Go网络编程——http

目录

1. Http概述
2. Http协议客户端实现
3. Http协议服务端实现

一、Http概述

（一）概念

Hypertext Transfer Protocol，超文本传输协议。是互联网应用最广泛的网络协议，定义了客户端和服务端之间请求与响应的传输标准。

HTTP是一个基于TCP/IP通信协议来传递数据，服务器传输超文本到本地浏览器的传送协议。HTTP协议工作于客户端-服务端架构上。浏览器可作为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。它是一个无状态的请求/响应协议。

客户端请求消息和服务器响应消息都会包含请求头和请求体。HTTP请求头提供了关于请求或响应，发送实体的信息，如：Content-Type、Content-Length、Date等。当浏览器接收并显示网页前，此网页所在的服务器会返回一个包含HTTP状态码的信息头（server header）用以响应浏览器的请求。
HTTP状态码的英文为HTTP Status Code。

下面是常见的HTTP状态码：

- 200 - 请求成功
- 301 - 资源（网页等）被永久转移到其它URL
- 404 - 请求的资源（网页等）不存在
- 413 - 由于请求的实体过大，服务器无法处理，因此拒绝请求。为防止客户端的连续请求，服务器可能会关闭连接。
- 500 - 内部服务器错误

（二）、HTTP请求方法

常用方法: Get\Post\Head

Http定义了与服务器交互的不同方法，最基本的方法有4种，分别是**GET，POST，PUT，DELETE，**对应着对这个资源的查，改，增，删 4个操作。

另外还有个Head方法. 类似GET方法，只请求页面的首部，不响应页面Body部分，用于获取资源的基本信息，即检查链接的可访问性及资源是否修改。

GET和POST的区别

- GET在浏览器回退时是无害的，而POST会再次提交请求。
- GET产生的URL地址可以被Bookmark，而POST不可以。
- GET请求会被浏览器主动cache，而POST不会，除非手动设置。
- GET请求只能进行url编码，而POST支持多种编码方式。
- GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。
- GET请求在URL中传送的参数是有长度限制的，而POST没有。
- 对参数的数据类型，GET只接受ASCII字符，而POST没有限制。
- GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。
- GET参数通过URL传递，POST放在Request body中。

HTTP的底层是TCP/IP。所以GET和POST的底层也是TCP/IP，也就是说，GET/POST都是TCP链接。

- GET产生一个TCP数据包；POST产生两个TCP数据包。
  - 对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；
  - 而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据）

（三）、HTTPS通信原理

HTTPS（Secure Hypertext Transfer Protocol）安全超文本传输协议 它是一个安全通信通道

HTTPS是HTTP over SSL/TLS，HTTP是应用层协议，TCP是传输层协议，在应用层和传输层之间，增加了一个安全套接层SSL。

服务器 用RSA生成公钥和私钥把公钥放在证书里发送给客户端，私钥自己保存客户端首先向一个权威的服务器检查证书的合法性，如果证书合法，客户端产生一段随机数，这个随机数就作为通信的密钥，我们称之为对称密钥，用公钥加

密这段随机数，然后发送到服务器服务器用密钥解密获取对称密钥，然后，双方就已对称密钥进行加密解密通信了。

（四）、Https的作用

- 内容加密 建立一个信息安全通道，来保证数据传输的安全；
- 身份认证 确认网站的真实性
- 数据完整性 防止内容被第三方冒充或者篡改

Https和Http的区别

- https协议需要到CA申请证书。
- http是超文本传输协议，信息是明文传输；https 则是具有安全性的ssl加密传输协议。
- http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。
- http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

二、http客户端实现

Go语言标准库内置了net/http包，涵盖了HTTP客户端和服务端的具体实现。

内置的net/http包，提供了最简洁的HTTP客户端实现，无需借助第三方网络通信库就可以直接使用HTTP中用的最多的GET和POST方式请求数据。

- http.Get()
- http.Post()
- http.PostForm()
- http.Head()
- http.Do()

