5 应用层协议 HTTP

HTTP 协议

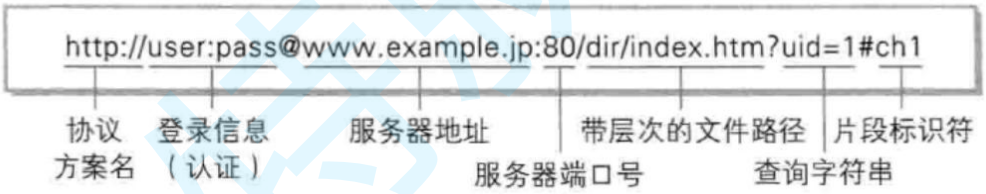
虽然我们说, 应用层协议是我们程序猿自己定的. 但实际上, 已经有大佬们定义了一些现成的, 又非常好用的应用层协议, 供我们直接参考使用. HTTP(超文本传输协议)就是其中之一。

在互联网世界中，HTTP（**H**yper**T**ext **T**ransfer **P**rotocol，超文本传输协议）是一个至关重要的协议。它定义了客户端（如浏览器）与服务器之间如何通信，以交换或传输超文本（如 HTML 文档）。

HTTP 协议是客户端与服务器之间通信的基础。客户端通过 HTTP 协议向服务器发送请求，服务器收到请求后处理并返回响应。HTTP 协议是一个**无连接、无状态**的协议，即每次请求都需要建立新的连接，且服务器不会保存客户端的状态信息。

认识 URL

平时我们俗称的 "网址" 其实就是说的 URL



urlencode 和 urldecode

像 / ? : 等这样的字符, 已经被 url 当做特殊意义理解了. 因此这些字符不能随意出现.

比如, 某个参数中需要带有这些特殊字符, 就必须先对特殊字符进行转义.

转义的规则如下:

将需要转码的字符转为 16 进制，然后从右到左，取 4 位(不足 4 位直接处理)，每 2 位做一位，前面加上%，编码成%XY 格式

例如:

"+" 被转义成了 "%2B"

urldecode 就是 urlencode 的逆过程;

urlencode 工具

HTTP 协议请求与响应格式

HTTP 请求

```
POST http://job.xjtu.edu.cn/companyLogin.do HTTP/1.1
Host: job.xjtu.edu.cn
Connection: keep-alive
Content-Length: 36
Cache-Control: max-age=0
Origin: http://job.xjtu.edu.cn
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/61.0.3163.100 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/
apng,*/*;q=0.8
Referer: http://job.xjtu.edu.cn/companyLogin.do
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.8
Cookie: JSESSIONID=D628A75845A74D29D991DB47A461E4FC;
Hm_lvt_783e83ce0ee350e23a9d389df580f658=1504963710,1506661798;
Hm_lpvt_783e83ce0ee350e23a9d389df580f658=1506661802

username=hgtz2222&password=222222222
```

- 首行: [方法] + [url] + [版本]
- Header: 请求的属性, 冒号分割的键值对;每组属性之间使用\r\n 分隔;遇到空行表示 Header 部分结束
- Body: 空行后面的内容都是 Body. Body 允许为空字符串. 如果 Body 存在, 则在

> Header 中会有一个 Content-Length 属性来标识 Body 的长度;

