尚硅谷区块链技术之 GoWeb

第 1 章：简介

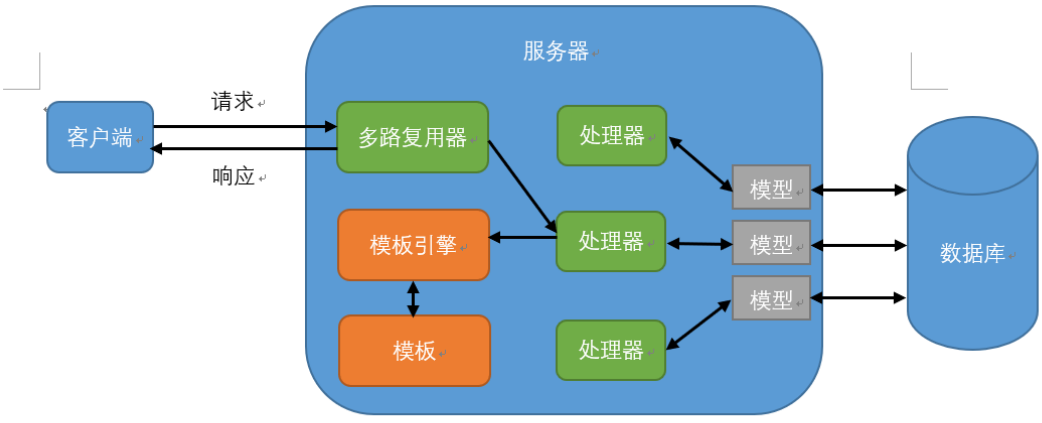
1.1 Web 应用简介

Web 应用在我们的生活中无处不在。看看我们日常使用的各个应用程序，它们要么是 Web 应用，要么是移动 App 这类 Web 应用的变种。无论哪一种编程语言，只要它能够开发出与人类交互的软件，它就必然会支持 Web 应用开发。对一门崭新的编程语言来说，它的开发者首先要做的一件事，就是构建与互联网(Internet)和万维网(World Wide Web)交互的库(library)和框架，而那些更为成熟的编程语言还会有各种五花八门的 Web 开发工具。

Go 是一门刚开始崭露头角的语言，它是为了让人们能够简单而高效地编写后端系统而创建的。这门语言拥有众多的先进特性，如**函数式编程方面的特性**、内置了对**并发编程**的支持、现代化的**包管理系统**、**垃圾收集特性**、以及一些包罗万象威力**强大的标准库**，而且如果需要我们还可以引入第三方开源库。

使用 Go 语言进行 Web 开发正变得日益流行，很多大公司都在使用，如 Google、Facebook、腾讯、百度、阿里巴巴、京东、小米以及 360、美团、滴滴以及新浪等。

1.2 Web 应用的工作原理



Web 应用简单工作流程

1.3 Hello World

下面，就让我们使用 Go 语言创建一个简单的 Web 应用。

1）在 GOPATH 下的 src 目录下创建一个 webapp 的文件夹，并在该目录中创建一个 main.go 的文件，代码如下

```
package main


import (
    "fmt"
    "net/http"
)


//创建处理器函数
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello World!", r.URL.Path)
}


func main() {


    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

2）在终端执行以下命令（使用 Vscode 开发工具时）：

a) **方式一（建议使用）**：在 webapp 目录中**右键→在命令提示符中打开**

执行 **go build main.go** 命令；然后在当前目录中就会生成一个 **main.exe** 的二进制可执行文件；最后再执行 **./main.exe** 就可以启动服务器

b) 方式二：在 webapp 目录中**右键→在命令提示符中打开**

执行 **go install webapp** 命令；然后在 bin 目录中会生成一个 **webapp.exe** 的二进制可执行文件；进入 bin 目录之后再 bin 目录中执行 **./webapp.exe** 就可以启动服务器

3） 在浏览器地址栏输入 http://localhost:8080，在浏览器中就会显示 Hello World! / 在浏览器地址栏输入 http://localhost:8080/hello，在浏览器中就会显示 Hello World! /hello

第 2 章：Web 服务器的创建

2.1 简介

Go 提供了一系列用于创建 Web 服务器的标准库，而且通过 Go 创建一个服务器的步骤非常简单，只要通过 net/http 包调用 **ListenAndServe** 函数并传入**网络地址**以及负责处理请求的**处理器**(handler)作为参数就可以了。如果网络地址参数为空字符串，那么服务器默认使用 80 端口进行网络连接；如果处理器参数为 nil，那么服务器将使用默认的多路复用器 **DefaultServeMux**，当然，我们也可以通过调用 **NewServeMux** 函数创建一个多路复用器。多路复用器接收到用户的请求之后根据请求的 URL 来判断使用哪个**处理器**来处理请求，找到后就会重定向到对应的处理器来处理请求，

2.2 使用默认的多路复用器（DefaultServeMux）

1） 使用处理器函数处理请求

```
package main


import (
    "fmt"
    "net/http"
)


//创建处理器函数
```

```
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "正在通过处理器函数处理你的请求")
}

func main() {

    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

a) HandleFunc 方法的说明

func HandleFunc

```
func HandleFunc(pattern string, handler func(ResponseWriter, *Request))
```

HandleFunc注册一个处理器函数handler和对应的模式pattern（注册到DefaultServeMux）。ServeMux的文档解释了模式的匹配机制。

b) **处理器函数的实现原理**：

- Go 语言拥有一种 HandlerFunc 函数类型，它可以将一个带有正确签名的函数 f 转换成一个带有方法 f 的 Handler。

type HandlerFunc

```
type HandlerFunc func(ResponseWriter, *Request)
```

HandlerFunc type是一个适配器，通过类型转换让我们可以将普通的函数作为HTTP处理器使用。如果f是一个具有适当签名的函数，HandlerFunc(f)通过调用f实现了Handler接口。

2） 使用处理器处理请求

```
package main

import (
    "fmt"
    "net/http"
)

type MyHandler struct{}
```

```
func (h *MyHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "正在通过处理器处理你的请求")
}

func main() {
    myHandler := MyHandler{}
    //调用处理器
    http.Handle("/", &myHandler)
    http.ListenAndServe(":8080", nil)
}
```

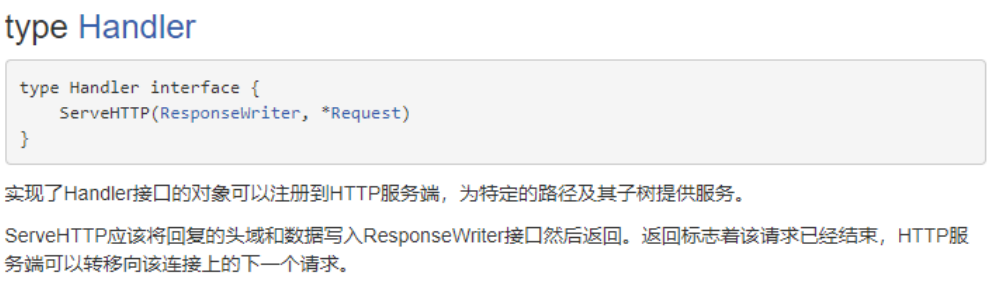
i. Handle 方法的说明

func Handle

```
func Handle(pattern string, handler Handler)
```

Handle注册HTTP处理器handler和对应的模式pattern（注册到DefaultServeMux）。如果该模式已经注册有一个处理器，Handle会panic。ServeMux的文档解释了模式的匹配机制。

ii. 关于处理器 Handler 的说明



type Handler

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

实现了Handler接口的对象可以注册到HTTP服务端，为特定的路径及其子树提供服务。

ServeHTTP应该将回复的头域和数据写入ResponseWriter接口然后返回。返回标志着该请求已经结束，HTTP服务端可以转移向该连接上的下一个请求。

- 也就是说只要某个结构体实现了 Handler 接口中的 ServeHTTP 方法那么它就是一个处理器

iii. 我们还可以通过 Server 结构对服务器进行更详细的配置

type Server

```
type Server struct {
    Addr           string        // 监听的TCP地址，如果为空字符串会使用":http"
    Handler        Handler       // 调用的处理器，如为nil会调用http.DefaultServeMux
    ReadTimeout    time.Duration // 请求的读取操作在超时前的最大持续时间
    WriteTimeout   time.Duration // 回复的写入操作在超时前的最大持续时间
    MaxHeaderBytes int           // 请求的头域最大长度，如为0则用DefaultMaxHeaderBytes
    TLSConfig      *tls.Config   // 可选的TLS配置，用于ListenAndServeTLS方法
    // TLSNextProto（可选地）指定一个函数来在一个NPN型协议升级出现时接管TLS连接的所有权。
    // 映射的键为商谈的协议名；映射的值为函数，该函数的Handler参数应处理HTTP请求，
    // 并且初始化Handler.ServeHTTP的*Request参数的TLS和RemoteAddr字段（如果未设置）。
    // 连接在函数返回时会自动关闭。
    TLSNextProto map[string]func(*Server, *tls.Conn, Handler)
    // ConnState字段指定一个可选的回调函数，该函数会在一个与客户端的连接改变状态时被调用。
    // 参见ConnState类型和相关常数获取细节。
    ConnState func(net.Conn, ConnState)
    // ErrorLog指定一个可选的日志记录器，用于记录接收连接时的错误和处理器不正常的行为。
    // 如果本字段为nil，日志会通过log包的标准日志记录器写入os.Stderr。
    ErrorLog *log.Logger
    // 内含隐藏或非导出字段
}
```

Server类型定义了运行HTTP服务端的参数。Server的零值是合法的配置。

```
package main

import (
    "fmt"
    "net/http"
    "time"
)

type MyHandler struct{}

func (h *MyHandler) ServeHTTP(w http.ResponseWriter, r
*http.Request) {
    fmt.Fprintln(w, "测试通过 Server 结构详细配置服务
器")
}

func main() {
    myHandler := MyHandler{}
```

```
        //通过 Server 结构对服务器进行更详细的配置
        server := http.Server{
            Addr:        ":8080",
            Handler:     &myHandler,
            ReadTimeout: 2 * time.Second,
        }
        server.ListenAndServe()
    }
```

2.3 使用自己创建的多路复用器

1) 在创建服务器时，我们还可以通过 **NewServeMux** 方法创建一个多路复用器

func NewServeMux

```
func NewServeMux() *ServeMux
```

NewServeMux创建并返回一个新的*ServeMux

```
package main

import (
    "fmt"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "通过自己创建的多路复用器来处理请求")
}

func main() {
    mux := http.NewServeMux()
```

```
        mux.HandleFunc("/myMux", handler)


        http.ListenAndServe(":8080", mux)


}
```

结构体 ServeMux

type ServeMux

```
type ServeMux struct {
    // 内含隐藏或非导出字段
}
```

ServeMux类型是HTTP请求的多路转接器。它会将每一个接收的请求的URL与一个注册模式的列表进行匹配，并调用和URL最匹配的模式的处理器。

模式是固定的、由根开始的路径，如"/favicon.ico"，或由根开始的子树，如"/images/"（注意结尾的斜杠）。较长的模式优先于较短的模式，因此如果模式"/images/"和"/images/thumbnails/"都注册了处理器，后一个处理器会用于路径以"/images/thumbnails/"开始的请求，前一个处理器会接收到其余的路径在"/images/"子树下的请求。