（一）、客户端基本方法

1、一个简单的get实现

```
func main() {
  requestUrl := "http://www.baidu.com"
  response, err := http.Get(requestUrl)
```

```
  if err != nil {
    fmt.Println("err:",err);
  }
  defer response.Body.Close();
  fmt.Println(response.Body)
}
```

err: Get www.baidu.com: unsupported protocol scheme ""
错误：不支持scheme为空的协议，这是什么情况呢，
requestUrl改为requestUrl := "http://www.baidu.com"
scheme反应出的就是协议名称
大家可以看到打印的是 &{0xc420124200 <nil> <nil>}
Body是io.ReadCloser这个类型的对象
response中的参数一般也比较多，我们需要的最多的通常是Body参数，用ioutil.ReadAll去转化io.ReadCloser类型，输出是byty[]，再通过string()强制转换就能看到string了

```
readBody, _ := ioutil.ReadAll(response.Body)
fmt.Println(string(readBody))
```

咱们这个是正确获取到了服务器响应的数据才能这么打印出来了，加入失败了呢，因此需要判断返回码，再去获取body的数据。

```
    if response.StatusCode == 200 {
        r, err := ioutil.ReadAll(response.Body)
        if err != nil {
            fmt.Println(err)
        }
        return string(r)
    }
```

2、方式一 使用http.NewRequest
先生成http.client -> 再生成 http.request -> 之后提交请求：client.Do(request) -> 处理返回结果，每一步的过程都可以设置一些具体的参数

3、方式二 先生成client，之后用client.get/post..
client结构自己也有一些发送api的方法，比如
client.get,client.post,client.postform..等等。基本上涵盖了主要的http请求的类

型，通常不进行什么特殊的配置的话，这样就可以了，其实client的get或者post方法，也是对http.Newerequest方法的封装，里面还额外添加了req.Header.Set("Content-Type", bodyType)一般用的话，也是ok的

4、方式三 http. Get/Post..
具体实现的时候，还是采用的方式一模式，先生成一个默认的client，之后调用http.Newrequest方法。

```
http.NewRequest
	client := &http.Client{}

	request, err := http.NewRequest("GET", requestUrl, nil)
	if err != nil {
		fmt.Println(err)
	}

	cookie := &http.Cookie{Name: "userId", Value: strconv.Itoa(12345)}
	request.AddCookie(cookie) //request中添加cookie

	//设置request的header
	request.Header.Set("Content-Type", "text/html")
	//Content-Length等

	response, err := client.Do(request)
```

5、Post

```
postvalue := url.Values{
  "theCityName": {"天津"},
}

body := bytes.NewBufferString(postvalue.Encode())
response, err := http.Post(requestUrl, "application/x-www-form-urlencoded", body) //Post方法
```

(二)、高级封装

三、http服务端实现

（一）、服务端代码实现

```
func ServeHTTP(w http.ResponseWriter, r *http.Request) {
	r.ParseForm()
	fmt.Println("PATH: ", r.URL.Path)
	fmt.Println("HOST: ", r.URL.Host)
	fmt.Println("METHOD: ", r.Method)
	fmt.Println()

	//fmt.Fprintf(w, "<h1>Index Page</h1>")
	io.WriteString(w, "<h1>Index Page3</h1>")
}

func main() {
	http.HandleFunc("/", ServeHTTP)
	err := http.ListenAndServe(":9000", nil)
	if err != nil {
		log.Fatal("ERROR: ", err)
	}
}
```

函数func ListenAndServe(addr string, handler Handler) error内部进行封装更简洁地实现HTTP服务器。你可以看到它创建了Server类的一个对象，然后调用了刚才说的第一个函数。第2个参数是一个Hander，它是一个接口。这个接口很简单，只要某个struct有ServeHTTP(http.ResponseWriter, *http.Request)这个方法，那这个struct就自动实现了Hander接口。

ServeHTTP方法，他需要2个参数，一个是http.ResponseWriter，另一个是http.Request往http.ResponseWriter写入什么内容，浏览器的网页源码就是什么内容。http.Request里面是封装了，浏览器发过来的请求（包含路径、浏览器类