编写 HTTP 请求的代码 - 验证 http 请求

C++
需要现场基于历史代码，先架构处一个基本的 HTTP 服务器，然后用浏览器进行验证

HTTP 响应

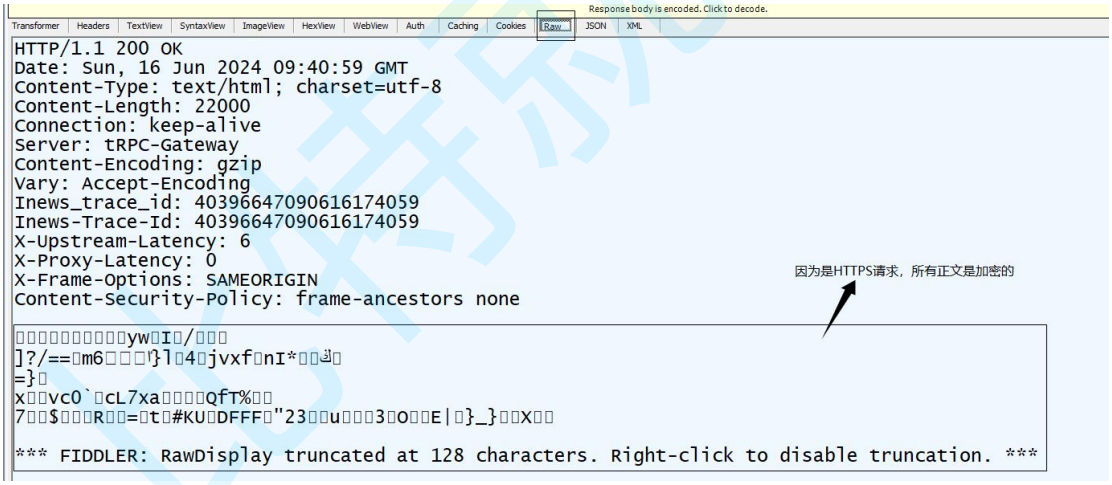
```
HTTP/1.1 200 OK
Server: YxlinkWAF
Content-Type: text/html;charset=UTF-8
Content-Language: zh-CN
Transfer-Encoding: chunked
Date: Fri, 29 Sep 2017 05:10:13 GMT

<!DOCTYPE html>
<html>
<head>
<title>西安交通大学就业网</title>
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<meta http-equiv="X-UA-Compatible" content="IE=Edge">
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<link rel="shortcut icon" href="/renovation/images/icon.ico">
<link href="/renovation/css/main.css" rel="stylesheet" media="screen" />
<link href="/renovation/css/art_default.css" rel="stylesheet" media="screen" />
<link href="/renovation/css/font-awesome.css" rel="stylesheet" media="screen" />
<script type="text/javascript" src="/renovation/js/jquery1.7.1.min.js"></script>
<script type="text/javascript" src="/renovation/js/main.js"></script><!--main-->
<link href="/style/warmTipsstyle.css" rel="stylesheet" type="text/css">
</head>
```

- 首行: [版本号] + [状态码] + [状态码解释]
- Header: 请求的属性, 冒号分割的键值对;每组属性之间使用\r\n 分隔;遇到空行

表示 Header 部分结束

- Body: 空行后面的内容都是 Body. Body 允许为空字符串. 如果 Body 存在, 则在 Header 中会有一个 Content-Length 属性来标识 Body 的长度; 如果服务器返回了一个 html 页面, 那么 html 页面内容就是在 body 中.

基本的应答格式



HTTP 的方法

方法	说明	支持的HTTP协议版本
GET	获取资源	1.0、1.1
POST	传输实体主体	1.0、1.1
PUT	传输文件	1.0、1.1
HEAD	获得报文首部	1.0、1.1
DELETE	删除文件	1.0、1.1
OPTIONS	询问支持的方法	1.1
TRACE	追踪路径	1.1
CONNECT	要求用隧道协议连接代理	1.1
LINK	建立和资源之间的联系	1.0
UNLINE	断开连接关系	1.0

其中最常用的就是 GET 方法和 POST 方法.

