搭建Web服务器

前面小节已经介绍了Web是基于http协议的一个服务，Go语言里面提供了一个完善的net/http包，通过http包可以很方便的就搭建起来-一个可以运行的web服务。同时使用这个包能很简单地对web的路由，静态文件，模版，cookie等数据进行设置和操作。

一、创建第一个GoWeb程序：Hello GoWeb

打开Goland开发工具并创建一个项目：http。再创建一个go文件(demo01_hello.go)，并输入以下代码：

```
package main

import (
    "net/http"
    "log"
    "fmt"
    "strings"
)

/**
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）。
 */
func sayHello(w http.ResponseWriter, r *http.Request) {
    //w.Write([]byte("hello 世界！"))
    fmt.Println("------------------")
    r.ParseForm()       //解析参数，默认是不会解析的
    fmt.Println(r.Form) // 这些信息是输出到服务器端的打印信息
    fmt.Println("path：", r.URL.Path)
    fmt.Println("scheme：", r.URL.Scheme)

    fmt.Println(r.Form["url_ long"])
```

```
    for k, v := range r.Form {
        fmt.Println("key:", k)
        fmt.Println("val:", strings.Join(v, ""))
    }
    fmt.Fprintf(w, "Hello GoWeb!") //这个写入到w的是输出到客户端的
}
func main() {
    /**
    第一个参数：pattern string,
    第二个参数：handler func(ResponseWriter, *Request)
     */
    http.HandleFunc("/", sayHello)          // 设置访问的路由
    /**
    第一个参数addr：监听地址
    第二个参数handler：通常为空，意味着服务端调用http.DefaultServerMux进行处理，而服务端编写的业务逻辑处理程序http.Handle()或http.HandleFunc()默认注入http.DefaultServeMux中
     */
    err := http.ListenAndServe(":8080", nil) //设置监听的端口
    if err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}

/*
1.运行该程序
2.打开浏览器，输入：http://localhost:8080
 */
```

然后打开浏览器：输入以下网址：

```
http://localhost:8080/
```

页面显示内容：

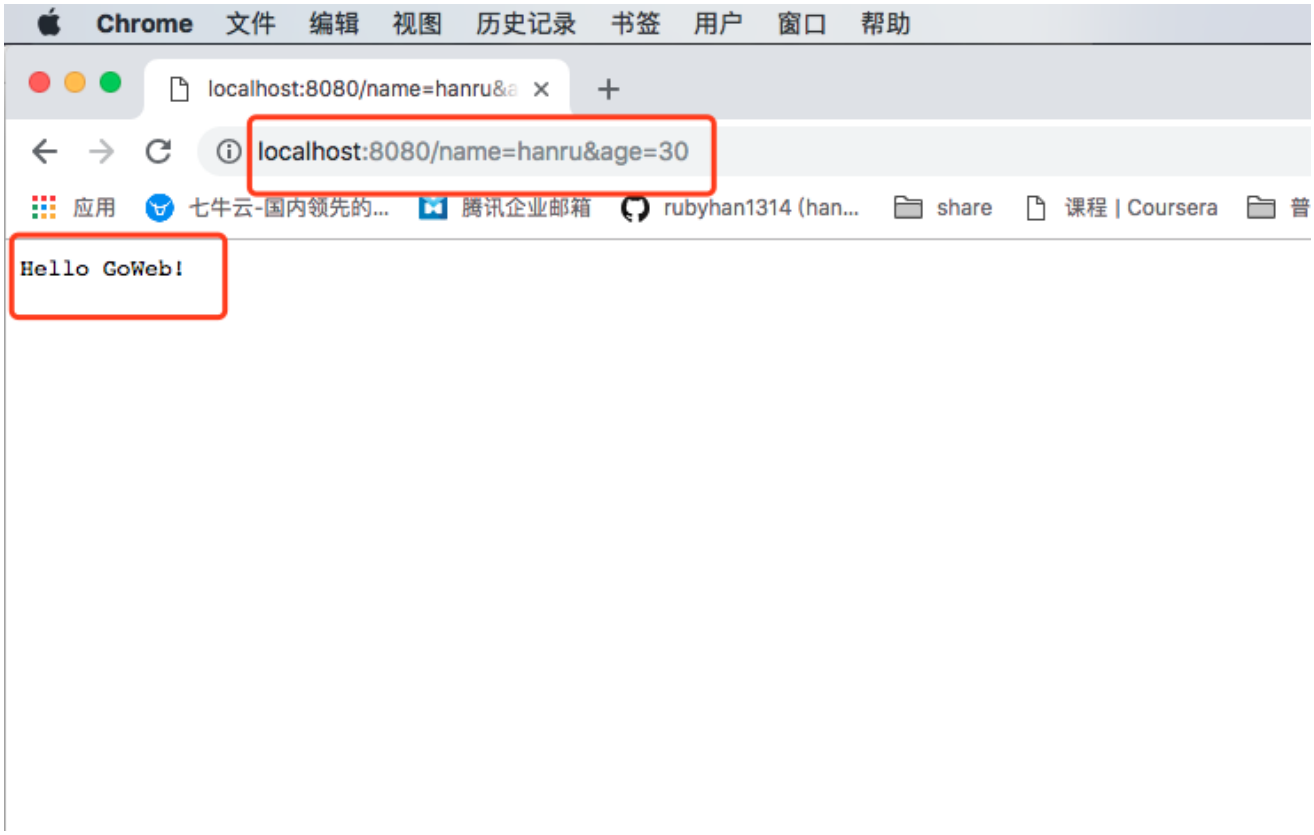
服务器端运行结果如下：

```
------------------
map[]
path： /
scheme：
[]
------------------
map[]
path： /favicon.ico
scheme：
[]
```

