base 日常工作中所学

1 安装依赖

于 base 文件目录下，执行如下命令：

```
go init go.mod

go mod tidy
```

2 目录分布

2.1 godaily

初步接触 go 语言学习过程中所写代码，由于旧文件丢失，目前正在重写

- arrmap

拼接一个 map 中含有三对 string 类型的 map 信息，实现匹配

[map[deviceSerial:deviceSerial merchantId:merchantId terminalId:terminalId] map[deviceSerial:33333 merchantId:111111111 terminalId:2222222] map[deviceSerial:6666666 merchantId:44444444 terminalId:55555555]]

- bibao

闭包是指一个函数 A，返回一个或多个匿名函数 B，这一个或多个匿名函数 B 和函数 A 共享一个变量

闭包的延迟绑定：实际上闭包并不会在声明 var sum = func() int {...}这句话之后，就将外部环境的 变量绑定到闭包中，而是在函数返回闭包函数的时候，才绑定的

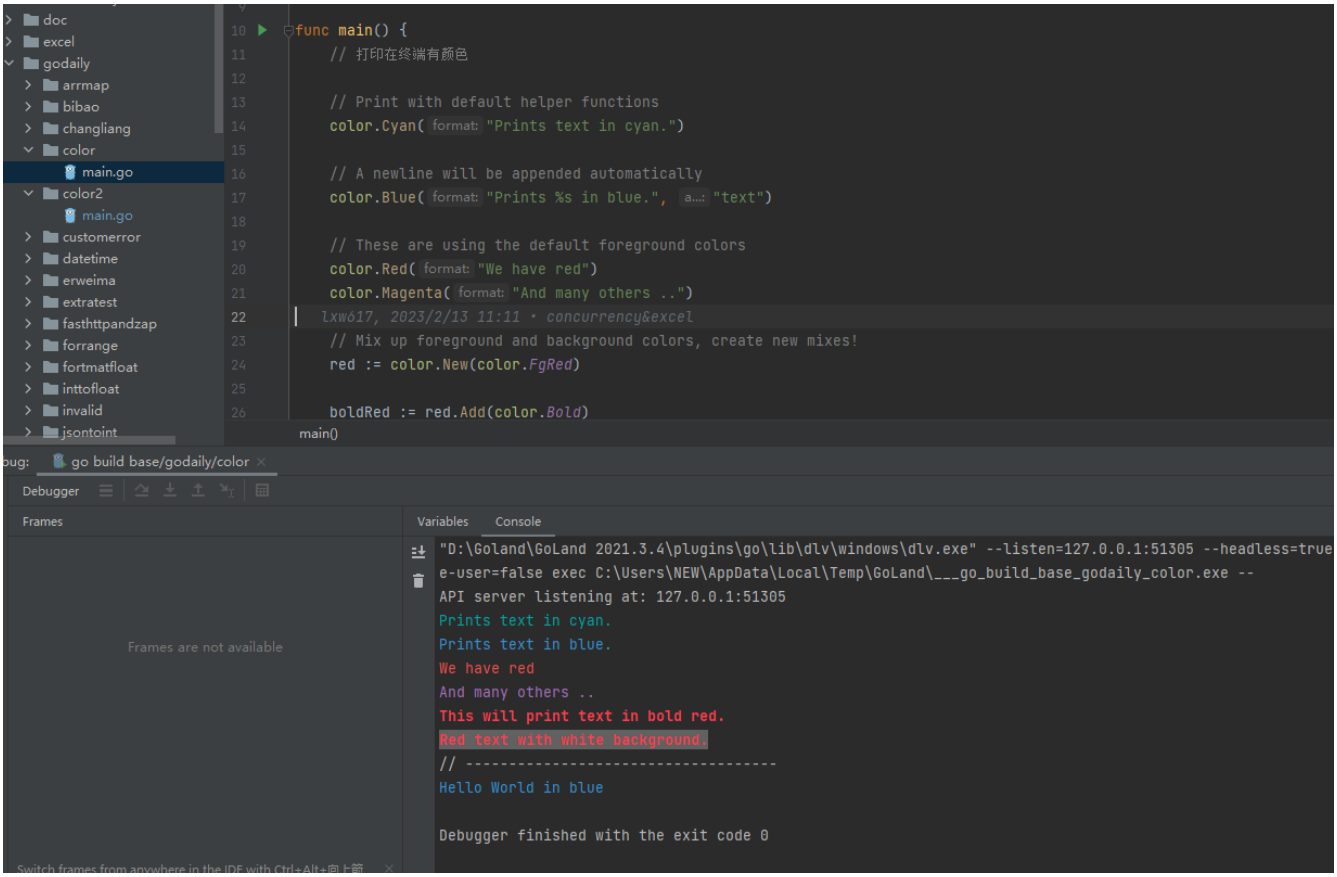
具体可参考 https://blog.csdn.net/Aphysia/article/details/127956948