注意，因为以斜杠结尾的模式代表一个由根开始的子树，模式"/"会匹配所有的未被其他注册的模式匹配的路径，而不仅仅是路径"/"。

模式也能（可选地）以主机名开始，表示只匹配该主机上的路径。指定主机的模式优先于一般的模式，因此一个注册了两个模式"/codesearch"和"codesearch.google.com/"的处理器不会接管目标为"http://www.google.com/"的请求。

ServeMux还会注意到请求的URL路径的无害化，将任何路径中包含"."或".."元素的请求重定向到等价的没有这两种元素的URL。（参见path.Clean函数）

结构体 ServeMux 的相关方法

func (*ServeMux) Handle

```
func (mux *ServeMux) Handle(pattern string, handler Handler)
```

Handle注册HTTP处理器handler和对应的模式pattern。如果该模式已经注册有一个处理器，Handle会panic。

Example

func (*ServeMux) HandleFunc

```
func (mux *ServeMux) HandleFunc(pattern string, handler func(ResponseWriter, *Request))
```

HandleFunc注册一个处理器函数handler和对应的模式pattern。

func (*ServeMux) Handler

```
func (mux *ServeMux) Handler(r *Request) (h Handler, pattern string)
```

Handler根据r.Method、r.Host和r.URL.Path等数据，返回将用于处理该请求的HTTP处理器。它总是返回一个非nil的处理器。如果路径不是它的规范格式，将返回内建的用于重定向到等价的规范路径的处理器。

Handler也会返回匹配该请求的的已注册模式；在内建重定向处理器的情况下，pattern会在重定向后进行匹配。如果没有已注册模式可以应用于该请求，本方法将返回一个内建的"404 page not found"处理器和一个空字符串模式。

func (*ServeMux) ServeHTTP

```
func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request)
```

ServeHTTP将请求派遣到与请求的URL最匹配的模式对应的处理器。

第 3 章：HTTP 协议

因为编写 Web 应用必须对 HTTP 有所了解，所以接下来我们对 HTTP 进行介绍。

3.1 HTTP 协议简介

HTTP **超文本传输协议**（HTTP-Hypertext transfer protocol），是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。它于 1990 年提出，经过几年的使用与发展，得到不断地完善和扩展。**它是一种详细规定了浏览器和万维网服务器之间互相通信的规则**，通过因特网传送万维网文档的数据传送协议。

客户端与服务端通信时传输的内容我们称之为**报文**

HTTP 就是一个通信规则，这个规则规定了客户端发送给服务器的报文格式，也规定了服务器发送给客户端的报文格式。实际我们要学习的就是这两种报文。客户端发送给服务器的称为”**请求报文**“，服务器发送给客户端的称为”**响应报文**“。

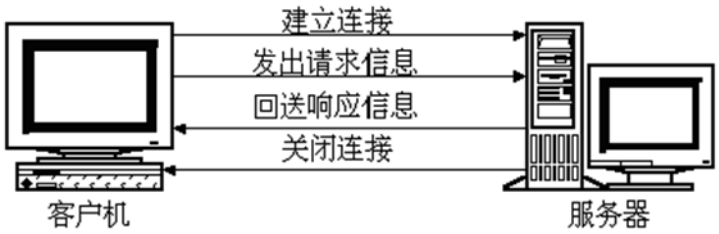
3.2 HTTP 协议的发展历程

超文本传输协议的前身是世外桃源(Xanadu)项目，超文本的概念是泰德·纳尔森(Ted Nelson)在 1960 年代提出的。进入哈佛大学后，纳尔森一直致力于超文本协议和该项目的研究，但他从未公开发表过资料。1989 年，**蒂姆·伯纳斯·李**(Tim Berners Lee)在 CERN(欧洲原子核研究委员会 = European Organization for Nuclear Research)担任软件咨询师的时候，开发了一套程序，**奠定了万维网(WWW = World Wide Web)的基础**。1990 年 12 月，超文本在 CERN 首次上线。1991 年夏天，继 Telnet 等协议之后，超文本转移协议成为互联网诸多协议的一分子。

当时，Telnet 协议解决了一台计算机和另外一台计算机之间一对一的控制型通信的要求。邮件协议解决了一个发件人向少量人员发送信息的通信要求。文件传输协议解决一台计算机从另外一台计算机批量获取文件的通信要求，但是它不具备一边获取文件一边显示文件或对文件进行某种处理的功能。新闻传输协议解决了一对多新闻广播的通信要求。而超文本要解决的通信要求是：在一台计算机上获取并显示存放在多台计算机里的文本、数据、图片和其他类型的文件；它包含两大部分：超文本转移协议和超文本标记语言(HTML)。HTTP、HTML 以及浏览器的诞生给互联网的普及带来了飞跃。

3.3 HTTP 协议的会话方式

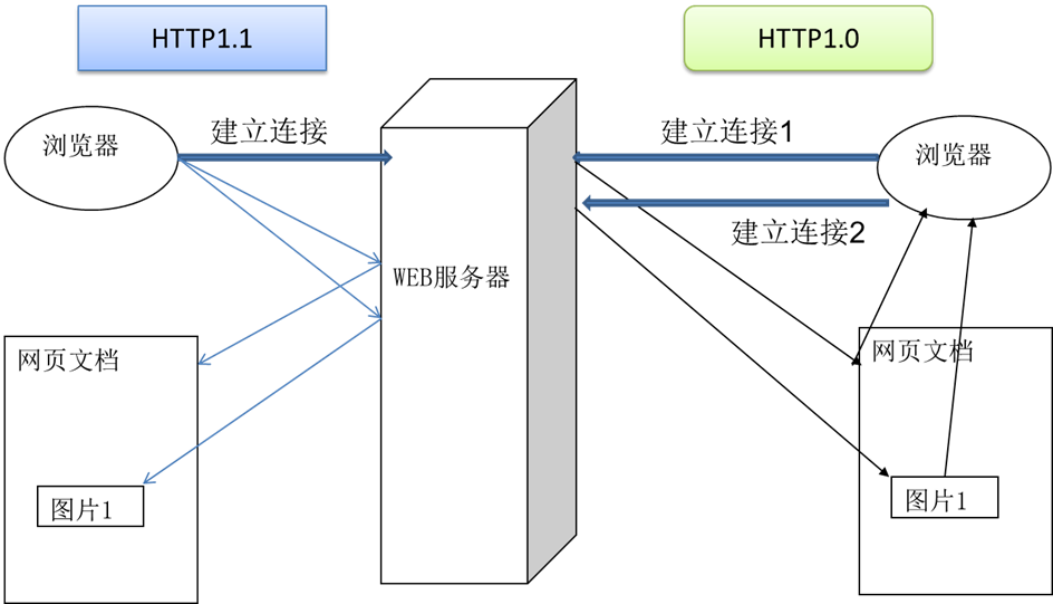
- 浏览器与服务器之间的通信过程要经历**四个步骤**



- 浏览器与 WEB 服务器的连接过程是短暂的，每次连接只处理一个请求和响应。对每一个页面的访问，浏览器与 WEB 服务器都要建立一次单独的连接。
- 浏览器到 WEB 服务器之间的所有通讯都是完全独立分开的请求和响应对。

3.4 HTTP1.0 和 HTTP1.1 的区别

在 HTTP1.0 版本中，浏览器请求一个带有图片的网页，会由于下载图片而与服务器之间开启一个新的连接；但在 HTTP1.1 版本中，允许浏览器在拿到当前请求对应的全部资源后再断开连接，提高了效率。

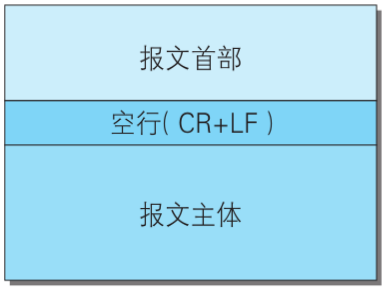


HTTP 1.1 是目前使用最为广泛的一个版本，而**最新的一个版本则是 HTTP 2.0**，又称 HTTP/2。在开放互联网上 HTTP 2.0 将只用于 https://网址。HTTPS,即 SSL（Secure Socket Layer, 安全套接字层）之上的 HTTP,实际上就是在 SSL/TLS 连接的上层进行 HTTP 通信。

备注：SSL 最初由 Netscape 公司开发，之后由 IETF（Internet Engineering Task Force，互联网工程任务组）接手并将其改名为 TLS（Transport Layer Security，传输层安全协议）

3.5 报文

3.5.1 报文格式



【报文首部】
服务器端或客户端需处理的请求或响应的内容及属性

【CR+LF】
CR(Carriage Return，回车符：16进制 0x0d)和
LF(Line Feed，换行符：16进制 0x0a)

【报文主体】
应被发送的数据

3.5.2 请求报文

1） 报文格式

请求首行（**请求行**）；

请求头信息（**请求头**）；

空行；

请求体；

2） Get 请求

```
GET /Hello/index.jsp HTTP/1.1
Accept: */*
Accept-Language: zh-CN
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; WOW64;
Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729;
Media Center PC 6.0; .NET4.0C; .NET4.0E)
Accept-Encoding: gzip, deflate
Host: localhost:8080
Connection: Keep-Alive
Cookie: JSESSIONID=C55836CDA892D9124C03CF8FE8311B15
```

- **Get 请求没有请求体，Post 请求才有请求体**

- GET /Hello/index.jsp HTTP/1.1：GET 请求，请求服务器路径为 Hello/index.jsp，协议为 1.1；
- Host:localhost：请求的主机名为 localhost；
- User-Agent: Mozilla/4.0 (compatible; MSIE 8.0…：与浏览器和 OS 相关的信息。有些网站会显示用户的系统版本和浏览器版本信息，这都是通过获取 User-Agent 头信息而来的；
- Accept: */*：告诉服务器，当前客户端可以接收的文档类型， */*，就表示什么都可以接收；
- Accept-Language: zh-CN：当前客户端支持的语言，可以在浏览器的工具→选项中找到语言相关信息；
- Accept-Encoding: gzip, deflate：支持的压缩格式。数据在网络上传递时，可能服务器会把数据压缩后再发送；
- Connection: keep-alive：客户端支持的链接方式，保持一段时间链接，默认为 3000ms；
- Cookie: JSESSIONID=369766FDF6220F7803433C0B2DE36D98：因为不是第一次访问这个地址，所以会在请求中把上一次服务器响应中发送过来的 Cookie 在请求中一并发送过去。