重新输入新的网址：

```
http://localhost:8080/name=hanru&age=30
```

页面显示内容没有变化：



但是服务器端，运行结果如下：

```
-----------------
map[]
path： /name=hanru&age=30
scheme：
[]
-----------------
map[]
path： /favicon.ico
scheme：
[]
```

我们看到上面的代码，要编写一个web服务器很简单，只要调用http包的两个函数就可以了。

如果你以前是PHP程序员，那你也许就会问，我们的nginx、apache服务器不需要吗？Go就是不需要这些，因为他直接就监听tcp端口了，做了nginx做的事情，然后sayhelloName这个其实就是我们写的逻辑函数了，跟php里面的控制层（controller）函数类似。

如果你以前是python程序员，那么你一定听说过tornado，这个代码和他是不是很像，对，没错，go就是拥有类似python这样动态语言的特性，写web应用很方便。

如果你以前是ruby程序员，会发现和ROR的/script/server启动有点类似。

我们看到Go通过简单的几行代码就已经运行起来一个web服务了，而且这个Web服务内部有支持高并发的特性，我将会

在接下来我们详细的讲解一下go是如何实现Web高并发的。

二、Go如何使得Web工作

我们可以通过net/http包，很方便的搭建了一个简单应用。那么Go在底层到底是怎么做的呢？首先我们介绍web工作方式的几个概念。

2.1 web中的几个概念

以下均是服务器端的几个概念：

Request：用户请求的信息，用来解析用户的请求信息，包括post、get、cookie、url等信息

Response：服务器需要反馈给客户端的信息

Conn：用户的每次请求链接

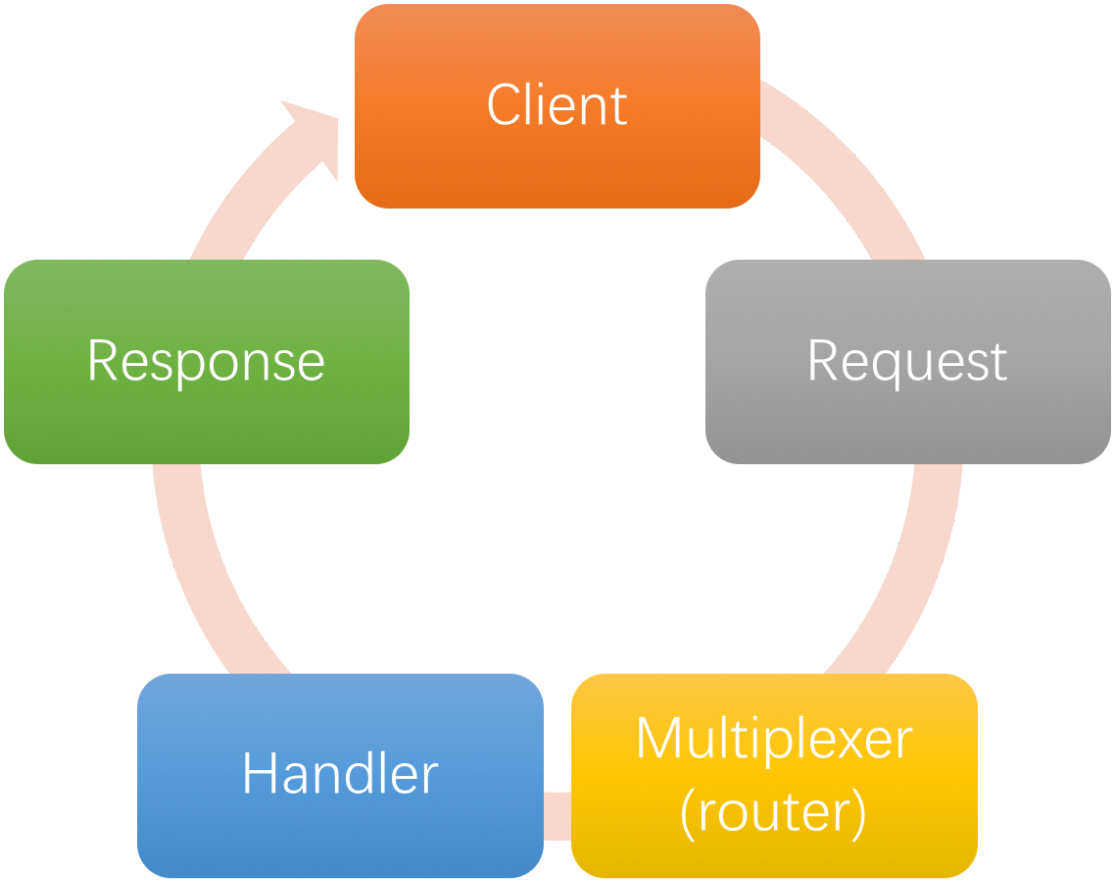
Handler：处理请求和生成返回信息的处理逻辑

除去细节，理解HTTP构建的网络应用只要关注两个端--客户端(client)和服务端(server)，两个端的交互来自client的request，以及server端的response。所谓的http服务器，主要在于如何接受client的request，并向client返回response。