- changliang

常量声明的同底类型不同类型变量可以相加，变量声明的同底类型不同类型变量不可以相加。同底指的是代码的根本类型一致

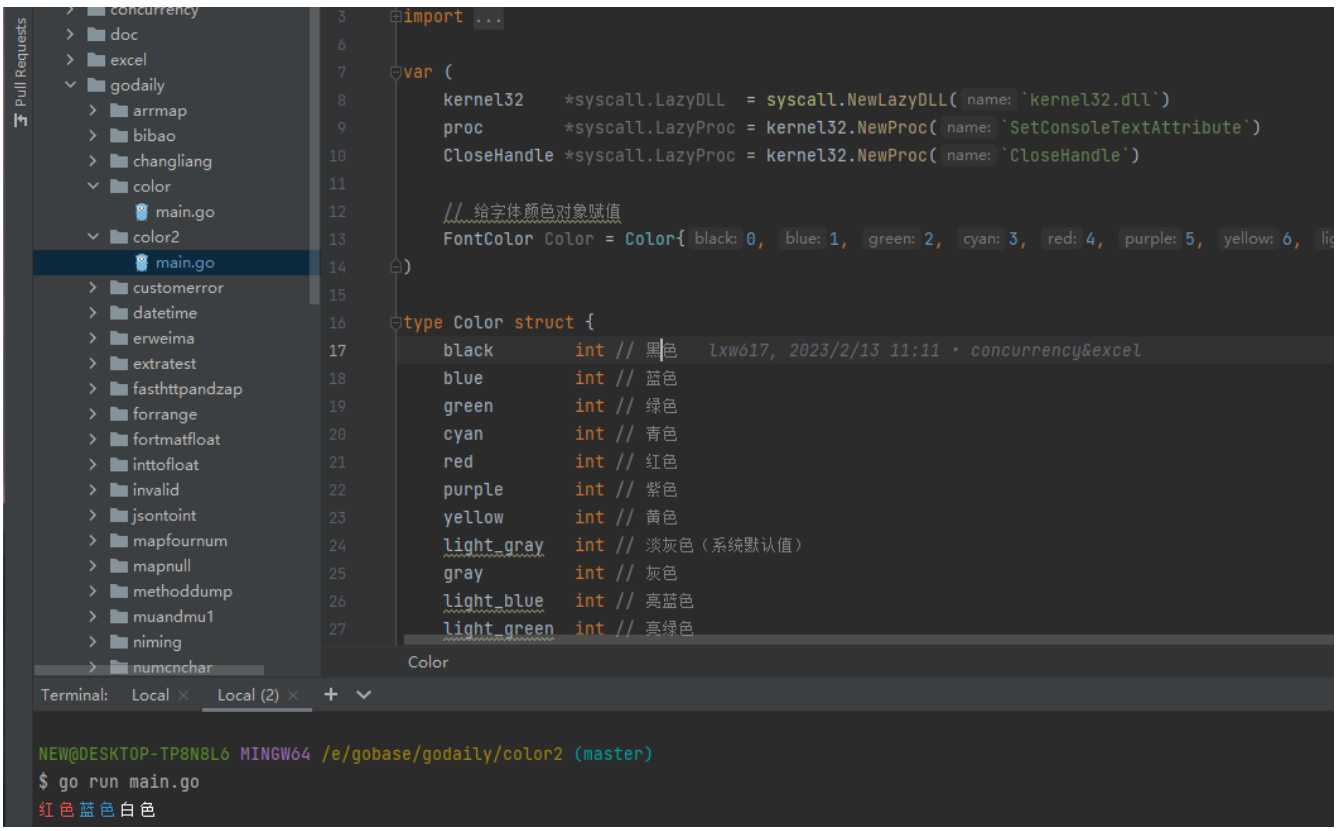
- color1

打印在终端有颜色



- color2

打印在终端有颜色



- customererror

测试自定义的错误

- datetime

获取当前时间，日期格式化

```
  // 时间的常量:在程序中可用于获取指定时间单位的时间，比如想得到100 毫秒100 * time. Millisecond
const (
        Nanosecond Duration = 1 //纳秒
        Microsecond = 1000 * Nanosecond //微秒
        Millisecond = 1000 * Microsecond //毫秒
        Second = 1000 * Millisecond //秒
        Minute = 60 * Second //分钟
        Hour = 60 * Minute //小时
)
```

- fasthttpandzap

fastHTTPHandler 和 zap

- forrange

for range 循环

- inttofloat

int 转化为 float

- invalid

utf8.Valid([]byte{}) 判断是否为 unicode 字符“界”的 utf8 编码，即是否有效

- jsontoint

json 中提取 int 类型字段内容

- jsonunmarshal

格式化 json 为指定格式

- mapnil

键值对都为 nil 的 map 不为空，即 len(map) = 1

- methoddump

输出结构体拥有的方法

- muscramble

当 time.Sleep(10 * time.Second) 时，g0 先执行；当 time.Sleep(1 * time.Second) 时，g1 先执行

- niming

全局匿名函数可使用多次，在定义匿名函数时直接调用，这种匿名函数只能使用一次

- numcnchar

获取指定字符串汉字个数

- pointer

指针与非指针值的区别

- pool

常用业务逻辑，两个 channel，一个用来放置工作项，一个用来存放处理结果。再来一个处理工作线程的方法 worker，处理具体的业务逻辑，将 channel 中用来放置工作项 jobs 中的任务取出，处理后将处理结果存放到处理结果 results 的channel 中

- poolwork

绑定结构体数据，将初始化后的 list 数据存入 channal 中，开启2个线程 for 无限循环执行 process 方法，处理 channal 数据，sync.WaitGroup 操作位于 process 方法内部

- querystruct

实现 url 中 get 方法的参数拼接 q=foo&all=true&page=2

- saletickets

售票，一个线程生产，另一个线程售卖

- slice

切片与数组区别

- stringbytestr

string 存储为 byte 和输出

- string

string 相关方法

统计字符串的长度，按字节 len(str)，含有汉字的统计使用 utf8.RuneCountInString()，golang 的编码统一为 utf-8 (ascii 的字符(字母和数字)占1个字节，汉字占用3个字节)

```
    // 统计字符串的长度,按字节len(str)
    // golang的编码统一为utf-8(ascii的字符(字母和数字)占一个字节，汉字占用3个字节)
    str := "hello北"
    fmt.Println("str len=", len(str)) // 8
    fmt.Println(utf8.RuneCountInString(str)) // 6

    str2 := "he1lo北京"
    // 字符串遍历,同时处理有中文的问题r:=[]rune(str)
    r := []rune(str2)
    for i := 0; i < len(r); i++ {
        fmt.Printf("字符=%c\n", r[i])
    }

    // 字符串转整数:n,err := strconv.Atoi("12")
    n, err := strconv.Atoi("he11o")
    if err != nil {
        fmt.Println("转换错误", err)
    } else {
        fmt.Println("转成的结果是", n)
    }
```

```
    // 整数转字符串 str = strconv.Itoa(12345)
    str = strconv.Itoa(12345)
    fmt.Printf("str=%v,str=%T\n", str, str)

    // 字符串转[]byte: var bytes=[]byte("hello go")
    bytes := []byte("hello go")
    fmt.Printf("bytes=%v\n", bytes)

    //[]byte转字符串:str = string([]byte{97, 98,99})
    str = string([]byte{97, 98, 99})
    fmt.Printf("str=%v\n", str)

    // 10进制转2，8，16进制:str = strconv.FormatInt(123，2),返回对应的字符串
    str = strconv.FormatInt(123, 2)
    fmt.Printf("123对应的二进制是=%v\n", str)
    str = strconv.FormatInt(123, 16)
    fmt.Printf("123对应的16进制是=%v\n", str)

    // 查找子串是否在指定的字符串中:strings.Contains("seafood", "foo")//true
    b := strings.Contains("seafood", "maryt")
    fmt.Printf("b=%v\n", b)

    // 统计一个字符串有几个指定的子串 :strings.Count(" ceheese", "e")//4
    num := strings.Count("ceheese", "e")
    fmt.Printf("num=%v\n", num)

    // 不区分大小写的字符串比较(=是区分字母大小写的):fmt.Println(strin
    b = strings.EqualFold("abc", "Abc")
    fmt.Printf("b=%v\n", b)           // true
    fmt.Println("结果", "abc" == "Abc") // false//区分字母大小写

    // 返回子串在字符串第一次出现的index值,如果没有返回-1://strings.Index("NLT_abc","abc")1/4
    index := strings.Index("NLT_abcabcabc", "abc") // 4
    fmt.Printf("index=%v\n", index)

    // 返回子串在字符串最后一次出现的index,
    // 如没有返回-1 : strings.LastIndex("go golang",“go")
    index = strings.LastIndex("go golang", "go") // 3
    fmt.Printf("index=%v\n", index)

    // 将指定的子串替换成另外一个子串:strings.Replace( "go go hello", "go", "go语言",n)//n可以指定
你希望替换几个,如果n=-1表示全部替换
    str2 = "go go hello"
    str = strings.Replace(str2, "go", "北京", -1)
    fmt.Printf("str=%v str2=%v\n", str, str2)

    // 按照指定的某个字符,为分割标识，将一个字符串拆分成字符串数
组://strings.Split("hello,wrold,ok",",")
    strArr := strings.Split("he1lo,wrold,ok", ",")
    for i := 0; i < len(strArr); i++ {
        fmt.Printf("str[%v]=%v\n", i, strArr[i])
        fmt.Printf("strArr=%v\n", strArr)
```

```
    }

    // 15)将字符串的字母进行大小写的转换:
    // strings.ToLower("co")l/ go strings.ToUpper("Go") //Go
    str = "goLang He1lo"
    str = strings.ToLower(str)
    str = strings.ToUpper(str)
    fmt.Printf("str=%v\n", str) // golang hello

    // 将字符串左右两边的空格去掉
    str = strings.TrimSpace(" tn a lone gopher ntrn ")
    fmt.Printf("str=%q\n", str)

    // 将字符串左右两边指定的字符去掉
    // strings. Trim("! hello! ",“I")//["hello"]//将左右两边!和”“去掉
    str = strings.Trim("! hel11o! ", "!")
    fmt.Printf("str=%q\n", str)
    // 将字符串左边指定的字符去掉 ： strings.TrimLeft("! hello! ", " !") // ["hello"] //将左边!和""去掉
    // 将字符串右边指定的字符去掉 : strings.TrimRight("! hello! ", " !") // ["hello"] //将右边!和""去掉

    // 判断字符串是否以指定的字符串开头: strings.HasPrefix("ftp://192.168.10.1", "ftp") // true
    b = strings.HasPrefix("ftp://192.168.10.1", "ftp")
    fmt.Printf("b=%v\n", b)
    // 判断字符串是否以指定的字符串结束: strings.HasSuffix("NLT_abc.jpg", "abc") //false
```