3） Post 请求

POST 请求要求将 form 标签的 method 的属性设置为 post

```
<form action="target.html" method="post">
    用户名：<input name="username" type="text" />
    <input type="submit" value="提交" />
</form>
```

```
POST /Hello/target.html HTTP/1.1
Accept: application/x-ms-application, image/jpeg, application/xaml+xml, image/gif, image/pjpeg, application/x-ms-xbap, */*
Referer: http://localhost:8080/Hello/
Accept-Language: zh-CN
```

```
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; WOW64; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; .NET4.0E)
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
Host: localhost:8080
Content-Length: 14
Connection: Keep-Alive
Cache-Control: no-cache
Cookie: JSESSIONID=774DA38C1B78AE288610D77621590345


username=admin
```

- Referer: http://localhost:8080/hello/index.jsp：请求来自哪个页面，例如你在百度上点击链接到了这里，那么 Referer:http://www.baidu.com；如果你是在浏览器的地址栏中直接输入的地址，那么就没有 Referer 这个请求头了；
- Content-Type: application/x-www-form-urlencoded：表单的数据类型，说明会使用 url 格式编码数据；url 编码的数据都是以 “%” 为前缀，后面跟随两位的 16 进制，例如 “传智” 这两个字使用 UTF-8 的 url 编码用为 “%E4%BC%A0%E6%99%BA”；
- Content-Length:13：请求体的长度，这里表示 13 个字节。
- keyword=hello：请求体内容！hello 是在表单中输入的数据，keyword 是表单字段的名字。

3.5.3 响应报文

1） 报文格式

```
响应首行（响应行）；
响应头信息（响应头）；
空行；
响应体；
```

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=UTF-8
Content-Length: 274
Date: Tue, 07 Apr 2015 10:08:26 GMT

    <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" >
    <html>
    <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Insert title here</title>
    </head>
    <body>
    <h1>Hello</h1>
    </body>
</html>
```

- HTTP/1.1 200 OK：响应协议为 HTTP1.1，状态码为 200，表示请求成功；
- Server: Apache-Coyote/1.1：服务器的版本信息；
- Content-Type: text/html;charset=UTF-8：响应体使用的编码为 UTF-8；
- Content-Length: 274：响应体为 274 字节；
- Date: Tue, 07 Apr 2015 10:08:26 GMT：响应的时间，这可能会有 8 小时的时区差；

3.6 响应状态码

1） 状态码用来告诉 HTTP 客户端,HTTP 服务器是否产生了预期的 Response。HTTP/1.1 协议中定义了 5 类状态码， 状态码由三位数字组成，第一个数字定义了响应的类别

- 1XX 提示信息 - 表示请求已被成功接收，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处理
- 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

2） 响应码对浏览器来说很重要，它告诉浏览器响应的结果，常见的状态码有：

- 200：请求成功，浏览器会把响应体内容（通常是 html）显示在浏览器中；
- 404：请求的资源没有找到，说明客户端错误的请求了不存在的资源；
- 500：请求资源找到了，但服务器内部出现了错误；
- 302 :重定向，当响应码为 302 时，表示服务器要求浏览器重新再发一个请求，服务器会发送一个响应头 Location，它指定了新请求的 URL 地址；

第 4 章：操作数据库

Go 语言中的 **database/sql** 包定义了对数据库的一系列操作。database/sql/driver 包定义了应被数据库驱动实现的接口，这些接口会被 sql 包使用。但是 Go 语言没有提供任何官方的数据库驱动，所以我们需要导入**第三方**的数据库驱动。不过我们连接数据库之后对数据库操作的大部分代码都使用 sql 包。

4.1 获取数据库连接

1） 创建一个 db.go 文件，导入 database/sql 包以及第三方驱动包

```
import (
    "database/sql"

    _ "github.com/go-sql-driver/mysql"
)
```

2） 定义两个全局变量

```
var (
    Db *sql.DB
    err    error
)
```

```
)
```

- DB 结构体的说明

type DB

```
type DB struct {
    // 内含隐藏或非导出字段
}
```

DB是一个数据库（操作）句柄，代表一个具有零到多个底层连接的连接池。它可以安全的被多个go程同时使用。

sql包会自动创建和释放连接；它也会维护一个闲置连接的连接池。如果数据库具有单连接状态的概念，该状态只有在事务中被观察时才可信。一旦调用了BD.Begin，返回的Tx会绑定到单个连接。当调用事务Tx的Commit或Rollback后，该事务使用的连接会归还到DB的闲置连接池中。连接池的大小可以用SetMaxIdleConns方法控制。

3) 创建 init 函数，在函数体中调用 sql 包的 Open 函数获取连接

```
func init() {
    Db, err = sql.Open("mysql", "root:root@tcp(localhost:3306)/test")
    if err != nil {
        panic(err.Error())
    }
}
```

- Open 函数的说明

func Open

```
func Open(driverName, dataSourceName string) (*DB, error)
```

Open打开一个dirverName指定的数据库，dataSourceName指定数据源，一般包至少括数据库文件名和（可能的）连接信息。

- 参数 dataSourceName 的格式：

  数据库用户名:数据库密码@[tcp(localhost:3306)]/数据库名

- Open 函数可能只是验证其参数，**而不创建与数据库的连接**。如果要检查数据源的名称是否合法，应调用返回值的 Ping 方法。

func (*DB) Ping

```
func (db *DB) Ping() error
```

Ping检查与数据库的连接是否仍有效，如果需要会创建连接。

- 返回的 DB 可以安全的被多个 go 程同时使用，并会维护自身的闲置连接池。这样一来，**Open 函数只需调用一次。很少需要关闭 DB**

4.2 增删改操作

1) 在连接的 test 数据库中创建一个 users 表

```
CREATE TABLE users(
id INT PRIMARY KEY AUTO_INCREMENT,
username VARCHAR(100) UNIQUE NOT NULL,
PASSWORD VARCHAR(100) NOT NULL,
email VARCHAR(100)
)
```

2) 向 users 表中插入一条记录

- 创建 user.go 文件，文件中编写一下代码

```
package model

import (
    "fmt"
    "webproject/utils"
)

type User struct {
    ID       int
    Username string
    Password string
    Email    string
}

func (user *User) AddUser() error {
    // 写 sql 语句
    sqlStr := "insert into users(username , password , email) values(?,?,?)"
```

```
        //预编译
        stmt, err := utils.Db.Prepare(sqlStr)
        if err != nil {
            fmt.Println("预编译出现异常：", err)
            return err
        }
        //执行
        _, erro := stmt.Exec(user.Username, user.Password, user.Email)
        if erro != nil {
            fmt.Println("执行出现异常：", erro)
            return erro
        }
        return nil
}
```

- Prepare 方法的说明

func (*DB) Prepare

```
func (db *DB) Prepare(query string) (*Stmt, error)
```

Prepare创建一个准备好的状态用于之后的查询和命令。返回值可以同时执行多个查询和命令。

- Stmt 结构体及它的方法的说明

type Stmt

```
type Stmt struct {
    // 内含隐藏或非导出字段
}
```

Stmt是准备好的状态。Stmt可以安全的被多个go程同时使用。

func (*Stmt) Exec

```
func (s *Stmt) Exec(args ...interface{}) (Result, error)
```

Exec使用提供的参数执行准备好的命令状态，返回Result类型的该状态执行结果的总结。

func (*Stmt) Query

```
func (s *Stmt) Query(args ...interface{}) (*Rows, error)
```

Query使用提供的参数执行准备好的查询状态，返回Rows类型查询结果。

func (*Stmt) QueryRow

```
func (s *Stmt) QueryRow(args ...interface{}) *Row
```

QueryRow使用提供的参数执行准备好的查询状态。如果在执行时遇到了错误，该错误会被延迟，直到返回值的Scan方法被调用时才释放。返回值总是非nil的。如果没有查询到结果，*Row类型返回值的Scan方法会返回ErrNoRows；否则，Scan方法会扫描结果第一行并丢弃其余行。

- DB 结构体中也有 Exec、Query 和 QueryRow 方法

func (*DB) Exec

```
func (db *DB) Exec(query string, args ...interface{}) (Result, error)
```

Exec执行一次命令（包括查询、删除、更新、插入等），不返回任何执行结果。参数args表示query中的占位参数。

func (*DB) Query

```
func (db *DB) Query(query string, args ...interface{}) (*Rows, error)
```

Query执行一次查询，返回多行结果（即Rows），一般用于执行select命令。参数args表示query中的占位参数。

Example

func (*DB) QueryRow

```
func (db *DB) QueryRow(query string, args ...interface{}) *Row
```

QueryRow执行一次查询，并期望返回最多一行结果（即Row）。QueryRow总是返回非nil的值，直到返回值的Scan方法被调用时，才会返回被延迟的错误。（如：未找到结果）

- 所以还可以通过调用 DB 的 Exec 方法添加用户（以下只展示了 user.go 文件修改之后的 AddUser 方法）

```
func (user *User) AddUser() error {
    // 写 sql 语句
    sqlStr := "insert into users(username , password , email) values(?,?,?)"
```

```
        //执行
        _, erro := utils.Db.Exec(sqlStr, user.Username, user.Password, user.Email)
        if erro != nil {
            fmt.Println("执行出现异常：", erro)
            return erro
        }
        return nil
}
```

- 请同学们仿照着添加操作将删除和更新完成

4.3 单元测试

4.3.1 简介

顾名思义，单元测试(unit test)，就是一种为验证单元的正确性而设置的自动化测试，一个单元就是程序中的一个模块化部分。一般来说，一个单元通常会与程序中的一个函数或者一个方法相对应，但这并不是必须的。Go 的单元测试需要用到 **testing** 包以及 **go test** 命令，而且对测试文件也有以下要求

1) 被测试的源文件和测试文件必须**位于同一个包下**
2) 测试文件必须要以**_test.go** 结尾
   - 虽然 Go 对测试文件_test.go 的前缀没有强制要求，不过一般我们都设置为被测试的文件的文件名，如：要对 user.go 进行测试，那么测试文件的名字我们通常设置为 user_test.go
3) 测试文件中的测试函数为 **TestXxx**(t *testing.T)
   - 其中 Xxx 的首字母必须是大写的英文字母
   - 函数参数必须是 test.T 的指针类型（如果是 Benchmark 测试则参数是 testing.B 的指针类型）

4.3.2 测试代码：

1) 在 user_test.go 中测试添加员工的方法

```
func TestAddUser(t *testing.T) {
    fmt.Println("测试添加用户：")
    user := &User{
        Username: "admin3",
        Password: "123456",
        Email:    "admin3@atguigu.com",
    }
    //将 user 添加到数据库中
    user.AddUser()
}
```

- testing 包的说明

package testing

import "testing"

testing 提供对 Go 包的自动化测试的支持。通过 `go test` 命令，能够自动执行如下形式的任何函数：

```
func TestXxx(*testing.T)
```

其中 Xxx 可以是任何字母数字字符串（但第一个字母不能是 [a-z]），用于识别测试例程。

在这些函数中，使用 Error, Fail 或相关方法来发出失败信号。

要编写一个新的测试套件，需要创建一个名称以 _test.go 结尾的文件，该文件包含 `TestXxx` 函数，如上所述。将该文件放在与被测试的包相同的包中。该文件将被排除在正常的程序包之外，但在运行 "go test" 命令时将被包含。 有关详细信息，请运行 "go help test" 和 "go help testflag" 了解。

如果有需要，可以调用 *T 和 *B 的 Skip 方法，跳过该测试或基准测试：

