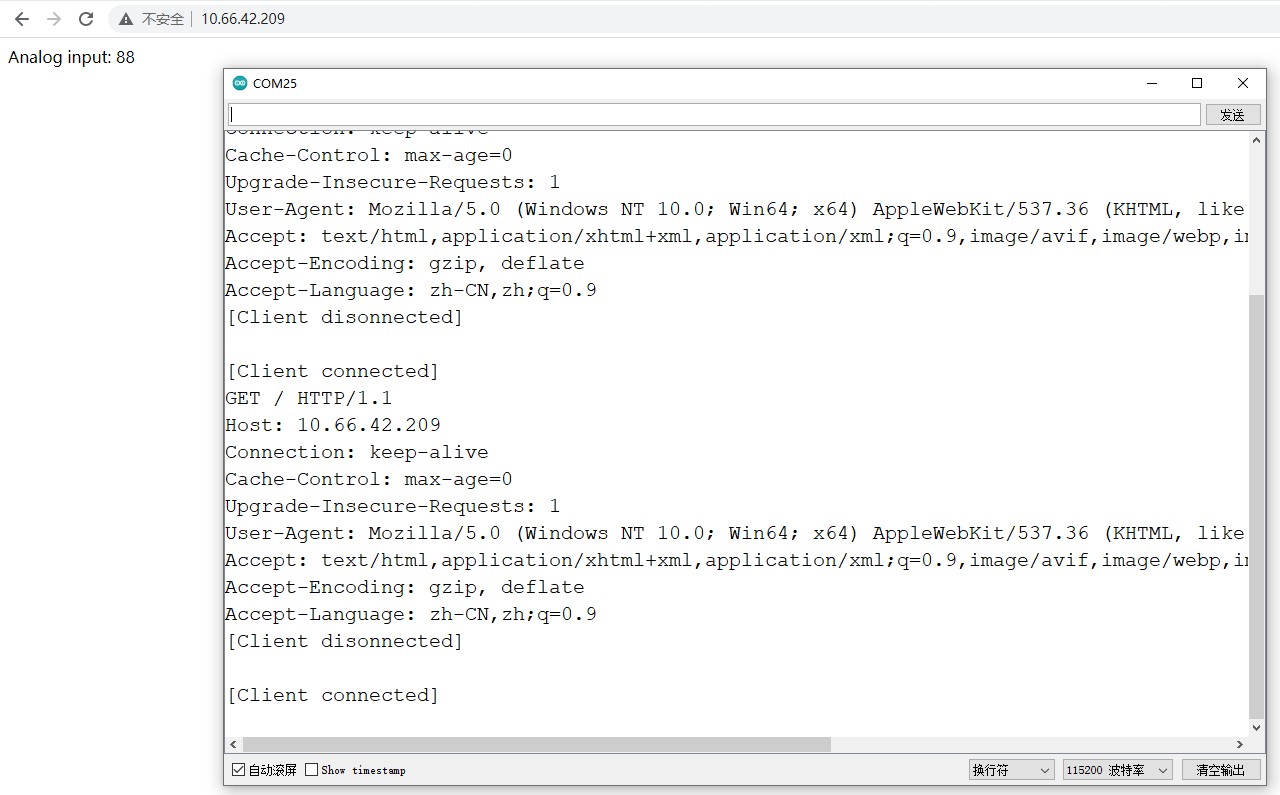
// close the connection: client.stop();

Serial.println("[Client disonnected]");

}

}

在浏览器地址输入 ESP8266 的 IP 地址：



### 3.3.2 响应浏览器不同请求的 WebServer

8266 作为 Web Server，可以响应浏览器的不同请求。

/\*

* 演示简单 web Server 功能

\* web server 会根据请求做不同的操作

* http://server\_ip/gpio/0 打印 /gpio0
* http://server\_ip/gpio/1 打印 /gpio1
* Server\_IP 就是 ESP8266 的 IP 地址