- structsize

结构体 unsafe.Sizeof() 长度

- switch1

case 为枚举常量

- switch2

case 为大于小于等表达式

- withgroup

withgroup 实现三个线程运行一个方法

- varstruct

结构体初始化

- unbufferedchannel

通道：无缓冲通道，由于缓冲区大小不足造成的延迟，追踪延迟

- zifu

%T 该变量的类型，占用字节数为 unsafe.Sizeof(变量)

十进制数形式:如:5.12 .512(必须有小数点)

科学计数法形式，以 e2 为结尾，其中2为指数，可为正数，也可为负数

byte 类型值直接输出了的对应的字符的码值。如果我们希望输出对应字符,需要使用格式化输出 %c

可以直接给某个变量赋一个数字，然后按格式化输出时 %c，会输出该数字对应的 unicode 字符

字符类型是可以进行运算的

2.2 doc

本项目 README.md pdf 版本

2.3 concurrency

并发相关代码

2.4 excel

go 操作 excel 文件相关代码，xlsx 部分功能不可实现，需结合 excelize 实现，具体参考官方文档

https://github.com/tealeg/xlsx

https://github.com/qax-os/excelize

2.5 gotest

测试 benchmark

2.6 hello_grpc

grpc 测试调用

```
PS E:\base\hello_grpc\pb> .\build.bat

E:\base\hello_grpc\pb>protoc --go_out=. --go_opt=paths=source_relative --go-grpc_out=. --
go-grpc_opt=paths=source_relative --grpc-gateway_out . --grpc-gateway_opt
paths=source_relative  ./hello/hello_grpc.proto
PS E:\base\hello_grpc\pb> go run .\client\main.go
CreateFile .\client\main.go: The system cannot find the path specified.
PS E:\base\hello_grpc\pb> cd ..
PS E:\base\hello_grpc> go run .\client\main.go
从客户端接收到的消息为好久不见
22

PS E:\base\hello_grpc> go run .\server\main.go
好久不见
12
exit status 0xc000013a
```

2.7 leetcode

力扣相关代码，编号与力扣题号对应，数据库相关刷新完毕，本人未开会员

2.8 temp

目前均由 excel 文件生成的

2.9 .gitignore

上传到 github 忽略目录及文件

2.10 tools

- adderss

测试 "github.com/pupuk/addr" 地址识别

- erweima

生成内容为 Hello World 的二维码

- formatfloat

格式化 float

- regex

测试正则匹配

- wire_easy

给背景图加字

- stringtomap

字符串转化为 map

2.11 goweb

阶段一：创建一个服务器

2.11.1 http-server

创建一个简单的 HTTP 服务器，它将呈现 Hello World！当我们从命令行浏览 http://localhost:8080 或执行 curl http://localhost:8080 时。

- `package main`：定义程序的包名。
- `import ( "fmt" "log" "net/http" )`：这是一个预处理器命令，告诉 Go 编译器包含来自 `fmt`、`log` 和 `net/http` 包的所有文件。
- `const ( CONN_HOST = "localhost" CONN_PORT = "8080" )`：我们使用 `const` 关键字在 Go 程序中声明常量。这里我们声明了两个常量，一个是以 localhost 为值的 `CONN_HOST`，另一个是以 `8080` 为值的 `CONN_PORT`。
- `func helloWorld(w http.ResponseWriter, r *http.Request) { fmt.Fprintf(w, "Hello World!") }`：这是一个 Go 函数，以 `ResponseWriter` 和 `Request` 为输入，将 `Hello World!` 写入 HTTP 响应流。
- `http.HandleFunc("/", helloWorld)`：在这里，我们使用 `net/http` 包的 `HandleFunc` 使用 `/` URL 模式注册 `helloWorld` 函数，这意味着只要我们使用模式 `/` 访问 HTTP URL，就会执行 `helloWorld`，将 `(http.ResponseWriter`、`*http.Request)` 作为参数传递给它。
- `err := http.ListenAndServe(CONN_HOST+":"+CONN_PORT, nil)`：在这里，我们调用 `http.ListenAndServe` 为 HTTP 请求提供服务，这些请求在单独的 Goroutine 中处理每个传入连接。

`ListenAndServe` 接受服务器地址和处理程序两个参数。在这里，我们将服务器地址传递为 `localhost:8080`，将处理程序传递为 `nil`，这意味着我们要求服务器使用 `DefaultServeMux` 作为处理程序。

- `if err != nil { log.Fatal("error starting http server : ", err) return}`：在这里，我们检查服务器启动是否有问题。如果有，则记录错误并使用状态代码 `1` 退出。

2.11.2 http-server-basic-authentication

添加一个 `BasicAuth` 函数并修改 `HandleFunc` 以调用它来更新我们在前面配方中创建的 HTTP 服务器。一旦服务器启动，在浏览器中访问 `http://localhost:8080` 将提示您输入用户名和密码。分别提供为 `admin`、`admin` 将呈现 Hello World！在屏幕上，对于用户名和密码的每一个其他组合，它将使您无权访问应用。并设置 `WWW-Authenticate` 和状态码 `401` 并在 HTTP 响应流上写入 `You are Unauthorized to access the application`。

要从命令行访问服务器，我们必须提供 `--user` 标志作为 `curl` 命令的一部分，如下所示：

```
$ curl --user admin:admin http://localhost:8080/
Hello World!
```

我们也可以使用 `username:password` 的 `base64` 编码令牌访问服务器，我们可以从任何网站获取，如 `https://www.base64encode.org/`，并在 `curl` 命令中将其作为授权头传递，如下所示：

```
$ curl -i -H 'Authorization:Basic YWRtaW46YWRtaW4=' http://localhost:8080/

HTTP/1.1 200 OK
Date: Sat, 12 Aug 2017 12:02:51 GMT
Content-Length: 12
Content-Type: text/plain; charset=utf-8
Hello World!
```

- `import` 函数添加了一个额外的包 `crypto/subtle`，我们将使用它来比较用户输入的凭证中的用户名和密码。
- 使用 `const` 函数，我们定义了两个额外的常量 `ADMIN_USER` 和 `ADMIN_PASSWORD`，我们将在验证用户时使用它们。
- 声明了一个 `BasicAuth()` 方法，该方法接受两个输入参数，一个是处理程序，在用户成功通过身份验证后执行，另一个是 realm，返回 `HandlerFunc`。
- 传递一个 `BasicAuth` 处理程序，而不是 `nil` 或 `DefaultServeMux` 来处理 URL 模式为 `/` 的所有传入请求。