接收request的过程中，最重要的莫过于路由(router)，即实现一个Multiplexer器。Go中既可以使用内置的mutilplexer--DefaultServeMux,也可以自定义。Multiplexer路由的目的就是为了找到处理器函数(hander),后者将对request进行处理，同时构建response。

2.2 Go实现的流程

简单总结就是这个流程：



> Client --> Request --> Multiplexer(router) --> Handler --> Response --
> >Client

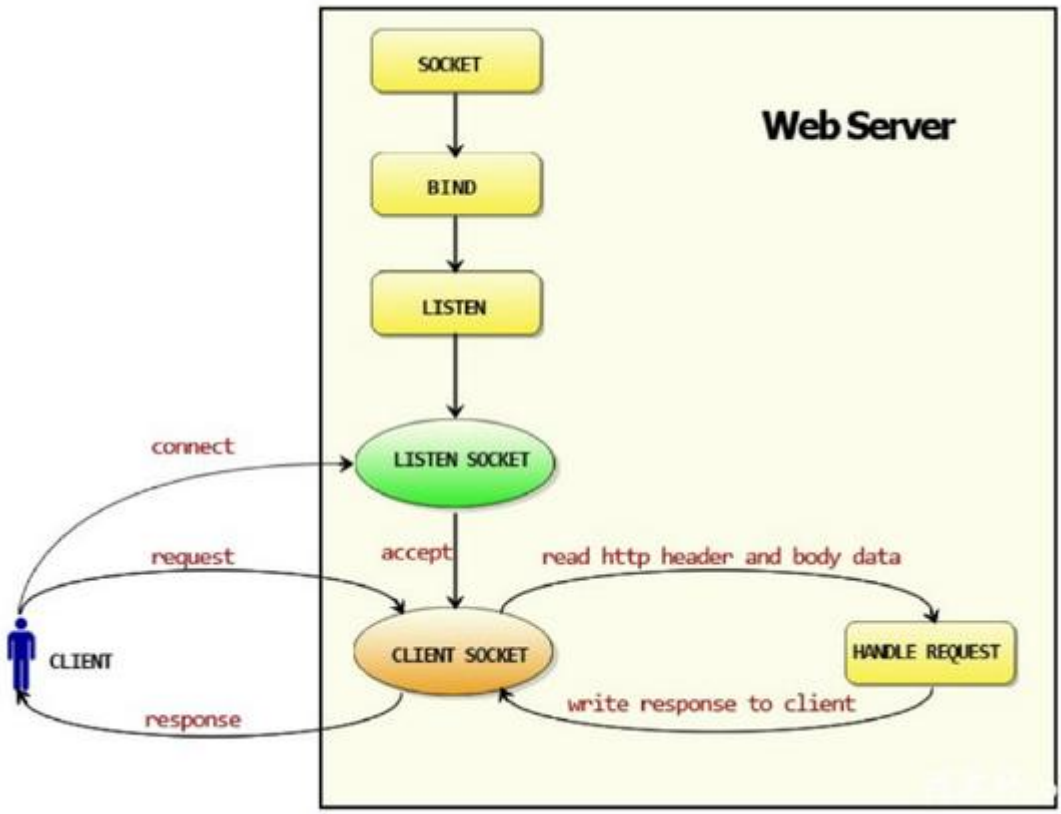
因此，理解go中的http服务，最重要的就是要理解Multiplexer和hander，Golang中的Multiplexer基于ServerMux结构，同时也实现了Handler接口。

- hander函数：具有func(w http.ResponseWriter, r *http.Requests)签名的函数
- handler处理器(函数)：经过HanderFunc结构包装的handler函数，它实现了ServeHTTP接口方法的函数。调用handler处理器的ServeHTTP方法时，即调用handler函数本身。
- handler对象：实现了Hander接口ServeHTTP方法的结构。

> Handler处理器和handler对象的差别在于，一个是函数，另一个是结构，它们都实现了ServerHttp方法。很多情况下它们的功能类似。

2.3 分析http包运行机制

下面是Go实现Web服务的工作模式的流程图



http包执行流程：

1. 创建Listen Socket, 监听指定的端口, 等待客户端请求到来。
2. Listen Socket接受客户端的请求, 得到Client Socket, 接下来通过Client Socket与客户端通信。
3. 处理客户端的请求, 首先从Client Socket读取HTTP请求的协议头, 如果是POST方法, 还可能要读取客户端提交的数据, 然后交给相应的handler处理请求, handler处理完毕准备好客户端需要的数据, 通过Client Socket写给客户端。

这整个的过程里面我们只要了解清楚下面三个问题，也就知道Go是如何让Web运行起来了

- 如何监听端口？
- 如何接收客户端请求？
- 如何分配handler？

通过前面的代码我们可以看到，Go是通过一个函数ListenAndServe来处理这些事情的，这个底层其实这样处理的：初始化一个server对象，然后调用了net.Listen("tcp", addr)，也就是底层用TCP协议搭建了一个服务，然后监控我们设置的端口。