```
func TestTimeConsuming(t *testing.T) {
    if testing.Short() {
        t.Skip("skipping test in short mode.")
    }
    ...
}
```

2) 如果一个测试函数的函数名的不是以 Test 开头，那么在使用 go test 命令时默认不会执行，不过我们可以设置该函数时一个**子测试函数**，可以在其他测试函数里通过 t.Run 方法来执行子测试函数，具体代码如下

```
func TestUser(t *testing.T) {
    t.Run("正在测试添加用户：", testAddUser)
```

```
        t.Run("正在测试获取一个用户：", testGetUserById)
    }
    //子测试函数
    func testAddUser(t *testing.T) {
        fmt.Println("测试添加用户：")
        user := &User{
            Username: "admin5",
            Password: "123456",
            Email:    "admin5@atguigu.com",
        }
        //将 user 添加到数据库中
        user.AddUser()
    }
    //子测试函数
    func testGetUserByID(t *testing.T) {
        u := &User{}
        user, _ := u.GetUserByID(1)
        fmt.Println("用户的信息是：", *user)
}
```

3) 我们还可以通过 **TestMain(m *testing.M)**函数在测试之前和之后做一些其他的操作

a) 测试文件中有 TestMain 函数时，执行 go test 命令将直接运行 TestMain 函数，不直接运行测试函数，只有在 TestMain 函数中执行 **m.Run()**时才会执行测试函数

b) 如果想查看测试的详细过程，可以使用 **go test -v** 命令

c) 具体代码

```
    func TestMain(m *testing.M) {
        fmt.Println("开始测试")
```

```
        m.Run()
    }

    func TestUser(t *testing.T) {
        t.Run("正在测试添加用户：", testAddUser)
    }

    func testAddUser(t *testing.T) {
        fmt.Println("测试添加用户：")
        user := &User{
            Username: "admin5",
            Password: "123456",
            Email:    "admin5@atguigu.com",
        }
        //将 user 添加到数据库中
        user.AddUser()
}
```

- Main 的说明

Main

测试程序有时需要在测试之前或之后进行额外的设置（setup）或拆卸（teardown）。有时，测试还需要控制在主线程上运行的代码。为了支持这些和其他一些情况，如果测试文件包含函数：

```
func TestMain(m *testing.M)
```

那么生成的测试将调用 TestMain(m)，而不是直接运行测试。TestMain 运行在主 goroutine 中，可以在调用 m.Run 前后做任何设置和拆卸。应该使用 m.Run 的返回值作为参数调用 os.Exit。在调用 TestMain 时，flag.Parse 并没有被调用。所以，如果 TestMain 依赖于 command-line 标志 (包括 testing 包的标记)，则应该显示的调用 flag.Parse。

一个简单的 TestMain 的实现：

```
func TestMain(m *testing.M) {
        // call flag.Parse() here if TestMain uses flags
    // 如果 TestMain 使用了 flags，这里应该加上 flag.Parse()
        os.Exit(m.Run())
}
```

4.4 获取一条记录

✓ 根据用户的 id 从数据库中获取一条记录

✓ 代码实现：

```
func (user *User) GetUserByID(userId int) (*User, error) {
    //写 sql 语句
    sqlStr := "select id , username , password , email from users where id = ?"
    //执行 sql
    row := utils.Db.QueryRow(sqlStr, userId)
    //声明三个变量
    var username string
    var password string
    var email string
    //将各个字段中的值读到以上三个变量中
    err := row.Scan(&userId, &username, &password, &email)
    if err != nil {
        return nil, err
    }
    //将三个变量的值赋给 User 结构体
    u := &User{
        ID:       userId,
        Username: username,
        Password: password,
        Email:    email,
    }
    return u, nil
}
```

- Row 结构体及 San 方法的说明

type Row

```
type Row struct {
    // 内含隐藏或非导出字段
}
```

QueryRow方法返回Row，代表单行查询结果。

func (*Row) Scan

```
func (r *Row) Scan(dest ...interface{}) error
```

Scan将该行查询结果各列分别保存进dest参数指定的值中。如果该查询匹配多行，Scan会使用第一行结果并丢弃其余各行。如果没有匹配查询的行，Scan会返回ErrNoRows。

4.5 获取多条记录

- ✓ 从数据库中查询出所有的记录
- ✓ 代码实现：

```
func (user *User) GetUsers() ([]*User, error) {
    //写 sql 语句
    sqlStr := "select id , username , password , email from users"
    //执行 sql
    rows, err := utils.Db.Query(sqlStr)
    if err != nil {
        return nil, err
    }
    //定义一个 User 切片
    var users []*User
    //遍历
    for rows.Next() {
        //声明四个个变量
        var userID int
        var username string
        var password string
        var email string
        //将各个字段中的值读到以上三个变量中
```

```
            err := rows.Scan(&userID, &username, &password, &email)
            if err != nil {
                return nil, err
            }
            //将三个变量的值赋给 User 结构体
            u := &User{
                ID:          userID,
                Username: username,
                Password: password,
                Email:      email,
            }
            //将 u 添加到 users 切片中
            users = append(users, u)
        }
        return users, nil
}
```

- Rows 结构体说明

type Rows

```
type Rows struct {
    // 内含隐藏或非导出字段
}
```

Rows是查询的结果。它的游标指向结果集的第零行，使用Next方法来遍历各行结果：

```
rows, err := db.Query("SELECT ...")
...
defer rows.Close()
for rows.Next() {
    var id int
    var name string
    err = rows.Scan(&id, &name)
    ...
}
err = rows.Err() // get any error encountered during iteration
...
```

- Next 方法和 Scan 方法说明

func (*Rows) Scan

```
func (rs *Rows) Scan(dest ...interface{}) error
```

Scan将当前行各列结果填充进dest指定的各个值中。

如果某个参数的类型为*[]byte，Scan会保存对应数据的拷贝，该拷贝为调用者所有，可以安全的,修改或无限期的保存。如果参数类型为*RawBytes可以避免拷贝；参见RawBytes的文档获取其使用的约束。

如果某个参数的类型为*interface{}，Scan会不做转换的拷贝底层驱动提供的值。如果值的类型为[]byte，会进行数据的拷贝，调用者可以安全使用该值。

func (*Rows) Next

```
func (rs *Rows) Next() bool
```

Next准备用于Scan方法的下一行结果。如果成功会返回真，如果没有下一行或者出现错误会返回假。Err应该被调用以区分这两种情况。

每一次调用Scan方法，甚至包括第一次调用该方法，都必须在前面先调用Next方法。

第 5 章：处理请求

Go 语言的 net/http 包提供了一系列用于表示 HTTP 报文的结构，我们可以使用它处理请求和发送相应，其中 Request 结构代表了客户端发送的请求报文，下面让我们看一下 Request 结构体

type Request

```
type Request struct {
    // Method指定HTTP方法（GET、POST、PUT等）。对客户端，""代表GET。
    Method string
    // URL在服务端表示被请求的URI，在客户端表示要访问的URL。
    //
    // 在服务端，URL字段是解析请求行的URI（保存在RequestURI字段）得到的，
    // 对大多数请求来说，除了Path和RawQuery之外的字段都是空字符串。
    // （参见RFC 2616, Section 5.1.2）
    //
    // 在客户端，URL的Host字段指定了要连接的服务器，
    // 而Request的Host字段（可选地）指定要发送的HTTP请求的Host头的值。
    URL *url.URL
    // 接收到的请求的协议版本。本包生产的Request总是使用HTTP/1.1
    Proto      string // "HTTP/1.0"
    ProtoMajor int    // 1
    ProtoMinor int    // 0
    // Header字段用来表示HTTP请求的头域。如果头域（多行键值对格式）为:
    //  accept-encoding: gzip, deflate
    //  Accept-Language: en-us
    //  Connection: keep-alive
    // 则:
    //  Header = map[string][]string{
    //          "Accept-Encoding": {"gzip, deflate"},
    //          "Accept-Language": {"en-us"},
    //          "Connection": {"keep-alive"},
    //  }
    // HTTP规定头域的键名（头名）是大小写敏感的，请求的解析器通过规范化头域的键名来实现这点。
    // 在客户端的请求，可能会被自动添加或重写Header中的特定的头，参见Request.Write方法。
    Header Header
    // Body是请求的主体。
    //
```

```
    // 在客户端，如果Body是nil表示该请求没有主体买入GET请求。
    // Client的Transport字段会负责调用Body的Close方法。
    //
    // 在服务端，Body字段总是非nil的；但在没有主体时，读取Body会立刻返回EOF。
    // Server会关闭请求的主体，ServeHTTP处理器不需要关闭Body字段。
    Body io.ReadCloser
    // ContentLength记录相关内容的长度。
    // 如果为-1，表示长度未知，如果>=0，表示可以从Body字段读取ContentLength字节数据。
    // 在客户端，如果Body非nil而该字段为0，表示不知道Body的长度。
    ContentLength int64
    // TransferEncoding按从最外到最里的顺序列出传输编码，空切片表示"identity"编码。
    // 本字段一般会被忽略。当发送或接受请求时，会自动添加或移除"chunked"传输编码。
    TransferEncoding []string
    // Close在服务端指定是否在回复请求后关闭连接，在客户端指定是否在发送请求后关闭连接。
    Close bool
    // 在服务端，Host指定URL会在其上寻找资源的主机。
    // 根据RFC 2616，该值可以是Host头的值，或者URL自身提供的主机名。
    // Host的格式可以是"host:port"。
    //
    // 在客户端，请求的Host字段（可选地）用来重写请求的Host头。
    // 如过该字段为""，Request.Write方法会使用URL字段的Host。
    Host string
    // Form是解析好的表单数据，包括URL字段的query参数和POST或PUT的表单数据。
    // 本字段只有在调用ParseForm后才有效。在客户端，会忽略请求中的本字段而使用Body替代。
    Form url.Values
    // PostForm是解析好的POST或PUT的表单数据。
    // 本字段只有在调用ParseForm后才有效。在客户端，会忽略请求中的本字段而使用Body替代。
    PostForm url.Values
    // MultipartForm是解析好的多部件表单，包括上传的文件。
    // 本字段只有在调用ParseMultipartForm后才有效。
    // 在客户端，会忽略请求中的本字段而使用Body替代。
    MultipartForm *multipart.Form
    // Trailer指定了会在请求主体之后发送的额外的头域。
    //
```