2.11.3 http-server

GZIP 压缩意味着以 `.gzip` 格式从服务器向客户机发送响应，而不是发送普通响应。如果客户机/浏览器支持，发送压缩响应通常是一种好的做法。

通过发送压缩响应，我们节省了网络带宽和下载时间，最终加快了页面渲染速度。GZIP 压缩中发生的情况是浏览器发送一个请求头，告诉服务器它接受压缩内容（`.gzip` 和 `.deflate`），如果服务器能够以压缩形式发送响应，则发送它。如果服务器支持压缩，那么它会将 `Content-Encoding: gzip` 设置为响应头，否则它会向客户端发送一个普通响应，这显然意味着请求压缩响应只是浏览器的请求，而不是请求。我们将使用 Gorilla 的 handlers 包在这个配方中实现它。

我们将创建一个带有单个处理程序的 HTTP 服务器，它将编写 Hello World！在 HTTP 响应流上，使用 Gorilla `CompressHandler` 以 `.gzip` 格式将所有响应发送回客户端，即服务器具有内容编码响应头值 gzip 。

要使用 Gorilla 处理程序，首先我们需要使用`go get`命令安装软件包，或者手动将其复制到`$GOPATH/src`或`$GOPATH`，如下所示：

```
$ go get github.com/gorilla/handlers
```

创建一个简单的 HTTP 服务器，它将呈现 Hello World！当我们从命令行浏览 http://localhost:8080 或执行 curl http://localhost:8080 时。

- `mux := http.NewServeMux()`：分配并返回一个新的 HTTP 请求多路复用器（`ServeMux`），该多路复用器根据注册模式列表匹配每个传入请求的 URL，并调用与 URL 最接近的模式的处理程序。使用它的一个好处是，程序可以完全控制服务器使用的处理程序，尽管在`DefaultServeMux`中注册的任何处理程序都会被忽略。
- `err := http.ListenAndServe(CONN_HOST+":"+CONN_PORT, handlers.CompressHandler(mux))`：在这里，我们调用`http.ListenAndServe`为 HTTP 请求提供服务，这些请求在单独的 Goroutine 中为我们处理每个传入连接。`ListenAndServe`接受服务器地址和处理程序两个参数。在这里，我们将服务器地址传递为`localhost:8080`，处理程序传递为`CompressHandler`，它用`.gzip`处理程序包装我们的服务器，以`.gzip`格式压缩所有响应。

2.11.4 tcp-server

每当您必须构建高性能的面向系统时，编写 TCP 服务器总是优于 HTTP 服务器的最佳选择，因为 TCP 套接字没有 HTTP 那么大。Go 支持并提供了一种使用`net`包编写 TCP 服务器的便捷方法，我们将创建一个简单的 TCP 服务器，它将接受`localhost:8080`上的连接。

- `listener, err := net.Listen(CONN_TYPE, CONN_HOST+":"+CONN_PORT)`：这将在端口`8080`的本地主机上创建一个运行的 TCP 服务器。
- `if err != nil { log.Fatal("Error starting tcp server: ", err) }`：在这里，我们检查启动 TCP 服务器是否有问题。如果存在，则记录错误并以状态代码 1 退出。
- `defer listener.Close()`：此延迟语句在应用关闭时关闭 TCP 套接字侦听器。

访问 http://localhost:8080 ，控制台打印

```
API server listening at: 127.0.0.1:53040
2023/03/28 14:57:29 Listening on localhost:8080
2023/03/28 14:57:32 &{{0xc00007ec80}}
2023/03/28 14:57:52 &{{0xc00007ef00}}
2023/03/28 14:58:00 &{{0xc00007f180}}
2023/03/28 14:58:00 &{{0xc00007f400}}
2023/03/28 14:58:00 &{{0xc00007f680}}
```

2.11.5 tcp-server-read-data & tcp-server-write-data

在一个恒定的循环中接受 TCP 服务器的传入请求，如果在接受请求时有任何错误，那么我们记录并退出；否则，我们只需在服务器控制台上打印 connection 对象。

任何应用中最常见的场景之一是客户端与服务器交互。TCP 是用于这种交互的最广泛使用的协议之一。Go 提供了一种通过`bufio`实现缓冲`Input/Output`读取传入连接数据的方便方法，我们将更新`main()`方法，以调用传递 connection 对象的`handleRequest`方法来读取和打印服务器控制台上的数据。

- 使用`go`关键字从`main()`方法调用`handleRequest`，这意味着我们正在调用 Goroutine 中的函数。
- 定义了`handleRequest`函数，该函数将传入连接读入缓冲区，直到第一次出现`\n`为止，并在控制台上打印消息。如果在读取消息时出现任何错误，则会将错误消息与错误对象一起打印，并最终关闭连接。

- `conn.Write([]byte(message + "\n"))`，将数据作为字节数组写入连接。
- server & client 为一组 tcp 通讯，首先启动监听 tcp-server-read-data/main.go，然后启动 client/main.go 或者 wirter/main.go 都可。

2.11.6 http-server-basic-routing

大多数情况下，您必须在 web 应用中定义多个 URL 路由，这涉及到将 URL 路径映射到处理程序或资源。我们将定义三条路线，例如`/`、`/login`和`/logout`以及它们的处理程序。

一旦我们运行程序，HTTP 服务器将开始在端口`8080`上进行本地监听，并从浏览器或命令行访问`http://localhost:8080/`、`http://localhost:8080/login`和`http://localhost:8080/logout`将呈现相应处理程序定义中定义的消息。

```
$ curl -X GET -i http://localhost:8080/
curl: (7) Failed to connect to localhost port 8080 after 2242 ms: Connection refused

$ curl -X GET -i http://localhost:8080/
HTTP/1.1 200 OK
Date: Tue, 28 Mar 2023 08:02:49 GMT
Content-Length: 12
Content-Type: text/plain; charset=utf-8

Hello World!

$ curl -X GET -i http://localhost:8080/login
HTTP/1.1 200 OK
Date: Tue, 28 Mar 2023 08:02:58 GMT
Content-Length: 11
Content-Type: text/plain; charset=utf-8

Login Page!

$ curl -X GET -i http://localhost:8080/logout
HTTP/1.1 200 OK
Date: Tue, 28 Mar 2023 08:03:10 GMT
Content-Length: 12
Content-Type: text/plain; charset=utf-8

Logout Page!
```

- 定义了三个处理程序或 web 资源，`helloWorld`处理程序在 HTTP 响应流上写入`Hello World!`。以类似的方式，登录和注销处理程序在 HTTP 响应流上写入`Login Page!`和`Logout Page!`。
- 使用`http.HandleFunc()`向`DefaultServeMux`注册了三个 URL 路径—`/`、`/login`和`/logout`。如果传入请求 URL 模式与其中一个注册路径匹配，则相应的处理程序被称为传递`(http.ResponseWriter`、`*http.Request)`作为参数。

2.11.7 http-server-gorilla-mux-routing

Go 的`net/http`包为 HTTP 请求的 URL 路由提供了许多功能。它做得不太好的一件事是动态 URL 路由。幸运的是，我们可以通过`gorilla/mux`包实现这一点。

```
$ go get github.com/gorilla/mux
```