下面代码来自Go的http包的源码，通过下面的代码我们可以看到整个的http处理过程：

```
func (srv *Server) Serve(l net.Listener) error {
    defer l.Close()
    if fn := testHookServerServe; fn != nil {
        fn(srv, l)
    }
    var tempDelay time.Duration // how long to sleep on accept failure

    if err := srv.setupHTTP2_Serve(); err != nil {
        return err
    }

    srv.trackListener(l, true)
    defer srv.trackListener(l, false)



    baseCtx := context.Background() // base is always background, per Issue
```

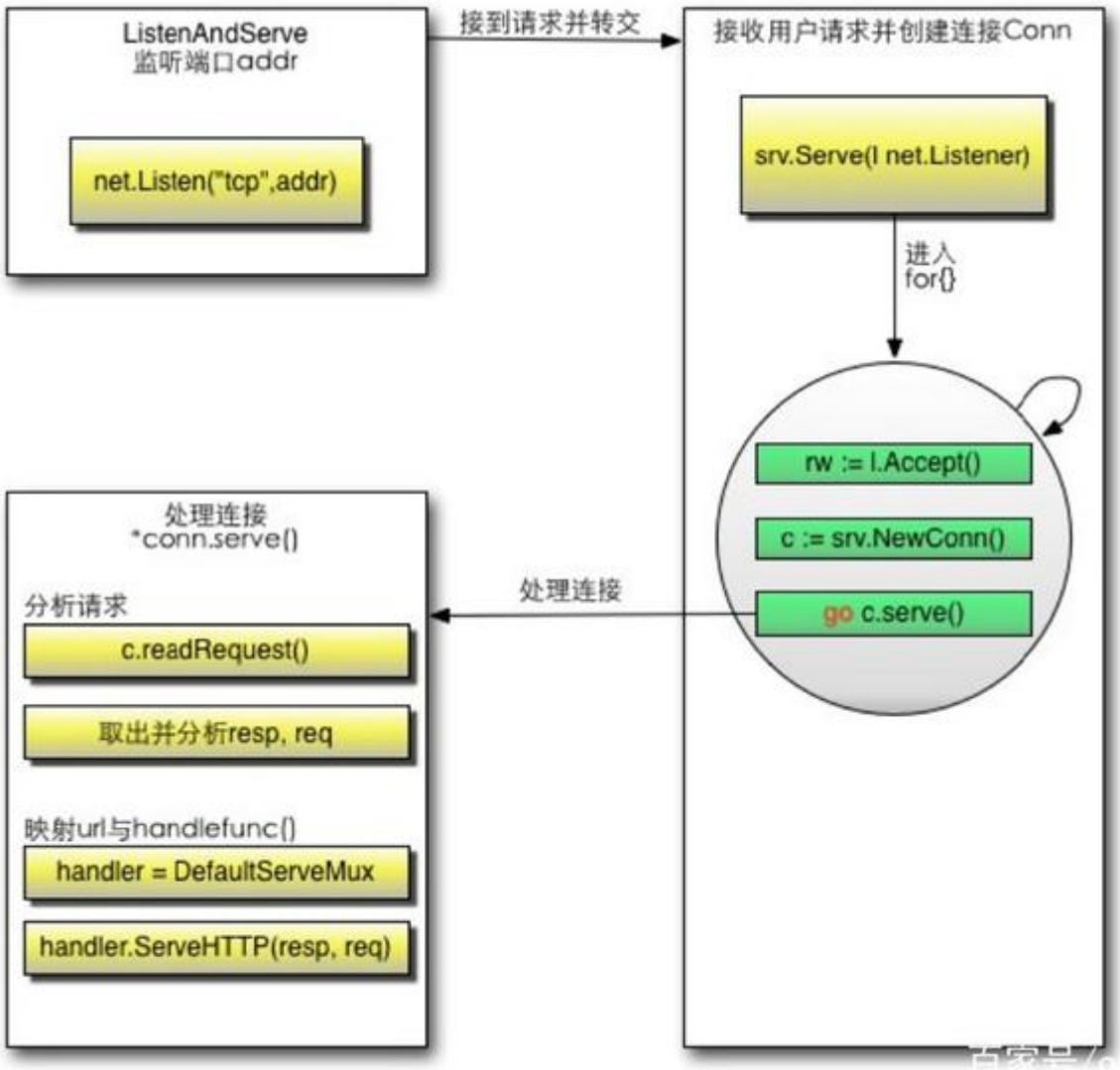
```
16220
    ctx := context.WithValue(baseCtx, ServerContextKey, srv)
    for {
        rw, e := l.Accept()
        if e != nil {
            select {
            case <-srv.getDoneChan():
                return ErrServerClosed
            default:
            }
            if ne, ok := e.(net.Error); ok && ne.Temporary() {
                if tempDelay == 0 {
                    tempDelay = 5 * time.Millisecond
                } else {
                    tempDelay *= 2
                }
                if max := 1 * time.Second; tempDelay > max {
                    tempDelay = max
                }
                srv.logf("http: Accept error: %v; retrying in %v", e, tempDelay)
                time.Sleep(tempDelay)
                continue
            }
            return e
        }
        tempDelay = 0
        c := srv.newConn(rw)
        c.setState(c.rwc, StateNew) // before Serve can return
        go c.serve(ctx)
    }
}
```