\*/

#include <ESP8266WiFi.h>

//以下三个定义为调试定义

#define DebugBegin(baud\_rate) Serial.begin(baud\_rate)

#define DebugPrintln(message) Serial.println(message)

#define DebugPrint(message) Serial.print(message)

const char\* ssid = "TP-LINK\_5344";//wifi 账号 这里需要修改

const char\* password = "xxxx";//wifi 密码 这里需要修改

// 创建 tcp server WiFiServer server(80);

void setup() {

DebugBegin(115200); delay(10);

// Connect to WiFi network DebugPrintln("");

DebugPrintln(String("Connecting to ") + ssid);

//STA WiFi.mode(WIFI\_STA);

//连接路由 wifi WiFi.begin(ssid, password);

while (WiFi.status() != WL\_CONNECTED) { delay(500);

DebugPrint(".");

}

DebugPrintln(""); DebugPrintln("WiFi connected");

// 启动 server server.begin();

DebugPrintln("Server started");

// 打印 IP 地址

DebugPrintln(WiFi.localIP().toString());

}

void loop() {

// 等待有效的 tcp 连接

WiFiClient client = server.available(); if (!client) {

return;

}

DebugPrintln("new client");

//等待 client 数据过来

while (!client.available()) { delay(1);

}

// 读取请求的第一行 会包括一个 url，这里只处理 url String req = client.readStringUntil('\r'); DebugPrintln(req);

//清掉缓冲区数据 据说这个方法没什么用 可以换种实现方式

client.flush();

// 开始匹配

int val;

if (req.indexOf("/gpio/0") != -1) { DebugPrintln("/gpio0");

val = 0;

} else if (req.indexOf("/gpio/1") != -1) { DebugPrintln("/gpio1");

val = 1;

} else {

DebugPrintln("invalid request");

//关闭这个 client 请求

client.stop(); return;

}

//清理缓冲区数据 client.flush();

// 准备响应数据

String s = "HTTP/1.1 200 OK\r\nContent-Type: text/html\r\n\r\n<!DOCTYPE HTML>\r\n<html>\r\nGPIO is now ";

s += (val) ? "high" : "low"; s += "</html>\n";

// 发送响应数据给 client client.print(s);

delay(1);

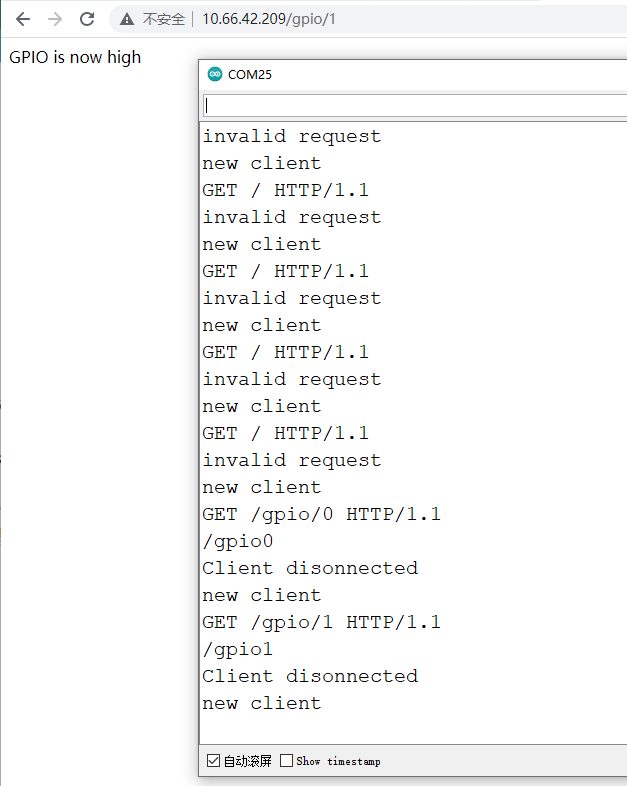
DebugPrintln("Client disonnected");