运行程序：

```
$ curl -X GET -i http://localhost:8080/
HTTP/1.1 200 OK
Date: Tue, 28 Mar 2023 08:11:28 GMT
Content-Length: 12
Content-Type: text/plain; charset=utf-8

Hello World!

$ curl -X GET -i http://localhost:8080/hello/foo
HTTP/1.1 200 OK
Date: Tue, 28 Mar 2023 08:11:38 GMT
Content-Length: 6
Content-Type: text/plain; charset=utf-8

Hi foo

$ curl -X POST -i http://localhost:8080/post
HTTP/1.1 200 OK
Date: Tue, 28 Mar 2023 08:12:35 GMT
Content-Length: 20
Content-Type: text/plain; charset=utf-8

It's a Post Request!
```

- 定义了`GetRequestHandler`和`PostRequestHandler`，它们只是在 HTTP 响应流上写消息，`PathVariableHandler`，它提取请求路径变量，获取值，并将其写入 HTTP 响应流。
- 将所有这些处理程序注册到`gorilla/mux`路由器并实例化，调用 mux 路由器的`NewRouter()`处理程序。

在对 web 应用进行故障排除时，记录 HTTP 请求总是很有用的，因此最好使用适当的消息和记录级别记录请求/响应。Go 提供了`log`包，可以帮助我们实现应用的登录。然而，在这个配方中，我们将使用 Gorilla 日志处理程序来实现它，因为该库提供了更多功能，如使用 Apache 组合日志格式和 Apache 通用日志格式进行日志记录，而 Go `log`包尚不支持这些功能。

由于我们已经在前面的配方中创建了 HTTP 服务器并使用 Gorilla Mux 定义了路由，我们将对其进行更新，以合并 Gorilla 日志处理程序。

让我们使用 Gorilla 处理程序实现日志记录。

```
$ go get github.com/gorilla/handlers
$ go get github.com/gorilla/mux
```

运行程序得到 server.log

- 导入了两个额外的包，一个是`os`，用于打开文件。另一个是`github.com/gorilla/handlers`，我们使用它导入日志处理程序来记录 HTTP 请求。
- 将所有这些处理程序注册到`gorilla/mux`路由器并实例化，调用 mux 路由器的`NewRouter()`处理程序。

- 以只写模式创建一个名为`server.log`的新文件，或者打开它（如果它已经存在）。如果有任何错误，则记录并退出，状态代码为 1。
- 使用`router.Handle("/hello/{name}", handlers.CombinedLoggingHandler(logFile, PathVariableHandler)).Methods("GET")`，我们用一个 Gorilla 日志处理程序包装`PathVariableHandler`并将该文件作为写入程序传递给它，这意味着我们只是要求将每个请求以 URL 路径`/hello/{name}`记录在一个名为`server.log`的文件中，该文件采用 Apache 组合日志格式。其它两种同上。

阶段二：使用模板，静态文件和表单

2.11.8 first-template

通常，我们希望创建 HTML 表单，以便以指定的格式从客户端获取信息，将文件或文件夹上载到服务器，并生成通用 HTML 模板，而不是重复相同的静态文本。

使用 Go 的`html/template`包从创建基本模板开始，然后继续从文件系统中提供静态文件，例如`.js`、`.css`和`images`，并最终创建、读取和验证 HTML 表单，并将文件上载到服务器。

模板允许我们为动态内容定义占位符，这些占位符可以在运行时由模板引擎替换为值。然后可以将它们转换为 HTML 文件并发送到客户端。

- `parsedTemplate, _ := template.ParseFiles("./goweb/09first-template/templates/first-template.html")`：这里我们调用`html/template`包的`ParseFiles`，该包创建一个新模板，并解析我们作为输入传递的文件名，该文件名位于模板目录中的`first-template.html ,`。生成的模板将具有输入文件的名称和内容。
- `err := parsedTemplate.Execute(w, person)`：我们在解析模板上调用`Execute`处理程序，将`person`数据注入模板，生成 HTML 输出，并将其写入 HTTP 响应流。
- `if err != nil {log.Printf("Error occurred while executing the template or writing its output : ", err) return }`：这里我们检查执行模板或将其输出写入响应流时是否有问题。如果有，则记录错误并以状态代码 1 退出。

2.11.9 serve-static-files

在设计 web 应用时，最好的做法是从文件系统或任何**内容交付网络（CDN**）提供静态资源，如`.js`、`.css`和`images`，如 Akamai 或 Amazon CloudFront，而不是从 web 服务器提供。这是因为所有这些类型的文件都是静态的，不需要处理；那么，我们为什么要在服务器上增加额外的负载呢？此外，它有助于提高应用的性能，因为对静态文件的所有请求都将从外部源提供，从而减少服务器上的负载。

- **/static/**会匹配 以`/static/`开发的路径，当浏览器请求`index.html`页面中的`main.css`文件时，`static`前缀会被替换为`staic`，然后去`/static/main.css`目录中取查找`main.css`文件。注意 first-template.html 中的 css 引入
- `fileServer := http.FileServer(http.Dir("static"))`：在这里，我们使用`net/http`包的`FileServer`处理程序创建了一个文件服务器，它为来自文件系统上`static`目录的 HTTP 请求提供服务。
- `http.Handle("/static/", http.StripPrefix("/static/", fileServer))`：这里，我们使用`net/http`包的`HandleFunc`将`http.StripPrefix("/static/", fileServer)`处理程序注册为`/static`URL 模式，这意味着无论何时我们使用`/static`模式访问 HTTP URL，都会执行`http.StripPrefix("/static/", fileServer)`并将`(http.ResponseWriter, *http.Request)`作为参数传递给它。
- `http.StripPrefix("/static/", fileServer)`：返回一个处理程序，该处理程序通过从请求 URL 的路径中删除`/static`来服务 HTTP 请求，并调用文件服务器。`StripPrefix`通过 HTTP 404 回复，处理对不以前缀开头的路径的请求。

2.11.10 serve-static-files-gorilla-mux(颜色未成)

在前面，我们通过 Go 的 HTTP 文件服务器提供`static`资源。这次，我们将了解如何通过 Gorilla Mux 路由器提供服务，这也是创建 HTTP 路由器的最常用方法之一。

让我们使用 Gorilla 处理程序实现日志记录。

```
$ go get github.com/gorilla/mux
```

- `router :=mux.NewRouter()`：这里我们实例化了`gorilla/mux`路由器调用 mux 路由器的`NewRouter()`处理程序。
- `router.HandleFunc("/",renderTemplate).Methods("GET")`：这里我们向`renderTemplate`处理程序注册了`/`URL 模式。这意味着`renderTemplate`将对 URL 模式为`/`的每个请求执行。
- `router.PathPrefix("/").Handler(http.StripPrefix("/static", http.FileServer(http.Dir("static/"))))`：这里我们将`/`注册为一个新的路由，并设置一旦调用就要执行的处理程序。
- `http.StripPrefix("/static", http.FileServer(http.Dir("static/")))`：返回一个处理程序，该处理程序通过从请求 URL 的路径中删除`/static`并调用文件服务器来服务 HTTP 请求。`StripPrefix`通过 HTTP 404 回复，处理对不以前缀开头的路径的请求。