监控之后如何接收客户端的请求呢？上面代码执行监控端口之后，调用了srv.Serve(net.Listener)函数，这个函数就是处理接收客户端的请求信息。这个函数里面起了一个for{}，首先通过Listener接收请求，其次创建一个Conn，最后单独开了一个goroutine，把这个请求的数据当做参数扔给这个conn去服务：go

c.serve()。这个就是高并发体现了，用户的每一次请求都是在一个新的goroutine去服务，相互不影响。

那么如何具体分配到相应的函数来处理请求呢？conn首先会解析request:c.readRequest(),然后获取相应的handler:handler := c.server.Handler，也就是我们刚才在调用函数ListenAndServe时候的第二个参数，我们前面例子传递的是nil，也就是为空，那么默认获取handler = DefaultServeMux，那么这个变量用来做什么的呢？对，这个变量就是一个路由器，它用来匹配url跳转到其相应的handle函数，那么这个我们有设置过吗?有，我们调用的代码里面第一句不是调用了http.HandleFunc("/", sayhello)嘛。这个作用就是注册了请求/的路由规则，当请求uri为"/"，路由就会转到函数sayhello，DefaultServeMux会调用ServeHTTP方法，这个方法内部其实就是调用sayhello本身，最后通过写入response的信息反馈到客户端。

详细的整个流程如下图所示：



梳理代码执行过程

1. 首先调用Http.HandleFunc，按顺序做如下操作调用DefaultServerMux的HandleFunc调用DefaultServerMux的Handle往DefaultServeMux的map[string]muxEntry中增加对应的handler和路由规则
2. 其次调用http.ListenAndServe，按顺序做如下操作实例化Server调用Server的ListenAndServe()调用net.listen("tcp",addr)监听端口启动一个for循环，在循环体中Accept请求对每个请求实例化一个Conn，并且开启一个goroutine为这个请求进行服务go c.serve()读取每个请求的内容w, err := c.readRequest()判断handler是否为空，如果没有设置handler，handler就设置为DefaultServeMux。

> 这个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。

3. 调用handler的ServeHttp根据request选择handler，并且进入到这个handler的ServeHttp选择handler判断是否有路由能满足这个request如果有路由满足，调用路由handler的ServeHttp如果没有路由满足，调用NotFoundHandler的ServeHttp

三、Go的http实现

接下来我们将详细地解剖一下http包，看它到底是怎样实现整个过程的。

Go的http有两个核心功能：Conn、ServeMux

3.1 Conn的goroutine

与我们一般编写的http服务器不同, Go为了实现高并发和高性能, 使用了goroutines来处理Conn的读写事件, 这样每个请求都能保持独立，相互不会阻塞，可以高效的响应网络事件。这是Go高效的保证。

Go在等待客户端请求里面是这样写的：

```
c, err := srv.newConn(rw)

if err != nil {
   continue
}
go c.serve()
```

这里我们可以看到客户端的每次请求都会创建一个Conn，这个Conn里面保存了该次请求的信息，然后再传递到对应的handler，该handler中便可以读取到相应的header信息，这样保证了每个请求的独立性。

3.2 ServeMux的自定义

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。

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。

我们前面讲述conn.server的时候，其实内部是调用了http包默认的路由器，通过路由器把本次请求的信息传递到了后端的处理函数。那么这个路由器是怎么实现的呢？

它的结构如下：

```
type ServeMux struct {
    mu sync.RWMutex   //锁，由于请求涉及到并发处理，因此这里需要一个锁机制
    m  map[string]muxEntry  // 路由规则，一个string对应一个mux实体，这里的string就是注册的路由表达式
    hosts bool // 是否在任意的规则中带有host信息
}
```

下面看一下muxEntry：

```
type muxEntry struct {
    explicit bool   // 是否精确匹配
    h       Handler // 这个路由表达式对应哪个handler
    pattern  string  //匹配字符串
}
```

接着看一下Handler的定义：

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request) // 路由实现器
}
```

Handler是一个接口，但是前面的sayhello函数并没有实现ServeHTTP这个接口，为什么能添加呢？原来在http包里面还定义了一个类型HandlerFunc，我们定义的函数sayhello就是这个HandlerFunc调用之后的结果，这个类型默认就实现了ServeHTTP这个接口，即我们调用了HandlerFunc(f),强制类型转换f成为HandlerFunc类型，这样f就拥有了ServHTTP方法。

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

// ServeHTTP calls f(w, r).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}
```

路由器里面存储好了相应的路由规则之后，那么具体的请求又是怎么分发的呢？

路由器接收到请求之后调用mux.handler(r).ServeHTTP(w, r)

也就是调用对应路由的handler的ServerHTTP接口，那么mux.handler(r)怎么处理的呢？

```
func (mux *ServeMux) handler(r *Request) Handler {
    mux.mu.RLock()
    defer mux.mu.RUnlock()
    // Host-specific pattern takes precedence over generic ones
    h := mux.match(r.Host + r.URL.Path)
    if h == nil {
        h = mux.match(r.URL.Path)
    }

    if h == nil {
```

```
        h = NotFoundHandler()
    }
    return h
}
```

原来他是根据用户请求的URL和路由器里面存储的map去匹配的，当匹配到之后返回存储的handler，调用这个handler的ServHTTP接口就可以执行到相应的函数了。