HTTP 常见方法

1. GET 方法（重点）

用途：用于请求 URL 指定的资源。

示例：`GET /index.html HTTP/1.1`

特性：指定资源经服务器端解析后返回响应内容。

form 表单：https://www.runoob.com/html/html-forms.html

C++

要通过历史写的 http 服务器，验证 GET 方法,这里需要了解一下 FORM 表单的问题

这里就要引入 web 根目录，文件读取的基本操作了

```
std::string GetFileContentHelper(const std::string &path)
{
    // 一份简单的读取二进制文件的代码
    std::ifstream in(path, std::ios::binary);
    if (!in.is_open())
        return "";
    in.seekg(0, in.end);
    int filesize = in.tellg();
    in.seekg(0, in.beg);

    std::string content;
    content.resize(filesize);
    in.read((char *)content.c_str(), filesize);
```

```
        // std::vector<char> content(filesize);
        // in.read(content.data(), filesize);

        in.close();

        return content;
    }
```

2. POST 方法（重点）

用途：用于传输实体的主体，通常用于提交表单数据。

示例：`POST /submit.cgi HTTP/1.1`

特性：可以发送大量的数据给服务器，并且数据包含在请求体中。

form 表单：https://www.runoob.com/html/html-forms.html

```
C++
要通过历史写的 http 服务器，验证 POST 方法,这里需要了解一下 FORM 表单的问题
```

3. PUT 方法（不常用）

用途：用于传输文件，将请求报文主体中的文件保存到请求 URL 指定的位置。

示例：`PUT /example.html HTTP/1.1`

特性：不太常用，但在某些情况下，如 RESTful API 中，用于更新资源。

4. HEAD 方法

用途：与 GET 方法类似，但不返回报文主体部分，仅返回响应头。

示例：`HEAD /index.html HTTP/1.1`

特性：用于确认 URL 的有效性及资源更新的日期时间等。

```
C++
// curl -i 显示
$ curl -i www.baidu.com
HTTP/1.1 200 OK
Accept-Ranges: bytes
Cache-Control: private, no-cache, no-store, proxy-revalidate, no-transform
Connection: keep-alive
```

```
Content-Length: 2381
Content-Type: text/html
Date: Sun, 16 Jun 2024 08:38:04 GMT
Etag: "588604dc-94d"
Last-Modified: Mon, 23 Jan 2017 13:27:56 GMT
Pragma: no-cache
Server: bfe/1.0.8.18
Set-Cookie: BDORZ=27315; max-age=86400; domain=.baidu.com; path=/

<!DOCTYPE html>
...

// 使用 head 方法，只会返回响应头
$ curl --head www.baidu.com
HTTP/1.1 200 OK
Accept-Ranges: bytes
Cache-Control: private, no-cache, no-store, proxy-revalidate, no-transform
Connection: keep-alive
Content-Length: 277
Content-Type: text/html
Date: Sun, 16 Jun 2024 08:43:38 GMT
Etag: "575e1f71-115"
Last-Modified: Mon, 13 Jun 2016 02:50:25 GMT
Pragma: no-cache
Server: bfe/1.0.8.18
```