型等等）。

OK，作为服务器我们会处理很多的请求，那下一步如何处理呢，难道switch r.URL.Path的值吗？那得多辛苦。那有什么好的办法呢，这点go官方已经考虑到这点帮我们提供了一个方法叫做ServeMux，去分发任务。

下面介绍下ServeMux
ServeMux大致作用是，他有一张map表，map里的key记录的是r.URL.String()，而value记录的是一个方法，这个方法和ServeHTTP是一样的，这样ServeMux是实现Handler接口的。这个方法有一个别名，叫HandlerFunc。

```
func main() {
	mux := http.NewServeMux()
	mux.HandleFunc("/h", func(w http.ResponseWriter, r *http.Request) {
		io.WriteString(w, "hello")
	})
	mux.HandleFunc("/bye", func(w http.ResponseWriter, r *http.Request) {
		io.WriteString(w, "byebye")
	})

	mux.HandleFunc("/baidu", func(w http.ResponseWriter, r *http.Request)
{
		http.Redirect(w, r, "http://www.baidu.com",
http.StatusTemporaryRedirect)
	})
	mux.HandleFunc("/hello", sayhello)
	http.ListenAndServe(":9000", mux)
}

func sayhello(w http.ResponseWriter, r *http.Request) {
	io.WriteString(w, "hello world")
}
```

回到开头，有让大家先忘掉http.HandleFunc("/", HandleIndex) 当http.ListenAndServe(":9000", nil)的第2个参数是nil时，http内部会自己建立一个叫DefaultServeMux的ServeMux，因为这个ServeMux是http自己维护的，如果要向这个ServeMux注册的话，就要用http.HandleFunc这个方法啦，现在看很

简单吧。
func FileServer
func FileServer(root FileSystem) Handler
FileServer返回一个使用FileSystem接口root提供文件访问服务的HTTP处理器。
要使用操作系统的FileSystem接口实现，可使用http.Dir：

```
    log.Fatal(http.ListenAndServe(":8080",
http.FileServer(http.Dir("/usr/share/doc"))))
```

（二）、服务端获取客户端请求数据
1. 获取GET参数
比较常见的是如下方式获取：

```
r.ParseForm()
if len(r.Form["id"]) > 0 {
   fmt.Fprintln(w, r.Form["id"][0])
}
```

其中r表示*http.Request类型，w表示http.ResponseWriter类型。
r.Form是url.Values字典类型，r.Form["id"]取到的是一个数组类型。因为http.request在解析参数的时候会将同名的参数都放进同一个数组里，所以这里要用[0]获取到第一个。

2. 获取POST参数
这里要分两种情况：
普通的post表单请求，Content-Type=application/x-www-form-urlencoded 有文件上传的表单，Content-Type=multipart/form-data
第一种情况比较简单，直接用PostFormValue就可以取到了。

```
fmt.Fprintln(w, r.PostFormValue("id"))
```

第二种文件上传的表单这里不作讲解。

3. 获取COOKIE参数

```
cookie, err := r.Cookie("id")
if err == nil {
   fmt.Fprintln(w, "Domain:", cookie.Domain)
   fmt.Fprintln(w, "Expires:", cookie.Expires)
   fmt.Fprintln(w, "Name:", cookie.Name)
```

```
    fmt.Fprintln(w, "Value:", cookie.Value)
}
```

r.Cookie返回*http.Cookie类型，可以获取到domain、过期时间、值等数据。

注意

application/x-www-form-urlencoded与 text/html 的区别

若需使用r.PostForm、r.Form等方法，必须先调用r.ParseForm()

客户端使用post提交数据到body时，若服务端需使用r.PostForm、r.Form等方法，客户端的请求体必须使用"Content-Type":"application/x-www-form-urlencoded") 类型，并且r.ParseForm()须放在ioutil.ReadAll(r.Body)解析body之前。否则数据将在body中。