// The client will actually be disconnected

// when the function returns and 'client' object is detroyed

}

测试结果：



### 带登录基础认证的 WebServer

#include <ESP8266WiFi.h>

#include <ESP8266mDNS.h>

#include <ArduinoOTA.h>

#include <ESP8266WebServer.h>

#ifndef STASSID

#define STASSID "GreenIoT"

#define STAPSK "\*\*\*\*\*\*\*"

#endif

const char\* ssid = STASSID; const char\* password = STAPSK;

ESP8266WebServer server(80);

const char\* www\_username = "admin"; const char\* www\_password = "esp8266";

void setup() {

Serial.begin(115200);

WiFi.mode(WIFI\_STA); WiFi.begin(ssid, password);

if (WiFi.waitForConnectResult() != WL\_CONNECTED) { Serial.println("WiFi Connect Failed! Rebooting..."); delay(1000);

ESP.restart();

}

ArduinoOTA.begin();

server.on("/", []() {

if (!server.authenticate(www\_username, www\_password)) { return server.requestAuthentication();

}

server.send(200, "text/plain", "Login OK");

});

server.begin();

Serial.print("Open http://"); Serial.print(WiFi.localIP());

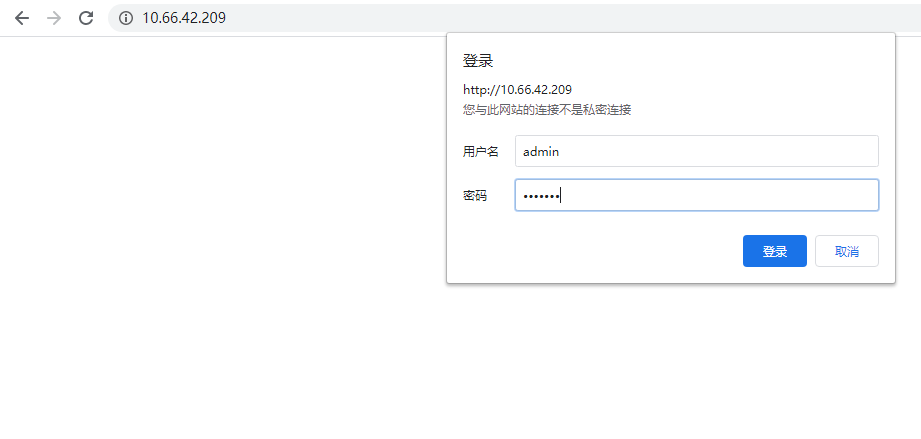
Serial.println("/ in your browser to see it working");

}

void loop() { ArduinoOTA.handle(); server.handleClient();

}

演示效果



### 带高级认证的 WebServer

#include <ESP8266WiFi.h>

#include <WiFiClient.h>

#include <ESP8266WebServer.h>

#ifndef STASSID

#define STASSID "GreenIoT"

#define STAPSK "\*\*\*\*\*\*"

#endif

const char\* ssid = STASSID; const char\* password = STAPSK; ESP8266WebServer server(80);

//Check if header is present and correct bool is\_authenticated() {

Serial.println("Enter is\_authenticated"); if (server.hasHeader("Cookie")) {

Serial.print("Found cookie: ");

String cookie = server.header("Cookie"); Serial.println(cookie);

if (cookie.indexOf("ESPSESSIONID=1") != -1) { Serial.println("Authentication Successful"); return true;

}

}

Serial.println("Authentication Failed"); return false;

}

//login page, also called for disconnect void handleLogin() {

String msg;

if (server.hasHeader("Cookie")) { Serial.print("Found cookie: ");

String cookie = server.header("Cookie"); Serial.println(cookie);

}

if (server.hasArg("DISCONNECT")) { Serial.println("Disconnection"); server.sendHeader("Location", "/login"); server.sendHeader("Cache-Control", "no-cache"); server.sendHeader("Set-Cookie", "ESPSESSIONID=0"); server.send(301);

return;