5. DELETE 方法（不常用）

用途：用于删除文件，是 PUT 的相反方法。

示例：`DELETE /example.html HTTP/1.1`

特性：按请求 URL 删除指定的资源。

6. OPTIONS 方法

用途：用于查询针对请求 URL 指定的资源支持的方法。

示例：`OPTIONS * HTTP/1.1`

特性：返回允许的方法，如 GET、POST 等。

不支持的效果

```
C++
```

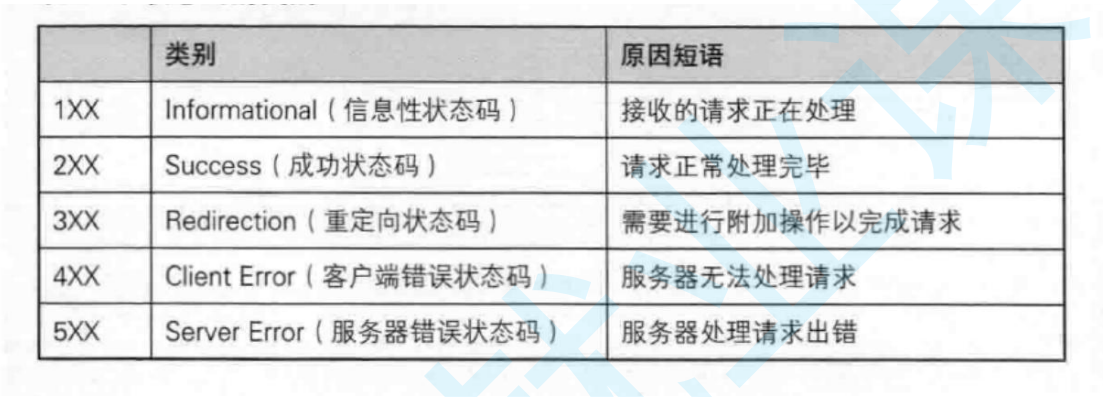
```
// 搭建一个 nginx 用来测试
// sudo apt install nginx
// sudo nginx -- 开启
// ps ajx | grep nginx -- 查看
// sudo nginx -s stop -- 停止服务

$ sudo nginx -s stop
$ ps ajx | grep nginx
2944845 2945390 2945389 2944845 pts/1    2945389 S+    1002   0:00
grep --color=auto nginx
$ sudo nginx
$ ps axj | grep nginx
      1 2945393 2945393 2945393 ?             -1 Ss       0   0:00
nginx: master process nginx
2945393 2945394 2945393 2945393 ?             -1 S       33   0:00
nginx: worker process
2945393 2945395 2945393 2945393 ?             -1 S       33   0:00
nginx: worker process
2944845 2945397 2945396 2944845 pts/1    2945396 S+    1002   0:00
grep --color=auto nginx

// -X(大 x) 指明方法
$ curl -X OPTIONS -i http://127.0.0.1/
HTTP/1.1 405 Not Allowed
Server: nginx/1.18.0 (Ubuntu)
Date: Sun, 16 Jun 2024 08:48:22 GMT
Content-Type: text/html
Content-Length: 166
Connection: keep-alive

<html>
<head><title>405 Not Allowed</title></head>
<body>
<center><h1>405 Not Allowed</h1></center>
<hr><center>nginx/1.18.0 (Ubuntu)</center>
</body>
</html>
```

支持的效果

```
C++
HTTP/1.1 200 OK
Allow: GET, HEAD, POST, OPTIONS
```

```
Content-Type: text/plain
Content-Length: 0
Server: nginx/1.18.0 (Ubuntu)
Date: Sun, 16 Jun 2024 09:04:44 GMT
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET, POST, OPTIONS
Access-Control-Allow-Headers: Content-Type, Authorization

// 注意：这里没有响应体，因为 Content-Length 为 0
```

HTTP 的状态码



	类别	原因短语
1XX	Informational（信息性状态码）	接收的请求正在处理
2XX	Success（成功状态码）	请求正常处理完毕
3XX	Redirection（重定向状态码）	需要进行附加操作以完成请求
4XX	Client Error（客户端错误状态码）	服务器无法处理请求
5XX	Server Error（服务器错误状态码）	服务器处理请求出错

最常见的状态码, 比如 200(OK), 404(Not Found), 403(Forbidden), 302(Redirect, 重定向), 504(Bad Gateway)

状态码	含义	应用样例
100	Continue	上传大文件时，服务器告诉客户端可以继续上传
200	OK	访问网站首页，服务器返回网页内容
201	Created	发布新文章，服务器返回文章创建成功的信息
204	No Content	删除文章后，服务器返回“无内容”表示操作成功
301	Moved Permanently	网站换域名后，自动跳转到新域名；搜索引擎更新网站链接时使用