```
    // 在服务端，Trailer字段必须初始化为只有trailer键，所有键都对应nil值。
    // （客户端会声明哪些trailer会发送）
    // 在处理器从Body读取时，不能使用本字段。
    // 在从Body的读取返回EOF后，Trailer字段会被更新完毕并包含非nil的值。
    // （如果客户端发送了这些键值对），此时才可以访问本字段。
    //
    // 在客户端，Trail必须初始化为一个包含将要发送的键值对的映射。（值可以是nil或其终值）
    // ContentLength字段必须是0或-1，以启用"chunked"传输编码发送请求。
    // 在开始发送请求后，Trailer可以在读取请求主体期间被修改，
    // 一旦请求主体返回EOF，调用者就不可再修改Trailer。
    //
    // 很少有HTTP客户端、服务端或代理支持HTTP trailer。
    Trailer Header
    // RemoteAddr允许HTTP服务器和其他软件记录该请求的来源地址，一般用于日志。
    // 本字段不是ReadRequest函数填写的，也没有定义格式。
    // 本包的HTTP服务器会在调用处理器之前设置RemoteAddr为"IP:port"格式的地址。
    // 客户端会忽略请求中的RemoteAddr字段。
    RemoteAddr string
    // RequestURI是被客户端发送到服务端的请求的请求行中未修改的请求URI
    // （参见RFC 2616, Section 5.1）
    // 一般应使用URI字段，在客户端设置请求的本字段会导致错误。
    RequestURI string
    // TLS字段允许HTTP服务器和其他软件记录接收到该请求的TLS连接的信息
    // 本字段不是ReadRequest函数填写的。
    // 对启用了TLS的连接，本包的HTTP服务器会在调用处理器之前设置TLS字段，否则将设TLS为nil。
    // 客户端会忽略请求中的TLS字段。
    TLS *tls.ConnectionState
}
```

Request类型代表一个服务端接受到的或者客户端发送出去的HTTP请求。

Request各字段的意义和用途在服务端和客户端是不同的。除了字段本身上方文档，还可参见Request.Write方法和RoundTripper接口的文档。

5.1 获取请求 URL

Request 结构中的 URL 字段用于表示请求行中包含的 URL，改字段是一个指向 url.URL 结构的指针，让我们来看一下 URL 结构

type URL

```
type URL struct {
    Scheme   string
    Opaque   string    // 编码后的不透明数据
    User     *Userinfo // 用户名和密码信息
    Host     string    // host或host:port
    Path     string
    RawQuery string // 编码后的查询字符串，没有'?'
    Fragment string // 引用的片段（文档位置），没有'#'
}
```

URL类型代表一个解析后的URL（或者说，一个URL参照）。URL基本格式如下：

```
scheme://[userinfo@]host/path[?query][#fragment]
```

scheme后不是冒号加双斜线的URL被解释为如下格式：

```
scheme:opaque[?query][#fragment]
```

注意路径字段是以解码后的格式保存的，如/%47%6f%2f会变成/Go/。这导致我们无法确定Path字段中的斜线是来自原始URL还是解码前的%2f。除非一个客户端必须使用其他程序/函数来解析原始URL或者重构原始URL，这个区别并不重要。此时，HTTP服务端可以查询req.RequestURI，而HTTP客户端可以使用URL{Host: "example.com", Opaque: "//example.com/Go%2f"}代替{Host: "example.com", Path: "/Go/"}。

1) Path 字段、

- 获取请求的 URL
- 例如：http://localhost:8080/hello?username=admin&password=123456
  i. 通过 r.URL.Path 只能得到 **/hello**

2) RawQuery 字段

- 获取请求的 URL 后面?后面的查询字符串
- 例如：http://localhost:8080/hello?username=admin&password=123456
  i. 通过 r.URL.RawQuery 得到的是 username=admin&password=123456

5.2 获取请求头中的信息

通过 Request 结果中的 Header 字段用来获取请求头中的所有信息，Header 字段的类型是 Header 类型，而 Header 类型是一个 map[string][]string，string 类型的 key，string 切片类型的值。下面是 Header 类型及它的方法：

type Header

```
type Header map[string][]string
```

Header代表HTTP头域的键值对。

func (Header) Get

```
func (h Header) Get(key string) string
```

Get返回键对应的第一个值，如果键不存在会返回""。如要获取该键对应的值切片，请直接用规范格式的键访问map。

func (Header) Set

```
func (h Header) Set(key, value string)
```

Set添加键值对到h，如键已存在则会用只有新值一个元素的切片取代旧值切片。

func (Header) Add

```
func (h Header) Add(key, value string)
```

Add添加键值对到h，如键已存在则会将新的值附加到旧值切片后面。

func (Header) Del

```
func (h Header) Del(key string)
```

Del删除键值对。

func (Header) Write

```
func (h Header) Write(w io.Writer) error
```

Write以有线格式将头域写入w。

1) 获取请求头中的所有信息

- **r.Header**
- 得到的结果如下：

> **map**[User-Agent:[Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/67.0.3396.62 Safari/537.36] Accept:[text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8] Accept-Encoding:[gzip, deflate, br] Accept-Language:[zh-CN,zh;q=0.9,en-US;q=0.8,en;q=0.7] Connection:[keep-alive] Upgrade-Insecure-Requests:[1]]

2) 获取请求头中的某个具体属性的值，如获取 Accept-Encoding 的值

- 方式一：**r.Header[“Accept-Encoding”]**
  i. 得到的是一个字符串切片

ii. 结果

```
[gzip, deflate, br]
```

- 方式二：**r.Header.Get(“Accept-Encoding”)**

  i. 得到的是字符串形式的值，多个值使用逗号分隔

  ii. 结果

```
gzip, deflate, br
```

5.3 获取请求体中的信息

请求和响应的主体都是有 Request 结构中的 Body 字段表示，这个字段的类型是 io.ReadCloser 接口，该接口包含了 Reader 接口和 Closer 接口，Reader 接口拥有 Read 方法，Closer 接口拥有 Close 方法

type ReadCloser

```
type ReadCloser interface {
    Reader
    Closer
}
```

ReadCloser接口聚合了基本的读取和关闭操作。

type Reader

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

Reader接口用于包装基本的读取方法。

Read方法读取len(p)字节数据写入p。它返回写入的字节数和遇到的任何错误。即使Read方法返回值n < len(p)，本方法在被调用时仍可能使用p的全部长度作为暂存空间。如果有部分可用数据，但不够len(p)字节，Read按惯例会返回可以读取到的数据，而不是等待更多数据。

当Read在读取n > 0个字节后遭遇错误或者到达文件结尾时，会返回读取的字节数。它可能会在该次调用返回一个非nil的错误，或者在下一次调用时返回0和该错误。一个常见的例子，Reader接口会在输入流的结尾返回非0的字节数，返回值err == EOF或err == nil。但不管怎样，下一次Read调用必然返回(0, EOF)。调用者应该总是先处理读取的n > 0字节再处理错误值。这么做可以正确的处理发生在读取部分数据后的I/O错误，也能正确处理EOF事件。

如果Read的某个实现返回0字节数和nil错误值，表示被阻碍；调用者应该将这种情况视为未进行操作。

type Closer

```
type Closer interface {
    Close() error
}
```

Closer接口用于包装基本的关闭方法。

在第一次调用之后再次被调用时，Close方法的的行为是未定义的。某些实现可能会说明他们自己的行为。

1) 由于 GET 请求没有请求体，所以我们需要在 HTML 页面中创建一个 form 表单，通过指定 method=" post" 来发送一个 POST 请求

a) 表单

```
<form action="http://localhost:8080/getBody" method="POST">
        用 户 名 : <input type="text" name="username" value="hanzong"><br/>
        密 码 : <input type="password" name="password" value="666666"><br/>
        <input type="submit">
    </form>
```

b) 服务器处理请求的代码

```
package main

import (
    "fmt"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    //获取内容的长度
    length := r.ContentLength
    //创建一个字节切片
    body := make([]byte, length)
    //读取请求体
    r.Body.Read(body)
    fmt.Fprintln(w, "请求体中的内容是：", string(body))
}
```

```
func main() {
    http.HandleFunc("/getBody", handler)

    http.ListenAndServe(":8080", nil)

}
```

c) 在浏览器上显示的结果

```
请求体中的内容是： username=hanzong&password=666666
```

5.4 获取请求参数

下面我们就通过 net/http 库中的 Request 结构的字段以及方法获取**请求 URL 后面**的请求参数以及 **form 表单**中提交的请求参数

5.4.1 Form 字段

1) 类型是 **url.Values** 类型，Form 是解析好的表单数据，包括 URL 字段的 query 参数和 POST 或 PUT 的表单数据。

type Values

```
type Values map[string][]string
```

Values将建映射到值的列表。它一般用于查询的参数和表单的属性。不同于http.Header这个字典类型，Values的键是大小写敏感的。

2) **Form 字段只有在调用 Request 的 ParseForm 方法后才有效。在客户端，会忽略请求中的本字段而使用 Body 替代**

func (*Request) ParseForm

```
func (r *Request) ParseForm() error
```

ParseForm解析URL中的查询字符串，并将解析结果更新到r.Form字段。

对于POST或PUT请求，ParseForm还会将body当作表单解析，并将结果既更新到r.PostForm也更新到r.Form。解析结果中，POST或PUT请求主体要优先于URL查询字符串（同名变量，主体的值在查询字符串的值前面）。

如果请求的主体的大小没有被MaxBytesReader函数设定限制，其大小默认限制为开头10MB。

ParseMultipartForm会自动调用ParseForm。重复调用本方法是无意义的。

3) 获取 5.3 中的表单中提交的请求参数（username 和 password）

- 代码

```
func handler(w http.ResponseWriter, r *http.Request) {
    //解析表单
    r.ParseForm()
    //获取请求参数
    fmt.Fprintln(w, "请求参数为：", r.Form)
}
```

注意：在执行 r.Form 之前一定要调用 ParseForm 方法

- 结果

```
请求参数为： map[password:[666666] username:[hanzong]]
```

d) 如果对 form 表单做一些修改，在 action 属性的 URL 后面也添加相同的请求参数，如下：

```
    <form action="http://localhost:8080/getBody?username=admin&pwd=123456" method="POST">
        用户名：<input type="text" name="username" value="hanzong"><br/>
        密码：<input type="password" name="password" value="666666"><br/>
        <input type="submit">
    </form>
```

- 则执行结果如下：

```
请求参数为：map[username:[hanzong admin] password:[666666] pwd:[123456]]
```

- 我们发现：表单中的请求参数 username 和 URL 中的请求参数 username 都获取到了，而且**表单中的请求参数的值排在 URL 请求参数值的前面**
- 如果此时我们只想获取表单中的请求参数该怎么办呢？那就需要使用 Request 结构中的 PostForm 字段

5.4.2 PostForm 字段

1) 类型也是 **url.Values** 类型，用来获取**表单中的请求参数**

- 将 r.Form 改为 r.PostForm 之后的代码

```
func handler(w http.ResponseWriter, r *http.Request) {
    //解析表单
    r.ParseForm()
    //获取请求参数
    fmt.Fprintln(w, "请求参数为：", r.PostForm)
}
```

- 结果

```
请求参数为： map[username:[hanzong] password:[666666]]
```

2) 但是 PostForm 字段只支持 application/x-www-form-urlencoded 编码，如果 form 表单的 enctype 属性值为 multipart/form-data，那么使用 PostForm 字段无法获取表单中的数据，此时需要使用 MultipartForm 字段

- **说明**：form 表单的 enctype 属性的默认值是 application/x-www-form-urlencoded 编码，实现上传文件时需要讲该属性的值设置为 multipart/form-data 编码格式

5.4.3 FormValue 方法和 PostFormValue 方法

1) FormValue 方法

a) 可以通过 **FormValue** 方法快速地获取某一个请求参数，该方法调用之前会自动调用 ParseMultipartForm 和 ParseForm 方法对表单进行解析

func (*Request) FormValue