}

if (server.hasArg("USERNAME") && server.hasArg("PASSWORD")) {

if (server.arg("USERNAME") == "admin" && server.arg("PASSWORD") == "admin") { server.sendHeader("Location", "/");

server.sendHeader("Cache-Control", "no-cache"); server.sendHeader("Set-Cookie", "ESPSESSIONID=1"); server.send(301);

Serial.println("Log in Successful"); return;

}

msg = "Wrong username/password! try again."; Serial.println("Log in Failed");

}

String content = "<html><body><form action='/login' method='POST'>To log in, please use : admin/admin<br>";

content += "User:<input type='text' name='USERNAME' placeholder='user name'><br>";

content += "Password:<input type='password' name='PASSWORD' placeholder='password'><br>";

content += "<input type='submit' name='SUBMIT' value='Submit'></form>" + msg + "<br>";

content += "You also can go <a href='/inline'>here</a></body></html>"; server.send(200, "text/html", content);

}

//root page can be accessed only if authentication is ok void handleRoot() {

Serial.println("Enter handleRoot"); String header;

if (!is\_authenticated()) { server.sendHeader("Location", "/login"); server.sendHeader("Cache-Control", "no-cache"); server.send(301);

return;

}

String content = "<html><body><H2>hello, you successfully connected to esp8266!</H2><br>";

if (server.hasHeader("User-Agent")) {

content += "the user agent used is : " + server.header("User-Agent") + "<br><br>";

}

content += "You can access this page until you <a href=\"/login?DISCONNECT=YES\">disconnect</a></body></html>";

server.send(200, "text/html", content);

}

//no need authentication

void handleNotFound() {

String message = "File Not Found\n\n"; message += "URI: ";

message += server.uri(); message += "\nMethod: ";

message += (server.method() == HTTP\_GET) ? "GET" : "POST"; message += "\nArguments: ";

message += server.args(); message += "\n";

for (uint8\_t i = 0; i < server.args(); i++) {

message += " " + server.argName(i) + ": " + server.arg(i) + "\n";

}

server.send(404, "text/plain", message);

}

void setup(void) { Serial.begin(115200); WiFi.mode(WIFI\_STA); WiFi.begin(ssid, password); Serial.println("");

// Wait for connection

while (WiFi.status() != WL\_CONNECTED) { delay(500);

Serial.print(".");

}

Serial.println(""); Serial.print("Connected to "); Serial.println(ssid); Serial.print("IP address: "); Serial.println(WiFi.localIP());

server.on("/", handleRoot); server.on("/login", handleLogin); server.on("/inline", []() {

server.send(200, "text/plain", "this works without need of authentication");

});

server.onNotFound(handleNotFound);

//here the list of headers to be recorded

const char \* headerkeys[] = {"User-Agent", "Cookie"} ; size\_t headerkeyssize = sizeof(headerkeys) / sizeof(char\*);

//ask server to track these headers

server.collectHeaders(headerkeys, headerkeyssize); server.begin();