302	Found 或 See Other	用户登录成功后，重定向到用户首页
304	Not Modified	浏览器缓存机制，对未修改的资源返回 304 状态码
400	Bad Request	填写表单时，格式不正确导致提交失败
401	Unauthorized	访问需要登录的页面时，未登录或认证失败
403	Forbidden	尝试访问你没有权限查看的页面
404	Not Found	访问不存在的网页链接
500	Internal Server Error	服务器崩溃或数据库错误导致页面无法加载
502	Bad Gateway	使用代理服务器时，代理服务器无法从上游服务器获取有效响应
503	Service Unavailable	服务器维护或过载，暂时无法处理请求

好的，以下是仅包含重定向相关状态码的表格：

状态码	含义	是否为临时重定向	应用样例
301	Moved Permanently	否（永久重定向）	网站换域名后，自动跳转到新域名；搜索引擎更新网站链接时使用
302	Found 或 See Other	是（临时重定向）	用户登录成功后，重定向到用户首页
307	Temporary Redirect	是（临时重定向）	临时重定向资源到新的位置（较少使用）

308	Permanent Redirect	否（永久重定向）	永久重定向资源到新的位置（较少使用）

关于重定向的验证，以 301 为代表

HTTP 状态码 301（永久重定向）和 302（临时重定向）都依赖 Location 选项。以下是关于两者依赖 Location 选项的详细说明：

HTTP 状态码 301（永久重定向）：

- 当服务器返回 HTTP 301 状态码时，表示请求的资源已经被永久移动到新的位置。
- 在这种情况下，服务器会在响应中添加一个 Location 头部，用于指定资源的新位置。这个 Location 头部包含了新的 URL 地址，浏览器会自动重定向到该地址。
- 例如，在 HTTP 响应中，可能会看到类似于以下的头部信息：

```
C++
HTTP/1.1 301 Moved Permanently\r\n
Location: https://www.new-url.com\r\n
```

HTTP 状态码 302（临时重定向）：

- 当服务器返回 HTTP 302 状态码时，表示请求的资源临时被移动到新的位置。
- 同样地，服务器也会在响应中添加一个 Location 头部来指定资源的新位置。浏览器会暂时使用新的 URL 进行后续的请求，但不会缓存这个重定向。
- 例如，在 HTTP 响应中，可能会看到类似于以下的头部信息：

```
C++
HTTP/1.1 302 Found\r\n
Location: https://www.new-url.com\r\n
```

总结：无论是 HTTP 301 还是 HTTP 302 重定向，都需要依赖 Location 选项来指定资源的新位置。这个 Location 选项是一个标准的 HTTP 响应头部，用于告诉浏览器应该将请求重定向到哪个新的 URL 地址。

HTTP 常见 Header

> - Content-Type: 数据类型(text/html 等)
> - Content-Length: Body 的长度

> - Host: 客户端告知服务器, 所请求的资源是在哪个主机的哪个端口上;
> - User-Agent: 声明用户的操作系统和浏览器版本信息;
> - referer: 当前页面是从哪个页面跳转过来的;
> - Location: 搭配 3xx 状态码使用, 告诉客户端接下来要去哪里访问;
> - Cookie: 用于在客户端存储少量信息. 通常用于实现会话(session)的功能;

关于 connection 报头

HTTP 中的 `Connection` 字段是 HTTP 报文头的一部分，它主要用于控制和管理客户端与服务器之间的连接状态

核心作用

- **管理持久连接**：`Connection` 字段还用于管理持久连接（也称为长连接）。持久连接允许客户端和服务器在请求/响应完成后不立即关闭 TCP 连接，以便在同一个连接上发送多个请求和接收多个响应。

持久连接（长连接）

- **HTTP/1.1**：在 HTTP/1.1 协议中，默认使用持久连接。当客户端和服务器都不明确指定关闭连接时，连接将保持打开状态，以便后续的请求和响应可以复用同一个连接。
- **HTTP/1.0**：在 HTTP/1.0 协议中，默认连接是非持久的。如果希望在 HTTP/1.0 上实现持久连接，需要在请求头中显式设置 `Connection: keep-alive`。