```
func (r *Request) FormValue(key string) string
```

FormValue返回key为键查询r.Form字段得到结果[]string切片的第一个值。POST和PUT主体中的同名参数优先于URL查询字符串。如果必要，本函数会隐式调用ParseMultipartForm和ParseForm。

- 代码

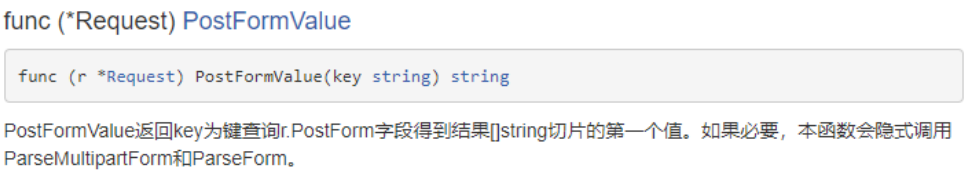
```
func handler(w http.ResponseWriter, r *http.Request) {
    //获取请求参数
    fmt.Fprintln(w, "请求参数 username 的值为 :", r.FormValue("username"))
}
```

- 结果

```
请求参数 username 的值为： hanzong
```

2) **PostFormValue 方法**

a) 可以通过 **PostFormValue** 方法快速地获取表单中的某一个请求参数，该方法调用之前会自动调用 ParseMultipartForm 和 ParseForm 方法对表单进行解析



- 代码

```
func handler(w http.ResponseWriter, r *http.Request) {
    //获取请求参数
    fmt.Fprintln(w, "请求参数 username 的值为 :", r.PostFormValue("username"))
}
```

- 结果

```
请求参数 username 的值为： hanzong
```

5.4.4 MultipartForm 字段

为了取得 multipart/form-data 编码的表单数据，我们需要用到 Request 结构的 ParseMultipartForm 方法和 MultipartForm 字段，我们通常上传文件时会将 form 表单的 enctype 属性值设置为 multipart/form-data

func (*Request) ParseMultipartForm

```
func (r *Request) ParseMultipartForm(maxMemory int64) error
```

ParseMultipartForm将请求的主体作为multipart/form-data解析。请求的整个主体都会被解析，得到的文件记录最多maxMemery字节保存在内存，其余部分保存在硬盘的temp文件里。如果必要，ParseMultipartForm会自行调用ParseForm。重复调用本方法是无意义的。

1) 表单

```
<form action="http://localhost:8080/upload" method="POST" enctype="multipart/form-data">
    用户名：<input type="text" name="username" value="hanzong"><br/>
    文件：<input type="file" name="photo" /><br/>
    <input type="submit">
</form>
```

2) 后台处理请求代码

```
package main

import (
    "fmt"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    //解析表单
    r.ParseMultipartForm(1024)
```

```
    //打印表单数据
    fmt.Fprintln(w, r.MultipartForm)
}

func main() {

    http.HandleFunc("/upload", handler)

    http.ListenAndServe(":8080", nil)

}
```

3) 浏览器显示结果

```
&{map[username:[hanzong]] map[photo:[0xc042126000]]}
```

- 结果中有两个映射，第一个映射映射的是用户名；第二个映射的值是一个地址
- MuiltipartForm 字段的类型为 *multipart.Form，multipart 包下 Form 结构的指针类型

type Form

```
type Form struct {
    Value map[string][]string
    File  map[string][]*FileHeader
}
```

Form是一个解析过的multipart表格。它的File参数部分保存在内存或者硬盘上，可以使用*FileHeader类型属性值的Open方法访问。它的Value 参数部分保存为字符串，两者都以属性名为键。

type FileHeader

```
type FileHeader struct {
    Filename string
    Header   textproto.MIMEHeader
    // 内含隐藏或非导出字段
}
```

FileHeader描述一个multipart请求的（一个）文件记录的信息。

func (*FileHeader) Open

```
func (fh *FileHeader) Open() (File, error)
```

Open方法打开并返回其关联的文件。

- 打开上传的文件

```
func handler(w http.ResponseWriter, r *http.Request) {
    //解析表单
    r.ParseMultipartForm(1024)

    fileHeader := r.MultipartForm.File["photo"][0]

    file, err := fileHeader.Open()

    if err == nil {
        data, err := ioutil.ReadAll(file)
        if err == nil {
            fmt.Fprintln(w, string(data))
        }
    }

}
```

4) FormFile 方法

- net/http 提供的 FormFile 方法可以快速的获取被上传的文件，但是只能处理上传一个文件的情况。

func (*Request) FormFile

```
func (r *Request) FormFile(key string) (multipart.File, *multipart.FileHeader, error)
```

FormFile返回以key为键查询r.MultipartForm字段得到结果中的第一个文件和它的信息。如果必要，本函数会隐式调用ParseMultipartForm和ParseForm。查询失败会返回ErrMissingFile错误。

- 代码

```
func handler(w http.ResponseWriter, r *http.Request) {

    file, _, err := r.FormFile("photo")

    if err == nil {
        data, err := ioutil.ReadAll(file)
```

```
            if err == nil {
                fmt.Fprintln(w, string(data))
            }
        }

}
```

5.5 给客户端响应

前面我们一直说的是如何使用处理器中的 *http.Request 处理用户的请求，下面我们来说一下如何使用 http.ResponseWriter 来给用户响应

type ResponseWriter

```
type ResponseWriter interface {
    // Header返回一个Header类型值，该值会被WriteHeader方法发送。
    // 在调用WriteHeader或Write方法后再改变该对象是没有意义的。
    Header() Header
    // WriteHeader该方法发送HTTP回复的头域和状态码。
    // 如果没有被显式调用，第一次调用Write时会触发隐式调用WriteHeader(http.StatusOK)
    // WriterHeader的显式调用主要用于发送错误码。
    WriteHeader(int)
    // Write向连接中写入作为HTTP的一部分回复的数据。
    // 如果被调用时还未调用WriteHeader，本方法会先调用WriteHeader(http.StatusOK)
    // 如果Header中没有"Content-Type"键，
    // 本方法会使用包函数DetectContentType检查数据的前512字节，将返回值作为该键的值。
    Write([]byte) (int, error)
}
```

ResponseWriter接口被HTTP处理器用于构造HTTP回复。

1) 给客户端响应一个字符串

- 处理器中的代码

```
func handler(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("你的请求我已经收到"))
}
```

- 浏览器中的结果

```
你的请求我已经收到
```

- 响应报文中的内容

```
HTTP/1.1 200 OK
```

```
Date: Fri, 10 Aug 2018 01:09:27 GMT
Content-Length: 27
Content-Type: text/plain; charset=utf-8
```

2) 给客户端响应一个 HTML 页面

1) 处理器中的代码

```
func handler(w http.ResponseWriter, r *http.Request) {
    html := `<html>
    <head>
        <title>测试响应内容为网页</title>
        <meta charset="utf-8"/>
    </head>
    <body>
        我是以网页的形式响应过来的！
    </body>
</html>`
    w.Write([]byte(html))
}
```

2) 浏览器中的结果

```
我是以网页的形式响应过来的！
```

- 通过在浏览器中右键→查看网页代码发现确实是一个 html 页面

3) 响应报文中的内容

```
HTTP/1.1 200 OK
Date: Fri, 10 Aug 2018 01:26:58 GMT
Content-Length: 194
Content-Type: text/html; charset=utf-8
```

3) 给客户端响应 JSON 格式的数据

1) 处理器端代码

```
func handler(w http.ResponseWriter, r *http.Request) {
    //设置响应头中内容的类型
    w.Header().Set("Content-Type", "application/json")
    user := User{
        ID:       1,
        Username: "admin",
        Password: "123456",
    }
    //将 user 转换为 json 格式
    json, _ := json.Marshal(user)
    w.Write(json)
}
```

2) 浏览器中的结果

```
{"ID":1,"Username":"admin","Password":"123456"}
```

3) 响应报文中的内容

```
HTTP/1.1 200 OK
Content-Type: application/json
Date: Fri, 10 Aug 2018 01:58:02 GMT
Content-Length: 47
```

4) 让客户端重定向

1) 处理器端代码

```
func handler(w http.ResponseWriter, r *http.Request) {
    //以下操作必须要在 WriteHeader 之前进行
    w.Header().Set("Location", "https:www.baidu.com")
    w.WriteHeader(302)
```

```
}
```

2) 响应报文中的内容

```
HTTP/1.1 302 Found
Location: https:www.baidu.com
Date: Fri, 10 Aug 2018 01:45:04 GMT
Content-Length: 0
Content-Type: text/plain; charset=utf-8
```

第 6 章：模板引擎

Go 为我们提供了 **text/template** 库和 **html/template** 库这两个模板引擎，模板引擎通过将数据和模板组合在一起生成最终的 HTML，而处理器负责调用模板引擎并将引擎生成的 HTMl 返回给客户端。

Go 的模板都是文本文档（其中 Web 应用的模板通常都是 HTML），它们都嵌入了一些称为**动作**的指令。从模板引擎的角度来说，模板就是嵌入了动作的文本（这些文本通常包含在模板文件里面），而模板引擎则通过分析并执行这些文本来生成出另外一些文本。

6.1 HelloWorld

使用 Go 的 Web 模板引擎需要以下两个步骤:

(1) 对文本格式的模板源进行语法分析，创建一个经过语法分析的模板结构，其中模板源既可以是一个字符串，也可以是模板文件中包含的内容。

(2)执行经过语法分析的模板，将 ResponseWriter 和模板所需的动态数据传递给模板引擎，被调用的模板引擎会把经过语法分析的模板和传入的数据结合起来，生成出最终的 HTML，并将这些 HTML 传递给 ResponseWriter。

下面就让我们写一个简单的 HelloWorld

1) 创建模板文件 hello.html

```
<html>
    <head>
        <title>模板文件</title>
        <meta charset="utf-8"/>
    </head>
    <body>
        //嵌入动作
        {{.}}
    </body>
</html>
```

2) 在处理器中触发模板引擎

```
func handler(w http.ResponseWriter, r *http.Request) {
    //解析模板文件
    t, _ := template.ParseFiles("hello.html")
    //执行模板
    t.Execute(w, "Hello World!")
}
```

3) 浏览器中的结果

```
Hello World!
```

6.2 解析模板

1) ParseFiles 函数

func ParseFiles

```
func ParseFiles(filenames ...string) (*Template, error)
```

ParseFiles函数创建一个模板并解析filenames指定的文件里的模板定义。返回的模板的名字是第一个文件的文件名（不含扩展名），内容为解析后的第一个文件的内容。至少要提供一个文件。如果发生错误，会停止解析并返回nil。

- 当我们调用 ParseFiles 函数解析模板文件时，Go 会创建一个新的模板，并**将给定的模板文件的名字作为新模板的名字**，如果该函数中传入了多个文件名，那么也只会返回一个模板，**而且以第一个文件的文件名作为模板的名字**，至于其他文件对应的模板则会被放到一个 **map** 中。让我们再来看一下 HelloWorld 中的代码：