2.11.11 html-form

每当我们想要从客户机收集数据并将其发送到服务器进行处理时，实现 HTML 表单是最佳选择。我们将创建一个简单的 HTML 表单，它有两个输入字段和一个提交表单的按钮。

- `func login(w http.ResponseWriter, r *http.Request) { parsedTemplate, _ := template.ParseFiles("./goweb/12html-form/templates/login-form.html") parsedTemplate.Execute(w, nil) }`：这是一个 Go 函数，接受`ResponseWriter`和`Request`作为输入参数，解析`login-form.html`，并返回一个新模板。

2.11.12 html-form-read

提交 HTML 表单后，我们必须读取服务器端的客户机数据以采取适当的操作。扩展 html-form 以读取其字段值。

```
$ go get github.com/gorilla/schema
```

定义了`readForm`处理程序，它以`HTTP Request`为输入参数，返回`User`。然后定义了一个`login`处理程序，检查调用该处理程序的 HTTP 请求是否为`GET`请求，然后从 templates 目录解析`login-form.html`并写入 HTTP 响应流；否则调用`readForm`处理程序。

- `r.ParseForm()`：这里我们将请求主体解析为一个表单，并将结果放入`r.PostForm`和`r.Form`中。
- `user := new(User)`：这里我们创建一个新的`User struct`类型。
- `decoder := schema.NewDecoder()`：我们正在创建一个解码器，我们将使用它向用户`struct`填充`Form`值。
- `decodeErr := decoder.Decode(user, r.PostForm)`：这里我们将解析后的表单数据从`POST`主体参数解码给用户`struct`。`r.PostForm` is only available after `ParseForm` is called.
- `if decodeErr != nil { log.Printf("error mapping parsed form data to struct : ", decodeErr) }`：这里我们检查表单数据到结构的映射是否有问题，如果有，则记录。

2.11.13 html-form-validation

大多数情况下，我们必须在处理客户的输入之前对其进行验证，这可以通过 Go 中的外部包数来实现，例如 `gopkg.in/go-playground/validator.v9`、`gopkg.in/validator.v2`、`github.com/asaskevich/govalidator`。

在这个配方中，我们将使用最著名和最常用的验证器`github.com/asaskevich/govalidator`，来验证我们的 HTML 表单。扩展 html-form-read 以验证其字段值。

```
$ go get github.com/asaskevich/govalidator
$ go get github.com/gorilla/schema
```

读取一个 HTML 表单，使用`github.com/gorilla/schema`对其进行解码，并使用`github.com/asaskevich/govalidator`根据`User struct`中定义的标记对其每个字段进行验证，

- `更新了 User struct 类型以包含一个字符串文字标记，其中 key 为 valid，而 value 为 alpha, required`。
- 定义了一个`validateUser`处理程序，该处理程序以`ResponseWriter`、`Request`和`User`作为输入，并返回一个`bool`和`string`，分别是结构的有效状态和验证错误消息。验证从`govalidator`调用`ValidateStruct`处理程序的结构标记。如果验证字段时出错，那么我们从`govalidator`获取调用`ErrorByField`处理程序的错误，并将结果与验证错误消息一起返回。
- 更新了`login`处理程序，以调用`validateUser`传递`(w http.ResponseWriter, r *http.Request, user *User)`作为输入参数，并检查是否存在任何验证错误。如果有错误，那么我们将错误消息写入 HTTP 响应流并返回它。

2.11.14 upload-file

任何 web 应用中最常见的场景之一是将文件或文件夹上载到服务器。例如，如果我们正在开发一个工作门户，那么我们可能必须提供一个选项，申请人可以上传他们的个人资料/简历，或者，比方说，我们必须开发一个电子商务网站，其中客户可以使用文件批量上传订单，使用 Go 的内置软件包实现在 Go 中上传文件的功能非常简单。

将创建一个 HTML 表单，其字段类型为`file`，用户可以通过表单提交选择一个或多个文件上传到服务器。定义了`fileHandler()`处理程序，它从请求中获取文件，读取其内容，并最终将其写入服务器上的文件。

- `file, header, err := r.FormFile("file")`：这里我们调用 HTTP 请求上的`FormFile`处理程序来获取所提供表单密钥的文件。
- `if err != nil { log.Printf("error getting a file for the provided form key : ", err) return }`：这里我们检查从请求中获取文件时是否有问题。如果有，则记录错误并使用状态代码`1`退出。
- `defer file.Close()`：`defer`语句在我们从函数返回后关闭`file`。
- `out, pathError := os.Create("/tmp/uploadedFile")`：我们在`/tmp`目录中创建一个名为`uploadedFile`的文件，模式为`666`，这意味着客户端可以读写，但不能执行该文件。
- `if pathError != nil { log.Printf("error creating a file for writing : ", pathError) return }`：我们在这里检查在服务器上创建文件是否有问题。如果有，则记录错误并使用状态代码`1`退出。
- `_, copyFileError := io.Copy(out, file)`：这里我们将收到的文件中的内容复制到`/tmp`目录中创建的文件中。
- `fmt.Fprintf(w, "File uploaded successfully : "+header.Filename)`：这里我们将一条消息和一个文件名一起写入 HTTP 响应流。

阶段三：会话，缓存和错误处理

有时，我们希望将诸如用户数据之类的信息持久化到应用级别，而不是将其持久化到数据库中，这可以通过会话和 cookie 轻松实现。两者的区别在于会话存储在服务器端，而 cookie 存储在客户端。我们可能还需要缓存静态数据，以避免对数据库或 web 服务进行不必要的调用，并在开发 web 应用时实现错误处理。

我们将从创建 HTTP 会话开始，然后学习如何使用 Redis 管理会话，创建 cookie，缓存 HTTP 响应，实现错误处理，最后在 Go 中实现登录和注销机制。

HTTP 是一种无状态协议，这意味着每次客户端检索网页时，客户端都会打开一个与服务器的单独连接，服务器会响应该连接，而不会保留以前客户端请求的任何记录。因此，如果我们想要实现一种机制，其中服务器知道客户端发送给它的请求，那么我们可以使用会话来实现它。

当我们使用会话时，客户端只需要发送一个 ID，并从服务器加载相应 ID 的数据。我们可以通过三种方式在 web 应用中实现这一点：

- 曲奇饼 cookies
- 隐藏表单字段
- URL 重写

2.11.15 http-session

我们将使用 HTTP cookies 实现一个会话。

```
$ go get github.com/gorilla/sessions
```

创建一个 Gorilla cookie 存储，以保存和检索会话信息，定义三个处理程序—`/login`、`/home`和`/logout`——其中我们将创建有效的会话 cookie，将响应写入 HTTP 响应流，并分别使会话 cookie 无效。

- 使用`var store *sessions.CookieStore`，我们声明了一个私有 cookie 存储，以使用安全 cookie 存储会话。
- 使用`func init() { store = sessions.NewCookieStore([]byte("secret-key")) }`，我们定义了一个在`main()`之前运行的`init()`函数来创建一个新的 cookie 存储并将其分配给`store`。`init()` is always called, regardless of whether there's a main function or not, so if you import a package that has an `init` function, it will be executed.
- 接下来，我们定义了一个`home`处理程序，在使用`store.Get`将会话添加到注册表并从缓存中获取`authenticated`键的值之后，我们从 cookie 存储中获取给定名称的会话。如果为真，则将`Home Page`写入 HTTP 响应流；否则，我们会写一封未经授权查看页面的邮件，并附上一个`403` HTTP 代码。
- 接下来，我们定义了一个`login`处理程序，在这里我们再次获得一个会话，将`authenticated`键设置为`true`值，保存它，最后将您已成功登录写入 HTTP 响应流。
- 接下来，我们定义了一个`logout`处理程序，在其中我们获得一个会话，设置一个`authenticated`键，值为`false`，保存它，最后将您已成功注销写入 HTTP 响应流。
- 最后，我们定义了`main()`，将所有处理程序`home`、`login`和`logout`分别映射到`/home`、`/login`和`/logout`，并在`localhost:8080`上启动 HTTP 服务器。