语法格式

- `Connection: keep-alive`：表示希望保持连接以复用 TCP 连接。
- `Connection: close`：表示请求/响应完成后，应该关闭 TCP 连接。

User-Agent 里的历史故事

下面附上一张关于 HTTP 常见 header 的表格

字段名	含义	样例
Accept	客户端可接受的响应内容类型	`Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8`
Accept-Encoding	客户端支持的数据压缩格式	`Accept-Encoding: gzip, deflate, br`

Accept-Language	客户端可接受的语言类型	`Accept-Language: zh-CN,zh;q=0.9,en;q=0.8`
Host	请求的主机名和端口号	`Host: www.example.com:8080`
User-Agent	客户端的软件环境信息	`User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36`
Cookie	客户端发送给服务器的 HTTP cookie 信息	`Cookie: session_id=abcdefg12345; user_id=123`
Referer	请求的来源 URL	`Referer: http://www.example.com/previous_page.html`
Content-Type	实体主体的媒体类型	`Content-Type: application/x-www-form-urlencoded` (对于表单提交) 或 `Content-Type: application/json` (对于 JSON 数据)
Content-Length	实体主体的字节大小	`Content-Length: 150`
Authorization	认证信息，如用户名和密码	`Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==` (Base64 编码后的用户名:密码)
Cache-Control	缓存控制指令	请求时：`Cache-Control: no-cache` 或 `Cache-Control: max-age=3600`；响应时：`Cache-Control: public, max-age=3600`
Connection	请求完后是关闭还是保持连接	`Connection: keep-alive` 或 `Connection: close`

Date	请求或响应的日期和时间	`Date: Wed, 21 Oct 2023 07:28:00 GMT`
Location	重定向的目标 URL（与 3xx 状态码配合使用）	`Location: http://www.example.com/new_location.html` (与 302 状态码配合使用)
Server	服务器类型	`Server: Apache/2.4.41 (Unix)`
Last-Modified	资源的最后修改时间	`Last-Modified: Wed, 21 Oct 2023 07:20:00 GMT`
ETag	资源的唯一标识符，用于缓存	`ETag: "3f80f-1b6-5f4e2512a4100"`
Expires	响应过期的日期和时间	`Expires: Wed, 21 Oct 2023 08:28:00 GMT`