通过上面这个介绍，我们了解了整个路由过程，Go其实支持外部实现的路由器ListenAndServe的第二个参数就是用以配置外部路由器的，它是一个Handler接口，即外部路由器只要实现了Handler接口就可以,我们可以在自己实现的路由器的ServHTTP里面实现自定义路由功能。

新建go文件(demo02_route.go)，如下代码所示，我们自己实现了一个简易的路由器：

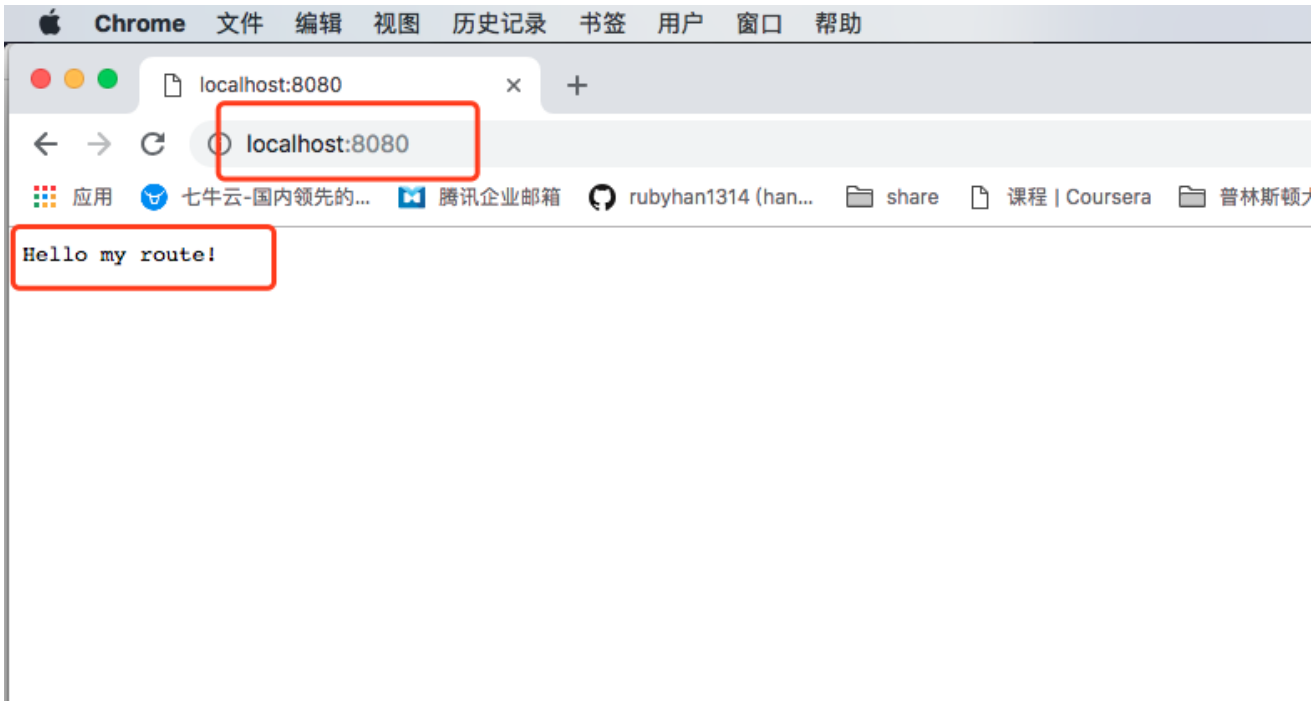
```
package main

import (
    "fmt"
    "net/http"
)

type MyMux struct {
}

func (p *MyMux) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path == "/" {
        sayHello(w, r)
        return
    }
    http.NotFound(w, r)
    return

}
```

```
func sayHello(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello my route!")
}
func main() {
    mux := &MyMux{}
    http.ListenAndServe(":8080", mux)
}
```

然后打开浏览器：输入以下网址：

```
http://localhost:8080/
```

页面显示内容：



Go代码的执行流程

通过对http包的分析之后，现在让我们来梳理一下整个的代码执行过程。

- 首先调用Http.HandleFunc

  按顺序做了几件事：

  1. 调用了DefaultServerMux的HandleFunc

2. 调用了DefaultServerMux的Handle
3. 往DefaultServeMux的map[string]muxEntry中增加对应的handler和路由规则

- 其次调用http.ListenAndServe(":8080", nil)

  按顺序做了几件事情：

  1. 实例化Server
  2. 调用Server的ListenAndServe()
  3. 调用net.Listen("tcp", addr)监听端口
  4. 启动一个for循环，在循环体中Accept请求
  5. 对每个请求实例化一个Conn，并且开启一个goroutine为这个请求进行服务go c.serve()
  6. 读取每个请求的内容w, err := c.readRequest()
  7. 判断handler是否为空，如果没有设置handler（这个例子就没有设置handler），handler就设置为DefaultServeMux
  8. 调用handler的ServeHttp
  9. 在这个例子中，下面就进入到DefaultServerMux.ServeHttp
  10. 根据request选择handler，并且进入到这个handler的ServeHTTP

      mux.handler(r).ServeHTTP(w, r)