2.11.16 http-session-redis(未做)

在使用分布式应用时，我们可能必须为前端用户实现无状态负载平衡。这样我们就可以在数据库或文件系统中持久保存会话信息，以便在服务器关闭或重新启动时识别用户并检索他们的信息。

我们将解决这个问题，作为使用 Redis 作为持久存储来保存会话的方法的一部分。

由于我们已经使用 Gorilla cookie store 在之前的配方中创建了一个会话变量，因此我们将扩展此配方以在 Redis 中保存会话信息，而不是在服务器上维护它。

Gorilla 会话存储有多种实现，您可以在`https://github.com/gorilla/sessions#store-implementations`中找到。由于我们使用 Redis 作为后端存储，我们将使用`https://github.com/boj/redistore`，它依赖于 Redigo Redis 库来存储会话。

此方法假设您已在端口 `6379` 和 `4567` 上分别安装并本地运行 Redis 和 Redis Browser。

```
$ go get gopkg.in/boj/redistore.v1
$ go get github.com/gorilla/sessions
```

- 使用 `var store *redisStore.RediStore`，我们声明了一个私有 `RediStore` 来在 Redis 中存储会话。
- 接下来，我们更新了 `init()` 函数以创建 `NewRediStore`，其中空闲连接的大小和最大数量为 `10`，并将其分配给存储。如果在创建存储时出现错误，那么我们将记录错误并使用状态代码 `1` 退出。
- 最后，我们更新了 `main()` 以引入 `defer store.Close()` 语句，一旦我们从函数返回，它将关闭 Redis 存储。

```
package main
import
(
  "fmt"
  "log"
  "net/http"
  "github.com/gorilla/sessions"
  redisStore "gopkg.in/boj/redistore.v1"
)
const
(
  CONN_HOST = "localhost"
  CONN_PORT = "8080"
)
var store *redisStore.RediStore
var err error
func init()
{
  store, err = redisStore.NewRediStore(10, "tcp", ":6379", "",
  []byte("secret-key"))
  if err != nil
  {
    log.Fatal("error getting redis store : ", err)
  }
}
func home(w http.ResponseWriter, r *http.Request)
{
  session, _ := store.Get(r, "session-name")
  var authenticated interface{} = session.Values["authenticated"]
  if authenticated != nil
  {
    isAuthenticated := session.Values["authenticated"].(bool)
    if !isAuthenticated
    {
      http.Error(w, "You are unauthorized to view the page",
      http.StatusForbidden)
      return
    }
    fmt.Fprintln(w, "Home Page")
  }
  else
  {
```

```
    http.Error(w, "You are unauthorized to view the page",
    http.StatusForbidden)
    return
  }
}
func login(w http.ResponseWriter, r *http.Request)
{
  session, _ := store.Get(r, "session-name")
  session.Values["authenticated"] = true
  if err = sessions.Save(r, w); err != nil
  {
    log.Fatalf("Error saving session: %v", err)
  }
  fmt.Fprintln(w, "You have successfully logged in.")
}
func logout(w http.ResponseWriter, r *http.Request)
{
  session, _ := store.Get(r, "session-name")
  session.Values["authenticated"] = false
  session.Save(r, w)
  fmt.Fprintln(w, "You have successfully logged out.")
}
func main()
{
  http.HandleFunc("/home", home)
  http.HandleFunc("/login", login)
  http.HandleFunc("/logout", logout)
  err := http.ListenAndServe(CONN_HOST+":"+CONN_PORT, nil)
  defer store.Close()
  if err != nil
  {
    log.Fatal("error starting http server : ", err)
    return
  }
}
```

2.11.17 http-cookie

Cookie 在客户端存储信息时起着重要作用，我们可以使用它们的值来识别用户。基本上，Cookie 的发明是为了解决记住用户信息或持久登录身份验证的问题，持久登录身份验证指的是网站能够在会话之间记住主体的身份。

Cookie 是 web 浏览器在您访问 internet 上的网站时创建的简单文本文件。您的设备将文本文件存储在本地，允许浏览器访问 cookie 并将数据传递回原始网站，并以名称-值对的形式保存。

```
$ go get github.com/gorilla/securecookie
```

创建一个 Gorilla 安全 cookie 来存储和检索 cookie，

- 使用`var cookieHandler *securecookie.SecureCookie`，我们声明了一个私有安全 cookie。
- 接下来，我们更新了`init()`函数，创建`SecureCookie`传递 64 字节的散列密钥，该散列密钥用于使用 HMAC 验证值，32 字节的块密钥用于加密值。

- 接下来，我们定义了一个`createCookie`处理程序，其中我们使用`gorilla/securecookie`的`Encode`处理程序创建一个`Base64`编码的 cookie，其键为`username`，值为`Foo`。然后，我们在提供的`ResponseWriter`头中添加`Set-Cookie`头，并将`Cookie created.`消息写入 HTTP 响应。
- 接下来，我们定义了一个`readCookie`处理程序，在这里我们从请求中检索一个 cookie，在我们的代码中是`first-cookie`，为它获取一个值，并将其写入 HTTP 响应。

2.11.18 http-caching

在 web 应用中缓存数据有时是必要的，以避免一次又一次地从数据库或外部服务请求静态数据。Go 不提供任何内置包来缓存响应，但它通过外部包来支持它。

有许多包，例如`https://github.com/coocood/freecache`和`https://github.com/patrickmn/go-cache`，可以帮助实现缓存，在这个配方中，我们将使用`https://github.com/patrickmn/go-cache`来实现缓存。

```
$ go get github.com/patrickmn/go-cache
```

- 使用`var newCache *cache.Cache`，我们声明了一个私有缓存。
- 接下来，我们更新了`init()`函数，其中我们创建了一个有 5 分钟过期时间和 10 分钟清理间隔的缓存，并向缓存中添加了一个项，该项的键为`foo`，其值为`bar`，其过期值为`0`，这意味着我们要使用缓存的默认过期时间。如果过期持续时间小于一（或`NoExpiration`，则缓存中的项目永远不会过期（默认），必须手动删除。如果清除间隔小于 1，则在调用`c.DeleteExpired()`之前不会从缓存中删除过期项目。
- 接下来，我们定义了`getFromCache`处理程序，在该处理程序中，我们从缓存中检索密钥的值。如果找到，我们将其写入 HTTP 响应；否则，我们将`Key Not Found in Cache`消息写入 HTTP 响应。

2.11.19 http-error-handling

在任何 web 应用中实现错误处理是主要方面之一，因为它有助于更快地排除故障和修复错误。错误处理意味着，每当应用中发生错误时，应该将其记录在某个地方，或者记录在文件中，或者记录在数据库中，并带有正确的错误消息以及堆栈跟踪。