最简单的 HTTP 服务器

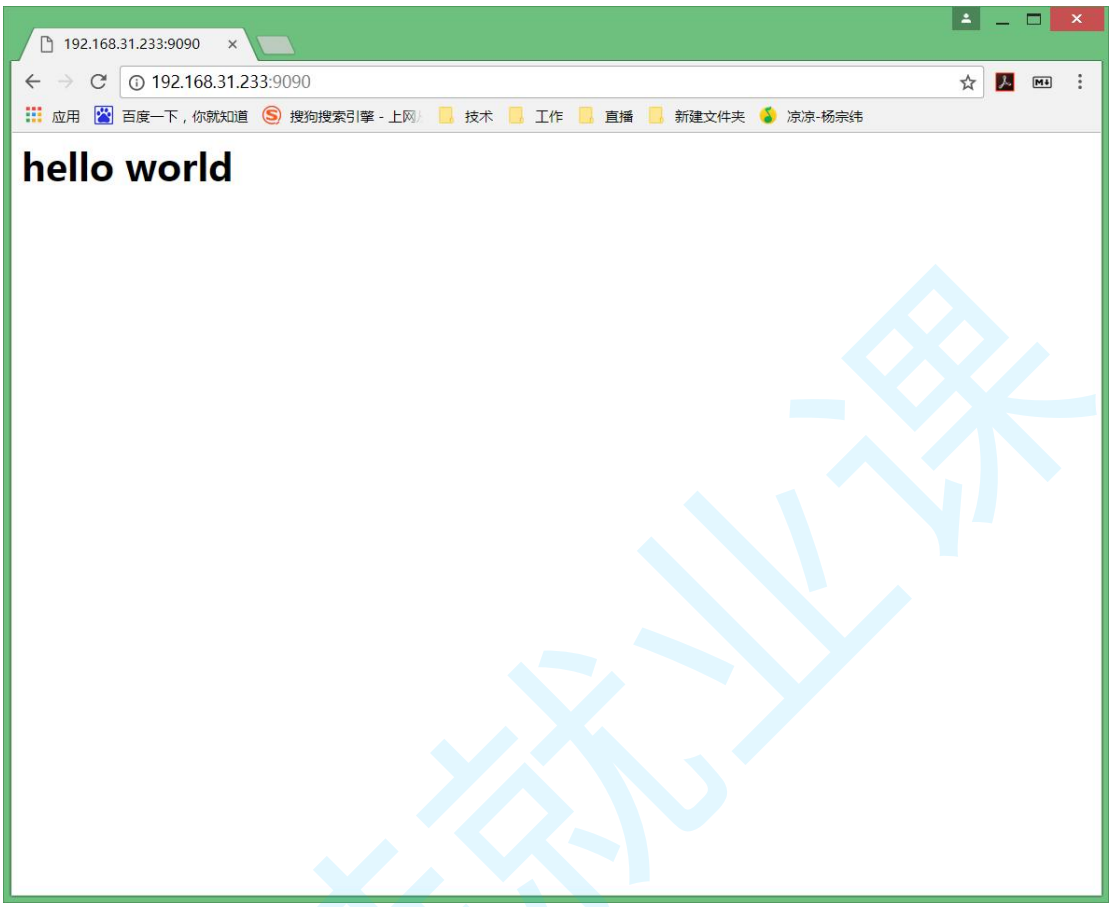
实现一个最简单的 HTTP 服务器, 只在网页上输出 "hello world"; 只要我们按照 HTTP 协议的要求构造数据, 就很容易能做到;

```c
C
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void Usage() {
  printf("usage: ./server [ip] [port]\n");
}

int main(int argc, char* argv[]) {
  if (argc != 3) {
    Usage();
    return 1;
  }
```

```
  int fd = socket(AF_INET, SOCK_STREAM, 0);
  if (fd < 0) {
    perror("socket");
    return 1;
  }
  struct sockaddr_in addr;
  addr.sin_family = AF_INET;
  addr.sin_addr.s_addr = inet_addr(argv[1]);
  addr.sin_port = htons(atoi(argv[2]));

  int ret = bind(fd, (struct sockaddr*)&addr, sizeof(addr));
  if (ret < 0) {
    perror("bind");
    return 1;
  }
  ret = listen(fd, 10);
  if (ret < 0) {
    perror("listen");
    return 1;
  }
  for (;;) {
    struct sockaddr_in client_addr;
    socklen_t len;
    int client_fd = accept(fd, (struct sockaddr*)&client_addr,
&len);
    if (client_fd < 0) {
      perror("accept");
      continue;
    }
    char input_buf[1024 * 10] = {0};  // 用一个足够大的缓冲区直接把数
据读完.
    ssize_t read_size = read(client_fd, input_buf,
sizeof(input_buf) - 1);
    if (read_size < 0) {
      return 1;
    }
    printf("[Request] %s", input_buf);
    char buf[1024] = {0};
    const char* hello = "<h1>hello world</h1>";
    sprintf(buf, "HTTP/1.0 200 OK\nContent-Length:%lu\n\n%s",
strlen(hello), hello);
    write(client_fd, buf, strlen(buf));
  }
  return 0;
```

```
}
```

编译, 启动服务. 在浏览器中输入 http://[ip]:[port], 就能看到显示的结果 "Hello World"




```
[tangzhong@tz http]$ ./server 0 9090
GET / HTTP/1.1
Host: 192.168.31.233:9090
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100
Safari/537.36
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.8
```

> **备注:**
>
> 此处我们使用 9090 端口号启动了 HTTP 服务器. 虽然 HTTP 服务器一般使用 80 端口,
>
> 但这只是一个通用的习惯. 并不是说 HTTP 服务器就不能使用其他的端口号.
>
> 使用 chrome 测试我们的服务器时, 可以看到服务器打出的请求中还有一个 `GET /favicon.ico HTTP/1.1` 这样的请求.
>
> 同学们自行查找资料, 去理解 favicon.ico 的作用.