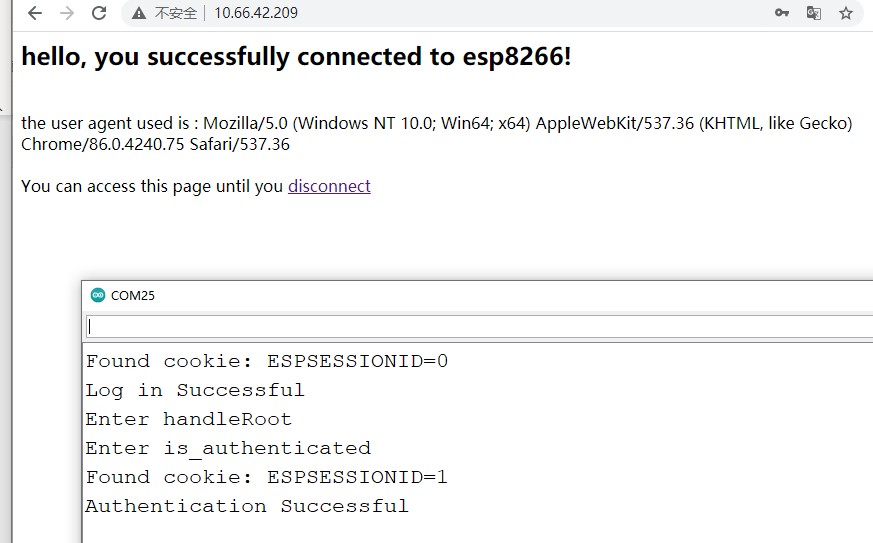
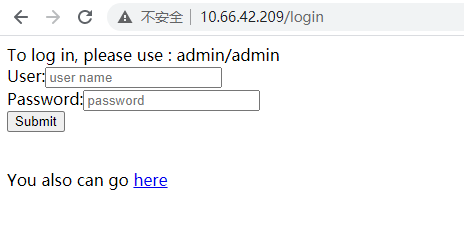
Serial.println("HTTP server started");

}

void loop(void) { server.handleClient();

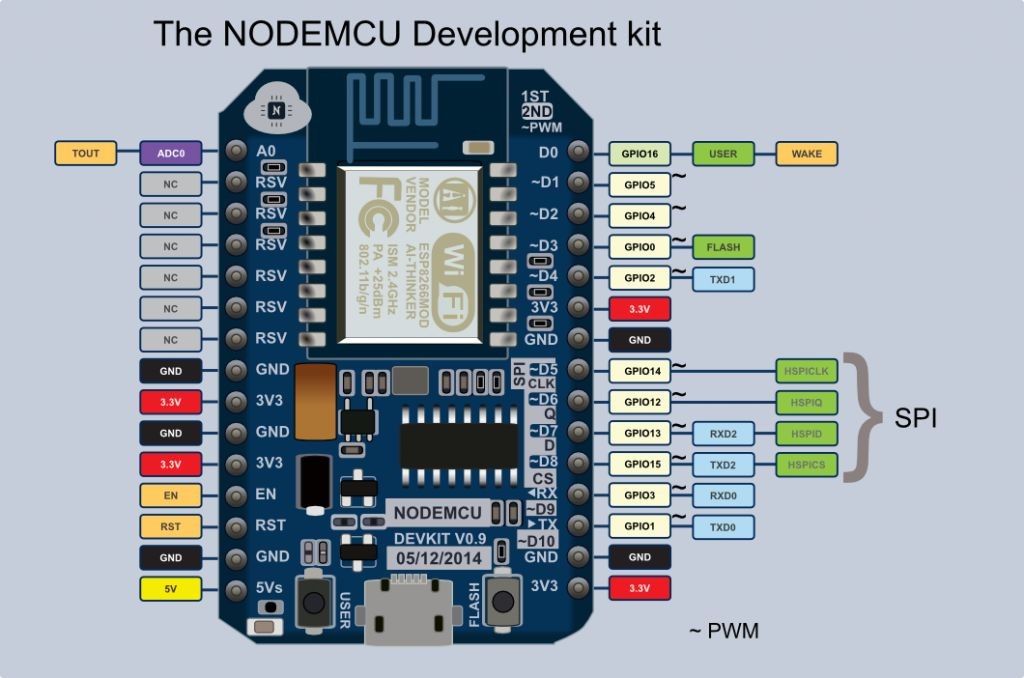
}

页面效果



### 

**附录**



### HTTP 教程

HTTP 协议（HyperText Transfer Protocol，超文本传输协议）是因特网上应用最为广泛 的一种网络传输协议，所有的 WWW 文件都必须遵守这个标准。

HTTP 是一个基于 TCP/IP 通信协议来传递数据（HTML 文件, 图片文件, 查询结果等）。

<https://www.runoob.com/http/http-tutorial.html>

<https://blog.csdn.net/aboutmn/article/details/88074507>