在 Go 中，它可以以多种方式实现。一种方法是编写自定义处理程序，

```
$ go get github.com/gorilla/mux
```

- 定义了一个具有两个字段的`NameNotFoundError`结构-`int`类型的`Code`和`error`类型的`Err`，这表示一个与相关 HTTP 状态代码相关的错误。
- 允许`NameNotFoundError`满足错误接口。
- 定义了一个用户定义的类型`WrapperHandler`，这是一个 Go 函数，它接受任何接受`func(http.ResponseWriter, *http.Request)`作为输入参数的处理程序，并返回一个错误。
- 定义了一个`ServeHTTP`处理程序，它调用我们传递给`WrapperHandler`的处理程序，将`(http.ResponseWriter, *http.Request)`作为参数传递给它，并检查处理程序是否返回任何错误。如果有，则使用开关盒对其进行适当处理。
- 定义了一个`getName`处理程序，它提取请求路径变量，获取`name`变量的值，并检查名称是否与`foo`匹配。如果是这样，那么它会将 Hello（后跟名称）写入 HTTP 响应；否则，它返回一个`Code`字段值为`500`的`NameNotFoundError`结构和一个带文本`Name Not Found`的`err`字段值为`error`的`NameNotFoundError`结构。
- 最后，我们定义了`main()`，其中我们注册`WrapperHandler`作为 URL 模式调用的处理程序`/get/{name}`。

2.11.20 html-form-login-logout

每当我们希望注册用户访问应用时，我们都必须实现一种机制，在允许用户查看任何网页之前要求用户提供凭据，将使用`gorilla/securecookie`包对其进行更新，以实现登录和注销机制。

```
$ go get github.com/gorilla/mux
$ go get github.com/gorilla/securecookie
```

- 使用`var cookieHandler = securecookie.New(securecookie. GenerateRandomKey(64), securecookie.GenerateRandomKey(32))`，我们正在创建一个安全 cookie，将哈希键作为第一个参数传递，将块键作为第二个参数传递。散列密钥用于使用 HMAC 对值进行身份验证，块密钥用于对值进行加密。
- `getUserName`处理程序，从 HTTP 请求中获取 cookie，初始化字符串`keys`到字符串`values`的`cookieValue`映射，解码 cookie，并获取用户名和返回值。
- `setSession`处理程序，在这里我们创建并初始化一个映射，`key`和`value`为`username`，**将其序列化，使用消息身份验证代码对其签名，使用cookieHandler.Encode处理程序对其编码，创建一个新的 HTTP cookie，并将其写入 HTTP 响应流。**
- `clearSession`，它基本上将 cookie 的值设置为空，并将其写入 HTTP 响应流。
- `login`处理程序，从 HTTP 表单中获取用户名和密码，检查两者是否都为空，然后调用`setSession`处理程序并重定向到`/home`，否则重定向到根 URL`/`。
- `logout`处理程序，在这里我们清除调用`clearSession`处理程序的会话值并重定向到根 URL。
- `loginPage`处理程序，在这里我们解析`login-form.html`，返回一个带有名称及其内容的新模板，调用解析模板上的`Execute`处理程序，该处理程序生成 HTML 输出，并将其写入 HTTP 响应流。
- `homePage`处理程序，它从调用`getUserName`处理程序的 HTTP 请求中获取用户名。然后，我们检查它是否为空或是否存在 cookie 值。如果用户名不为空，解析`home.html`，将用户名作为数据映射注入，生成 HTML 输出，写入 HTTP 响应流；否则，我们将其重定向到根 URL`/`。

最后，我们定义了`main()`方法，在这里我们开始执行程序。由于此方法可以做很多事情，让我们逐行查看：

- `var router = mux.NewRouter()`：这里我们创建一个新的路由器实例。
- `router.HandleFunc("/", loginPage)`：这里，我们使用`gorilla/mux`包的`HandleFunc`将`loginPageHandler`处理程序注册为`/`URL 模式，这意味着每当我们使用`/`模式访问 HTTP URL 时，`loginPage`处理程序通过将`(http.ResponseWriter, *http.Request)`作为参数传递给它来执行。
- `router.HandleFunc("/home", homePage)`：在这里，我们使用`gorilla/mux`包的`HandleFunc`将`homePageHandler`处理程序注册到`/home`URL 模式，这意味着每当我们使用`/home`模式访问 HTTP URL 时，`homePage`处理程序通过将`(http.ResponseWriter, *http.Request)`作为参数传递给它来执行。
- `router.HandleFunc("/login", login).Methods("POST")`：在这里，我们使用`gorilla/mux`包的`HandleFunc`将`loginHandler`处理程序注册到`/login`URL 模式，这意味着每当我们使用`/login`模式访问 HTTP URL 时，`login`处理程序通过将`(http.ResponseWriter, *http.Request)`作为参数传递给它来执行。
- `router.HandleFunc("/logout", logout).Methods("POST")`：在这里，我们使用`gorilla/mux`包的`HandleFunc`将`logoutHandler`处理程序注册到`/logout`URL 模式，这意味着每当我们使用`/logout`模式访问 HTTP URL 时，`logout`处理程序通过将`(http.ResponseWriter, *http.Request)`作为参数传递给它来执行。
- `http.Handle("/", router)`：在这里，我们使用`net/http`包的`HandleFunc`为`/`URL 模式注册路由器，这意味着所有具有`/`URL 模式的请求都由路由器处理程序处理。
- `err := http.ListenAndServe(CONN_HOST+":"+CONN_PORT, nil)`：在这里，我们调用`http.ListenAndServe`为 HTTP 请求提供服务，这些请求在单独的 Goroutine 中处理每个传入连接。`ListenAndServe`接受两个参数 server address 和 handler，其中 server address 为`localhost:8080`，handler 为`nil`，这意味着我们要求服务器使用`DefaultServeMux`作为处理程序。

- `if err != nil { log.Fatal("error starting http server : ", err) return}`：在这里，我们检查启动服务器是否有问题。如果有，则记录错误并使用状态代码`1`退出。

2.11.21 http-rest&http-rest-client

每当我们构建一个封装了对其他相关应用有帮助的逻辑的 web 应用时，我们通常也会编写和使用 web 服务。这是因为它们通过网络公开功能，而网络可以通过 HTTP 协议访问，从而使应用成为唯一的真实来源。

在编写 web 应用时，我们通常必须向客户机或 UI 公开我们的服务，以便他们能够使用在不同系统上运行的代码。可以使用 HTTP 协议方法公开服务。

编写一个支持`GET`、`POST`、`PUT`和`DELETE`HTTP 方法的 REST 式 API，对 REST API 进行版本化。

```
$ go get github.com/gorilla/mux

$ go get -u gopkg.in/resty.v1
```

- 在设计 RESTURL 时，如果客户端查询端点而不在 URL 路径中指定版本，我们更愿意返回默认数据。默认提供一个包含单个记录的列表，我们可以将其称为 REST 端点的默认或初始响应。
- 大多数与服务器通信的应用都使用 RESTful 服务。根据我们的需要，我们通过 JavaScript、jQuery 或 REST 客户机使用这些服务。使用`https://gopkg.in/resty.v1`包编写一个 REST 客户机，它本身就是受 Ruby REST 客户机启发来使用 RESTful 服务的。

2.12 cipher

加密算法

- base 64

通过 base64.StdEncoding.EncodeToString([]byte("admin:admin")) 加密 string 信息为 admin:admin