```
t, _ := template.ParseFiles("hello.html")
```

- 以上代码相当于调用 New 函数创建一个新模板，然后再调用 template 的 ParseFiles 方法：

```
t := template.New("hello.html")
t, _ = t.ParseFiles("hello.html")
```

func New

```
func New(name string) *Template
```

创建一个名为name的模板。

func (*Template) ParseFiles

```
func (t *Template) ParseFiles(filenames ...string) (*Template, error)
```

ParseGlob方法解析filenames指定的文件里的模板定义并将解析结果与t关联。如果发生错误，会停止解析并返回nil，否则返回(t, nil)。至少要提供一个文件。

- 我们在解析模板时都没有对错误进行处理，Go 提供了一个 Must 函数专门用来处理这个错误。Must 函数可以包裹起一个函数，被包裹的函数会返回一个指向模板的指针和一个错误，如果错误不是 nil，那么 Must 函数将产生一个 panic。

func Must

```
func Must(t *Template, err error) *Template
```

Must函数用于包装返回(*Template, error)的函数/方法调用，它会在err非nil时panic，一般用于变量初始化：

```
var t = template.Must(template.New("name").Parse("html"))
```

- 实验 Must 函数之后的代码

```
t := template.Must(template.ParseFiles("hello.html"))
```

2) ParseGlob 函数

func ParseGlob

```
func ParseGlob(pattern string) (*Template, error)
```

ParseGlob创建一个模板并解析匹配pattern的文件（参见glob规则）里的模板定义。返回的模板的名字是第一个匹配的文件的文件名（不含扩展名），内容为解析后的第一个文件的内容。至少要存在一个匹配的文件。如果发生错误，会停止解析并返回nil。ParseGlob等价于使用匹配pattern的文件的列表为参数调用ParseFiles。

- 通过该函数可以通过指定一个规则一次性传入多个模板文件，如：

```
t, _ := template.ParseGlob("*.html")
```

6.3 执行模板

1) 通过 Execute 方法

func (*Template) Execute

```
func (t *Template) Execute(wr io.Writer, data interface{}) error
```

Execute方法将解析好的模板应用到data上，并将输出写入wr。如果执行时出现错误，会停止执行，但有可能已经写入wr部分数据。模板可以安全的并发执行。

- 如果只有一个模板文件，调用这个方法总是可行的；但是如果有多个模板文件，调用这个方法只能得到第一个模板

2) 通过 ExecuteTemplate 方法

func (*Template) ExecuteTemplate

```
func (t *Template) ExecuteTemplate(wr io.Writer, name string, data interface{}) error
```

ExecuteTemplate方法类似Execute，但是使用名为name的t关联的模板产生输出。

- 例如：

```
t, _ := template.ParseFiles("hello.html", "hello2.html")
```

- 变量 t 就是一个包含了两个模板的模板集合，第一个模板的名字是 hello.html,第二个模板的名字是 hello2.html,如果直接调用 Execute 方法，则只有模板 hello.html 会被执行，如何想要执行模板 hello2.html，则需要调用 ExecuteTemplate 方法

```
t.ExecuteTemplate(w, "hello2.html", "我要在 hello2.html 中显示")
```

6.4 动作

Go 模板的动作就是一些嵌入到模板里面的命令，这些命令在模板中需要放到两个大括号里**{{ 动作 }}**，之前我们已经用过一个**很重要的动作**：点（.），它代表了传递给模板的数据。下面我们再介绍几个常用的动作，如果还想了解其他类型的动作，可以参考 text/template 库的文档。

6.4.1 条件动作

- 格式一：

```
{{ if arg}}
   要显示的内容
{{ end }}
```

- 格式二：

```
{{ if arg}}
要显示的内容
{{else}}
 当 if 条件不满足时要显示的内容
{{ end }}
```

  - 其中的 arg 是传递给条件动作的参数，该值可以是一个字符串常量、一个变量、一个返回单个值的函数获取方法等。

- 例如：
  - 模板文件

```
<html>
    <head>
        <title>模板文件</title>
        <meta charset="utf-8"/>
    </head>
    <body>
        <!-- 嵌入动作 -->
```

```
        {{if .}}
        你已经成年了！
        {{else}}
        你还未成年
        {{end}}
    </body>
</html>
```

- 处理器端代码

```
func handler(w http.ResponseWriter, r *http.Request) {
    //解析模板文件
    t := template.Must(template.ParseFiles("hello.html"))
    //声明一个变量
    age := 16
    //执行模板
    t.Execute(w, age > 18)
}
```

- 浏览器中的结果

```
你还未成年
```

6.4.2 迭代动作

迭代动作可以对数组、切片、映射或者通道进行迭代。

- 格式一：

```
{{range . }}
  遍历到的元素是 {{ . }}
{{ end }}
```

- 格式二：

```
{{range . }}
 遍历到的元素是 {{ . }}
{{ else }}
 没有任何元素
{{ end }}
```

- **range 后面的点代表被遍历的元素 ;要显示的内容里面的点代表遍历到的元素**

- 例如：
  - 模板文件

```
<html>
    <head>
        <title>模板文件</title>
        <meta charset="utf-8"/>
    </head>
    <body>
        <!-- 嵌入动作 -->
        {{range .}}
            <a href="#">{{.}}</a>
        {{else}}
            没有遍历到任何内容
        {{end}}
    </body>
</html>
```

  - 处理器端代码

```
func handler(w http.ResponseWriter, r *http.Request) {
    //解析模板文件
    t := template.Must(template.ParseFiles("hello.html"))
    //声明一个字符串切片
```

```
    stars := []string{"马蓉", "李小璐", "白百何"}
    //执行模板
    t.Execute(w, stars)
}
```

- 浏览器中的结果

```
马蓉 李小璐 白百何
```

- 如果迭代之后是一个个的结构体，获取结构体中的字段值使用 **.字段名** 方式获取

```
{{range . }}
  获取结构体的 Name 字段名 {{ .Name }}
{{ end }}
```

- 迭代 Map 时可以设置变量，变量以$开头：

```
{{ range $k , $v := . }}
  键是 {{ $k }}，值是 {{ $v }}
{{ end }}
```

- 迭代管道

```
{{ c1 | c2 | c3 }}
```

- c1、c2 和 c3 可以是参数或者函数。管道允许用户将一个参数的输出传递给下一个参数，各个参数之间使用 | 分割。

6.4.3 设置动作

设置动作允许在指定的范围内对点（.）设置值。

- 格式一：

```
{{ with arg }}
  为传过来的数据设置的新值是{{ . }}
{{ end }}
```

- 格式二：

```
{{ with arg }}
  为传过来的数据设置的新值是{{ . }}
{{ else }}
  传过来的数据仍然是{{ . }}
{{ end }}
```

- 例如：
  - 模板文件

```
<html>
    <head>
        <title>模板文件</title>
        <meta charset="utf-8"/>
    </head>
    <body>
        <!-- 嵌入动作 -->
        <div>得到的数据是：{{.}}</div>
        {{with "太子"}}
        <div>替换之后的数据是：{{.}}</div>
        {{end}}
        <hr/>
        {{with ""}}
        <div>看一下现在的数据是：{{.}}</div>
        {{else}}
        <div>数据没有被替换，还是：{{.}}</div>
        {{end}}
```

```
    </body>
</html>
```

- 处理器端代码

```
func handler(w http.ResponseWriter, r *http.Request) {
    //解析模板文件
    t := template.Must(template.ParseFiles("hello.html"))
    //执行模板
    t.Execute(w, "狸猫")
}
```

- 浏览器中的结果

```
得到的数据是：狸猫
替换之后的数据是：太子
---
数据没有被替换，还是：狸猫
```

6.4.4 包含动作

包含动作允许用户在**一个模板里面包含另一个模板**，从而构建出嵌套的模板。

- 格式一：{{ template “name” }}
  - name 为被包含的模板的名字
- 格式二：{{ template “name” arg }}
  - arg 是用户想要传递给被嵌套模板的数据
- 例如：
  - 模板文件
    - hello.html

```
<html>
    <head>
        <title>模板文件</title>
        <meta charset="utf-8"/>
    </head>
    <body>
        <!-- 嵌入动作 -->
        <div>从后台得到的数据是：{{.}}</div>
        <!-- 包含 hello2.html 模板 -->
        {{ template "hello2.html"}}
        <div>hello.html 文件内容结束</div>
        <hr/>
        <div>将 hello.html 模板文件中的数据传递给 hello2.html
模板文件</div>
        {{ template "hello2.html" . }}
    </body>
</html>
```

- hello2.html

```
<html>
    <head>
        <title>hello2 模板文件</title>
        <meta charset="utf-8"/>
    </head>
    <body>
        <!-- 嵌入动作 -->
        <div>hello2.html 模板文件中的数据是：{{.}}</div>
    </body>
</html>
```

- 处理器端代码

```
func handler(w http.ResponseWriter, r *http.Request) {
    //解析模板文件
    t := template.Must(template.ParseFiles("hello.html", "hello2.html"))
    //执行模板
    t.Execute(w, "测试包含")
}
```

  - **注意**：在解析模板文件时，当前文件以及被包含的文件都要解析

- 浏览器中的结果

```
从后台得到的数据是：测试包含
hello2.html 模板文件中的数据是：
hello.html 文件内容结束
---
将 hello.html 模板文件中的数据传递给 hello2.html 模板文件
hello2.html 模板文件中的数据是：测试包含
```

6.4.5 定义动作

当我们访问一些网站时，经常会看到好多网页中有相同的部分：比如导航栏、版权信息、联系方式等。这些相同的布局我们可以通过定义动作在模板文件中定义模板来实现。定义模板的格式是：以{{ define “layout” }}开头，以{{ end }}结尾。

1) 在一个模板文件（hello.html）中定义一个模板

```
<!-- 定义模板 -->
{{ define "model"}}
<html>
    <head>
        <title>模板文件</title>
        <meta charset="utf-8"/>
```

```
    </head>
    <body>
        {{ template "content"}}
    </body>
</html>
{{ end }}
```

2) 在一个模板文件中定义多个模板

- 模板文件（hello.html）

```
<!-- 定义模板 -->
{{ define "model"}}
<html>
    <head>
        <title>模板文件</title>
        <meta charset="utf-8"/>
    </head>
    <body>
        {{ template "content"}}
    </body>
</html>
{{ end }}

{{ define "content"}}
    <a href="#">点我有惊喜</a>
{{ end }}
```

- 处理器端代码

```
func handler(w http.ResponseWriter, r *http.Request) {
    //解析模板文件
    t := template.Must(template.ParseFiles("hello.html"))
```

```
    //执行模板
    t.ExecuteTemplate(w, "model", "")
}
```

- **注意**：需要调用 ExecuteTemplate 方法并指定模板的名字

- 浏览器中的结果

点我有惊喜

3) 在不同的模板文件中定义同名的模板

- 模板文件
  - hello.html