> **实验**
>
> 把返回的状态码改成 404, 403, 504 等, 看浏览器上分别会出现什么样的效果.

附录：

HTTP 历史及版本核心技术与时代背景

HTTP（Hypertext Transfer Protocol，超文本传输协议）作为互联网中浏览器和服务器间通信的基石，经历了从简单到复杂、从单一到多样的发展过程。以下将按照时间顺序，介绍 HTTP 的主要版本、核心技术及其对应的时代背景。

HTTP/0.9

核心技术:

- 仅支持 GET 请求方法。
- 仅支持纯文本传输，主要是 HTML 格式。
- 无请求和响应头信息。

时代背景:

- 1991 年，HTTP/0.9 版本作为 HTTP 协议的最初版本，用于传输基本的超文本 HTML 内容。
- 当时的互联网还处于起步阶段，网页内容相对简单，主要以文本为主。

HTTP/1.0

核心技术:

- 引入 POST 和 HEAD 请求方法。
- 请求和响应头信息，支持多种数据格式（MIME）。
- 支持缓存（cache）。
- 状态码（status code）、多字符集支持等。

时代背景:

- 1996 年，随着互联网的快速发展，网页内容逐渐丰富，HTTP/1.0 版本应运而生。
- 为了满足日益增长的网络应用需求，HTTP/1.0 增加了更多的功能和灵活性。
- 然而，HTTP/1.0 的工作方式是每次 TCP 连接只能发送一个请求，性能上存在一

定局限。

HTTP/1.1

核心技术:

- 引入持久连接（persistent connection），支持管道化（pipelining）。
- 允许在单个 TCP 连接上进行多个请求和响应，提高了性能。
- 引入分块传输编码（chunked transfer encoding）。
- 支持 Host 头，允许在一个 IP 地址上部署多个 Web 站点。

时代背景:

- 1999 年，随着网页加载的外部资源越来越多，HTTP/1.0 的性能问题愈发突出。
- HTTP/1.1 通过引入持久连接和管道化等技术，有效提高了数据传输效率。
- 同时，互联网应用开始呈现出多元化、复杂化的趋势，HTTP/1.1 的出现满足了这些需求。

HTTP/2.0

核心技术:

- 多路复用（multiplexing），一个 TCP 连接允许多个 HTTP 请求。
- 二进制帧格式（binary framing），优化数据传输。
- 头部压缩（header compression），减少传输开销。
- 服务器推送（server push），提前发送资源到客户端。

时代背景:

- 2015 年，随着移动互联网的兴起和云计算技术的发展，网络应用对性能的要求越来越高。
- HTTP/2.0 通过多路复用、二进制帧格式等技术，显著提高了数据传输效率和网络性能。
- 同时，HTTP/2.0 还支持加密传输（HTTPS），提高了数据传输的安全性。

HTTP/3.0

核心技术:

- 使用 QUIC 协议替代 TCP 协议，基于 UDP 构建的多路复用传输协议。
- 减少了 TCP 三次握手及 TLS 握手时间，提高了连接建立速度。

- 解决了 TCP 中的线头阻塞问题，提高了数据传输效率。

时代背景：

- 2022 年，随着 5G、物联网等技术的快速发展，网络应用对实时性、可靠性的要求越来越高。
- HTTP/3.0 通过使用 QUIC 协议，提高了连接建立速度和数据传输效率，满足了这些需求。
- 同时，HTTP/3.0 还支持加密传输（HTTPS），保证了数据传输的安全性。