  11. 选择handler：

      A. 判断是否有路由能满足这个request（循环遍历ServerMux的muxEntry）

      B. 如果有路由满足，调用这个路由handler的ServeHttp

      C. 如果没有路由满足，调用NotFoundHandler的ServeHttp

根据不同的URL，可以执行不同的handler，新建go文件(demo03_handler.go)，示例代码：

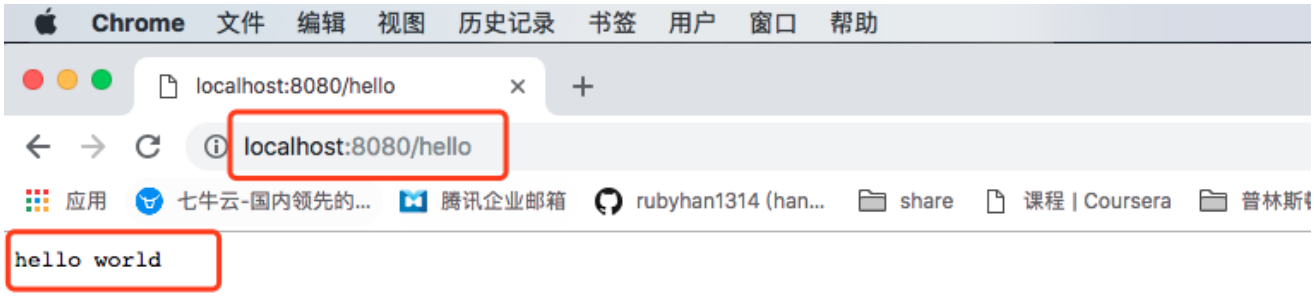
```
package main

import (
```

```
    "net/http"
    "io"
)

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/jack", func(w http.ResponseWriter, r *http.Request) {
        io.WriteString(w, "你好，jack...")
    })
    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(":8080", mux)
}

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

然后打开浏览器：输入以下网址：

```
http://localhost:8080/hello
```

页面显示内容：



然后打开浏览器：输入以下网址：

```
http://localhost:8080/bye
```

页面显示内容：

然后打开浏览器：输入以下网址：

```
http://localhost:8080/baidu
```

页面会自动跳转到百度，以为我们在代码中设置了重定向：

```
http.Redirect(w, r, "http://www.baidu.com", http.StatusTemporaryRedirect)
```

3.3 FileServer

用go来搭建一个文件服务器FileServer也非常的简单，并且我们简单分析一下，它究竟是如何工作的。

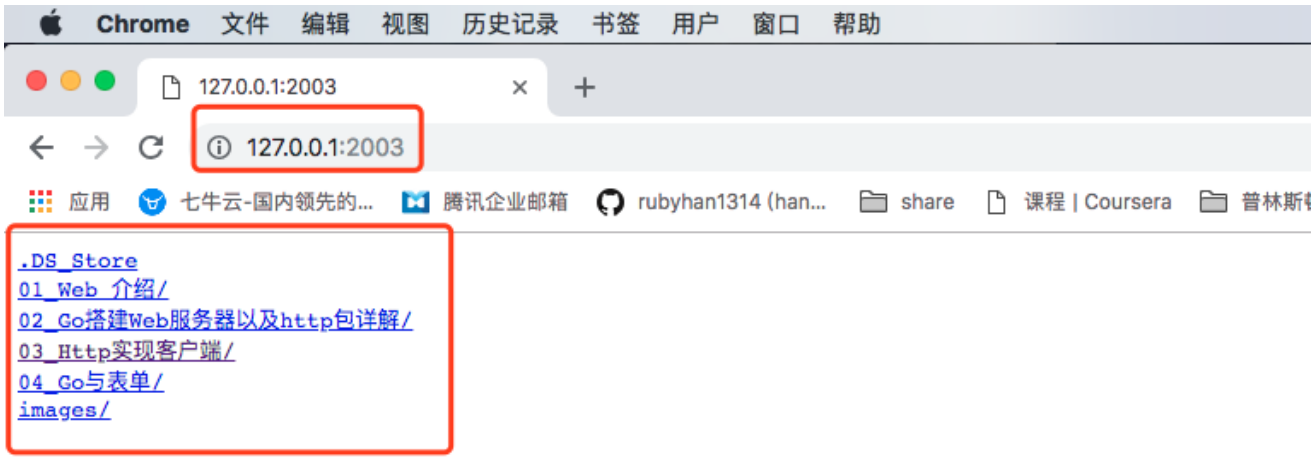
首先搭建一个最简单的文件服务器：

新建go文件(demo04_fileserver.go)，代码如下：

```
package main

import (
    "log"
    "net/http"
)

func main() {
    /**
    FileServer：
```

```
    1.www.xx.com/ 根路径 直接使用

                http.Handle("/", http.FileServer(http.Dir("/tmp")))


        2.www.xx.com/c/ 带有请求路径的 需要添加函数
                    http.Handle("/c/", http.StripPrefix("/c/",
http.FileServer(http.Dir("/tmp"))))
     */
    err := http.ListenAndServe(":2003", http.FileServer(http.Dir("D:/Doc/Golang/网络编程")))
    http.Handle("/", http.StripPrefix("/", http.FileServer(http.Dir("D:/Doc/Golang/网络编程"))))
    if err != nil {
        log.Fatal(err)
    }
}
```

打开浏览器，输入地址，运行如下：



接下来，我们分析以下FileServer。

打开源码，我们定位到net/http/fs.go文件中，看看http.FileServer是如何定义的