```
<!-- 定义模板 -->
{{ define "model"}}
<html>
    <head>
        <title>模板文件</title>
        <meta charset="utf-8"/>
    </head>
    <body>
        {{ template "content"}}
    </body>
</html>
{{ end }}
```

  - content1.html

```
<html>
    <head>
        <title>content 模板文件</title>
        <meta charset="utf-8"/>
    </head>
    <body>
```

```
        <!-- 定义 content 模板 -->
        {{ define "content" }}
            <h1>我是 content1.html 模板文件中的内容</h1>
        {{ end }}
    </body>
</html>
```

- content2.html

```
<html>
    <head>
        <title>content 模板文件</title>
        <meta charset="utf-8"/>
    </head>
    <body>
        <!-- 定义 content 模板 -->
        {{ define "content" }}
            <h1>我是 content2.html 模板文件中的内容</h1>
        {{ end }}
    </body>
</html>
```

- 处理器端代码

```
func handler(w http.ResponseWriter, r *http.Request) {
    rand.Seed(time.Now().Unix())
    var t *template.Template
    if rand.Intn(5) > 2 {
        //解析模板文件
        t = template.Must(template.ParseFiles("hello.html",
"content1.html"))
```

```
    } else {
        //解析模板文件
        t = template.Must(template.ParseFiles("hello.html", "content2.html"))
    }
    //执行模板
    t.ExecuteTemplate(w, "model", "")
}
```

- 浏览器中的结果

```
我是 content1.html 模板文件中的内容
```

6.4.6 块动作

Go 1.6 引入了一个新的块动作，这个动作允许用户定义一个模板并立即使用。相当于设置了一个默认的模板

- 格式：

```
{{ block arg }}
  如果找不到模板我就要显示了
{{ end }}
```

- 修改 6.4.5 中的模板文件 hello.html

```
<!-- 定义模板 -->
{{ define "model"}}
<html>
    <head>
        <title>模板文件</title>
        <meta charset="utf-8"/>
    </head>
```

```
    <body>
        {{ block "content" .}}
            如果找不到就显示我
        {{ end }}
    </body>
</html>
{{ end }}
```

- 稍微修改一下处理器端的代码

```
func handler(w http.ResponseWriter, r *http.Request) {
    rand.Seed(time.Now().Unix())
    var t *template.Template
    if rand.Intn(5) > 2 {
        //解析模板文件
        t = template.Must(template.ParseFiles("hello.html", "content1.html"))
    } else {
        //解析模板文件
        t = template.Must(template.ParseFiles("hello.html"))
    }
    //执行模板
    t.ExecuteTemplate(w, "model", "")
}
```

- 浏览器中的结果

```
如果找不到就显示我
```

第 7 章：会话控制

HTTP 是无状态协议，服务器不能记录浏览器的访问状态，也就是说服务器不能区分中两次请求是否由一个客户端发出。这样的设计严重阻碍的 Web 程序的设计。如：在我们进行网购时，买了一条裤子，又买了一个手机。由于 http 协议是无状态的，如果不通过其他手段，服务器是不能知道用户到底买了什么。而 Cookie 就是解决方案之一。

7.1 Cookie

7.1.1 简介

Cookie 实际上就是服务器保存在浏览器上的一段信息。浏览器有了 Cookie 之后，每次向服务器发送请求时都会同时将该信息发送给服务器，服务器收到请求后，就可以根据该信息处理请求。

type Cookie

```
type Cookie struct {
    Name       string
    Value      string
    Path       string
    Domain     string
    Expires    time.Time
    RawExpires string
    // MaxAge=0表示未设置Max-Age属性
    // MaxAge<0表示立刻删除该cookie，等价于"Max-Age: 0"
    // MaxAge>0表示存在Max-Age属性，单位是秒
    MaxAge   int
    Secure   bool
    HttpOnly bool
    Raw      string
    Unparsed []string // 未解析的“属性-值”对的原始文本
}
```

Cookie代表一个出现在HTTP回复的头域中Set-Cookie头的值里或者HTTP请求的头域中Cookie头的值里的HTTP cookie。

7.1.2 Cookie 的运行原理

1) 第一次向服务器发送请求时在服务器端创建 Cookie
2) 将在服务器端创建的 Cookie 以响应头的方式发送给浏览器

3) 以后再发送请求浏览器就会携带着该 Cookie

4) 服务器得到 Cookie 之后根据 Cookie 的信息来区分不同的用户

7.1.3 创建 Cookie 并将它发送给浏览器

1) 在服务器创建 Cookie 并将它发送给浏览器

- 服务器端代码

```
func handler(w http.ResponseWriter, r *http.Request) {
    cookie1 := http.Cookie{
        Name:     "user1",
        Value:    "admin",
        HttpOnly: true,
    }

    cookie2 := http.Cookie{
        Name:     "user2",
        Value:    "superAdmin",
        HttpOnly: true,
    }

    //将 Cookie 发送给浏览器,即添加第一个 Cookie
    w.Header().Set("Set-Cookie", cookie1.String())
    //再添加一个 Cookie
    w.Header().Add("Set-Cookie", cookie2.String())
}
```

- 浏览器响应报文中的内容

```
HTTP/1.1 200 OK
Set-Cookie: user1=admin; HttpOnly
Set-Cookie: user2=superAdmin; HttpOnly
```

```
Date: Sun, 12 Aug 2018 07:24:49 GMT
Content-Length: 0
Content-Type: text/plain; charset=utf-8
```

2) 以后每次发送请求浏览器都会携带着 Cookie

```
GET /cookie HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/67.0.3396.62 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9,en-US;q=0.8,en;q=0.7
Cookie: user1=admin; user2=superAdmin
```

3) 除了 Set 和 Add 方法之外，Go 还提供了一种更快捷的设置 Cookie 的方式，就是通过 net/http 库中的 **SetCookie 方法**

func SetCookie

```
func SetCookie(w ResponseWriter, cookie *Cookie)
```

SetCookie在w的头域中添加Set-Cookie头，该HTTP头的值为cookie。

- 将 1)中的代码进行修改

```
func handler(w http.ResponseWriter, r *http.Request) {
    cookie1 := http.Cookie{
        Name:    "user1",
        Value:   "admin",
```

```
        HttpOnly: true,
    }

    cookie2 := http.Cookie{
        Name:     "user2",
        Value:    "superAdmin",
        HttpOnly: true,
    }

    http.SetCookie(w, &cookie1)
    http.SetCookie(w, &cookie2)
}
```

7.1.4 读取 Cookie

由于我们在发送请求时 Cookie 在请求头中，所以我们可以通过 Request 结构中的 Header 字段来获取 Cookie

1) 处理器端代码

```
func handler(w http.ResponseWriter, r *http.Request) {
    //获取请求头中的 Cookie
    cookies := r.Header["Cookie"]
    fmt.Fprintln(w, cookies)
}
```

2) 浏览器中的结果

```
[user1=admin; user2=superAdmin]
```

7.1.5 设置 Cookie 的有效时间

Cookie **默认是会话级别的**, 当关闭浏览器之后 Cookie 将失效, 我们可以通过 Cookie 结构的 MaxAge 字段设置 Cookie 的有效时间

1) 处理器端代码

```
func handler(w http.ResponseWriter, r *http.Request) {


    cookie := http.Cookie{
        Name:     "user",
        Value:    "persistAdmin",
        HttpOnly: true,
        MaxAge:   60,
    }


    //将 Cookie 发送给浏览器
    w.Header().Set("Set-Cookie", cookie.String())
}
```

2) 浏览器响应报文中的内容

```
HTTP/1.1 200 OK
Set-Cookie: user=persistAdmin; Max-Age=60; HttpOnly
Date: Sun, 12 Aug 2018 07:32:57 GMT
Content-Length: 0
Content-Type: text/plain; charset=utf-8
```

7.1.6 Cookie 的用途

1) 广告推荐

2) 免登录

7.2 Session

7.2.1 简介

使用 Cookie 有一个非常大的局限，就是如果 Cookie 很多，则无形的增加了客户端与服务端的数据传输量。而且由于浏览器对 Cookie 数量的限制，注定我们不能再 Cookie 中保存过多的信息，于是 Session 出现。

Session 的作用就是**在服务器端保存一些用户的数据**，然后传递给用户一个特殊的 Cookie，这个 Cookie 对应着这个服务器中的一个 Session，通过它就可以获取到保存用户信息的 Session，进而就知道是那个用户再发送请求。

7.2.2 Session 的运行原理

1) 第一次向服务器发送请求时创建 Session，给它设置一个全球唯一的 ID（可以通过 UUID 生成）
2) 创建一个 Cookie，将 Cookie 的 Value 设置为 Session 的 ID 值，并将 Cookie 发送给浏览器
3) 以后再发送请求浏览器就会携带着该 Cookie
4) 服务器获取 Cookie 并根据它的 Value 值找到服务器中对应的 Session，也就知道了请求是那个用户发的

第 8 章：处理静态文件

对于 HTML 页面中的 css 以及 js 等静态文件，需要使用使用 net/http 包下的以下方法来处理

1) StripPrefix 函数

func StripPrefix

```
func StripPrefix(prefix string, h Handler) Handler
```

StripPrefix返回一个处理器，该处理器会将请求的URL.Path字段中给定前缀prefix去除后再交由h处理。StripPrefix会向URL.Path字段中没有给定前缀的请求回复404 page not found。

2) FileServer 函数

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:

type FileSystem

```
type FileSystem interface {
    Open(name string) (File, error)
}
```

FileSystem接口实现了对一系列命名文件的访问。文件路径的分隔符为'/'，不管主机操作系统的惯例如何。

type Dir

```
type Dir string
```

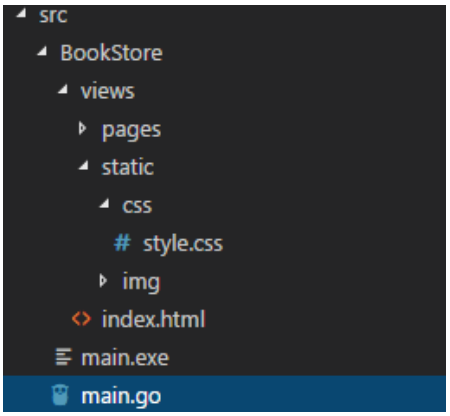
Dir使用限制到指定目录树的本地文件系统实现了http.FileSystem接口。空Dir被视为"."，即代表当前目录。

func (Dir) Open

```
func (d Dir) Open(name string) (File, error)
```

3) 例如：

a) 项目的静态文件的目录结构如下：



b) index.html 模板文件中引入的 css 样式的地址如下：

```
<link type="text/css" rel="stylesheet" href="static/css/style.css"
```

c) 对静态文件的处理

```
http.Handle("/static/", http.StripPrefix("/static/", http.FileServer(http.Dir("views/static"))))
```

- /static/会匹配 以 /static/开发的路径，当浏览器请求 index.html 页面中的 style.css 文件时，static 前缀会被替换为 views/staic，然后去 views/static/css 目录中取查找 style.css 文件