```
func FileServer(root FileSystem) Handler {
    return &fileHandler{root}
}
```

原来FileServer函数是返回一个Handler，接下来我们再看看fileHandler是怎么定义的

```
type fileHandler struct {
    root FileSystem
}
```

原来是个结构体，既然是个Handler，那么它一定实现了ServeHttp函数，找找看

```
func (f *fileHandler) ServeHTTP(w ResponseWriter, r *Request) {
    upath := r.URL.Path
    if !strings.HasPrefix(upath, "/") {
        upath = "/" + upath
        r.URL.Path = upath
    }
    serveFile(w, r, f.root, path.Clean(upath), true) //看来关键在这里
}
```

进入到关键函数serveFile看看，它的函数声明如下：

```
func serveFile(w ResponseWriter, r *Request, fs FileSystem, name string, redirect bool) //最后一个参数表示是否重新定向，在web服务中，它总是true
```

这里最后一个参数很重要，我们下面会揭示为什么，好啦，看看源码，无关部分我都砍掉：

```
func serveFile(w ResponseWriter, r *Request, fs FileSystem, name string, redirect bool) {
    const indexPage = "/index.html"

    // redirect .../index.html to .../
    // can't use Redirect() because that would make the path absolute,
    // which would be a problem running under StripPrefix
```

```
    if strings.HasSuffix(r.URL.Path, indexPage) {
        localRedirect(w, r, "./")
        return
    }

    f, err := fs.Open(name)
    if err != nil {
        msg, code := toHTTPError(err)
        Error(w, msg, code)
        return
    }
    defer f.Close()

    d, err := f.Stat()
    if err != nil {
        msg, code := toHTTPError(err)
        Error(w, msg, code)
        return
    }

    if redirect {
        // redirect to canonical path: / at end of directory url
        // r.URL.Path always begins with /
        url := r.URL.Path
        if d.IsDir() {
            if url[len(url)-1] != '/' {
                localRedirect(w, r, path.Base(url)+"/")    ---------------------------- 1
                return
            }
        } else {
            if url[len(url)-1] == '/' {
                localRedirect(w, r, "../"+path.Base(url))  ---------------------------- 2
                return
            }
        }
    }


    // serveContent will check modification time
```

```
    sizeFunc := func() (int64, error) { return d.Size(), nil }
    serveContent(w, r, d.Name(), d.ModTime(), sizeFunc, f)  ---------------------------- 3
}
```

重点看到标注部分，现在我们假设我们请求是http://127.0.0.1/abc/d.jpg。那么我们 r.URL.Path的值就是/abc/d.jpg，于是乎，程序进入到标注1部分，path.Base()函数是取函数最后/部分，也就是/d.jpg。现在请求变成了/d.jpg，然后进行重定向，这时浏览器根据重定向内容再次发送请求，这次请求的url.Path是我们上一次处理好的/d.jpg，最后，程序便顺利的进入到了标注3部分。serveContent 这个函数是最终向浏览器发送资源文件的。

大概的一个处理文件资源请求的流程就是这样子，现在我们来解释一下，为什么serveFile()函数的第四个参数那么重要：

```
func serveFile(w ResponseWriter, r *Request, fs FileSystem, name string, redirect bool)
```

因为在web服务中，我们发现它永远都是true，这就导致了我们的url无论是什么，都将会被它cut成只剩最后一部分/xxx.jpg类似的样子。换句话说，假设我们为文本服务器设置的路由格式是/xxx/xxx/xxx/x.jpg的话。 那么文本服务器根本没法正常工作，因为它只认识/xx.jpg的路由格式。

在net/http/server.go文件中，有这么一个函数：

```
// StripPrefix returns a handler that serves HTTP requests
// by removing the given prefix from the request URL's Path
// and invoking the handler h. StripPrefix handles a
// request for a path that doesn't begin with prefix by
// replying with an HTTP 404 not found error.
func StripPrefix(prefix string, h Handler) Handler {
    if prefix == "" {
        return h
    }

    return HandlerFunc(func(w ResponseWriter, r *Request) {
```

```
        if p := strings.TrimPrefix(r.URL.Path, prefix); len(p) < len(r.URL.Path) {
            r.URL.Path = p
            h.ServeHTTP(w, r)
        } else {
            NotFound(w, r)
        }
    })
}
```

根据注释以及代码来看，它的作用是返回一个Handler，但是这个Handler呢，有点点不一样，不一样在哪里呢，它会过滤掉一部分路由前缀。

比如我们有如下路由：/aaa/bbb/ccc.jpg，那么执行StripPrefix("/aaa/bbb"，..handler)之后，我们将会得到一个新的Handler，这个新Handler的执行函数和原来的handler是一样的，但是这个新Handler在处理路由请求的时候，会自动将/aaa/bbb/ccc.jpg理解为/aaa.jpg

四、总结

我们介绍了HTTP协议, DNS解析的过程, 如何用go实现一个简陋的web server。并深入到net/http包的源码中

为大家揭开实现此server的秘密。

希望通过这一章的学习，你能够对Go开发Web有了初步的了解，我们也看到相应的代码了，Go开发Web应用是很方便的，同时又是相当的灵活。