# 02 TCP编程

[02 TCP编程 1](#_Toc153475742)

[1. 基本介绍 1](#_Toc153475743)

[2. net/http源码分析 5](#_Toc153475744)

[前言 5](#_Toc153475745)

[HTTP 服务处理流程 5](#_Toc153475746)

[路由注册 7](#_Toc153475747)

[Handler 8](#_Toc153475748)

[ServeMux (服务复用器) 10](#_Toc153475749)

[注册路由 11](#_Toc153475750)

[自定义 ServeMux 12](#_Toc153475751)

[启动服务 14](#_Toc153475752)

[Server（服务器对象） 15](#_Toc153475753)

[处理连接 17](#_Toc153475754)

[停止服务 22](#_Toc153475755)

# 基本介绍

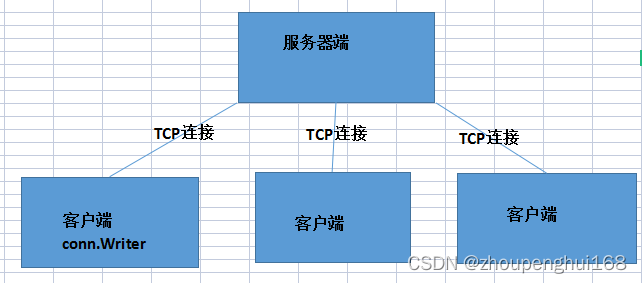
Golang 的主要设计目标之一就是面向大规模后端服务程序，网络通信这块是服务端程序必不可少也是至关重要的一部分.

网络编程有两种户

(1).TCP Socket 编程，是网络编程的主流。之所以叫TCP Socket 编程，是因为底层是基于 TCP/IP 协议的

(2).B/S结构的 http编程，我们使用浏览器去访问服务器时，使用的就是 http协议，而 http底层依旧是用TCP Socket 实现的, 比如：京东商城 【 这属于 go web 开发范畴 】

TCP Socket编程的客户端和服务器端：



(1).服务端的处理流程

1).监听端口8888

2).接收客户端的tcp链接，建立客户端和服务器端的链接

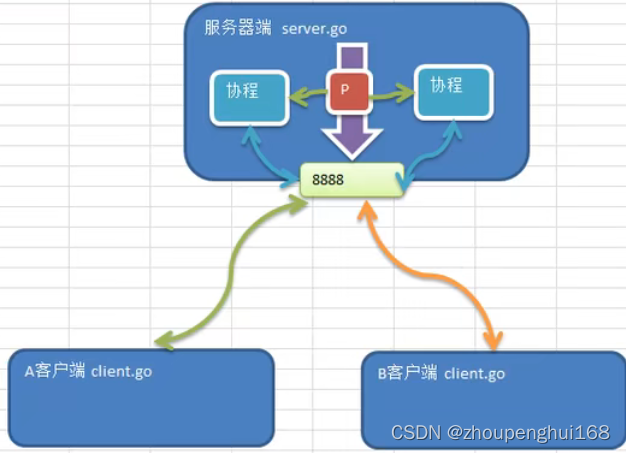
3).创建goroutine，处理该链接的请求（通常客户端会通过链接发送请求包）

(2).客户端的处理流程

1).建立与服务端的链接

2).发送请求数据[终端]，接收服务器端返回的结果数据

3).关闭链接



入门案例：

服务端功能:

编写一个服务器端程序,在8888端口监听

可以和多个客户端创建链接

链接成功后，客户端可以发送数据，服务器端接受数据，并显示在终端上

先使用telnet 来测试，然后编写客户端程序来测试

服务端的代码如下:

server.go

package main  
  
import (  
 "fmt"  
 "net" //做网络socket开发时,net包含需要的所有方法和函数  
 //"io"  
)  
  
func process(conn net.Conn) {  
 //循环接收客户端发送的消息  
 defer conn.Close() //延时关闭conn  
 for {  
 //创建一个新的切片  
 buf := make([]byte, 1024)  
 fmt.Printf("服务器等待客户端%s发送信息\n", conn.RemoteAddr().String())  
 // conn.Read(buf)  
 //1.等待客户端通过conn发送信息  
 //2.如果客户端没有Write[发送],那么协程将会阻塞在这里  
 n, err := conn.Read(buf) // 从conn读取  
 if err != nil {  
 fmt.Printf("服务器Read err = %v\n", err)  
 return  
 }  
 // if err == io.EOF {  
 // fmt.Println("远程客户端已退出")  
 // return  
 // }  
 //3.显示客户端发送的内容到服务器的终端  
 fmt.Print(string(buf[:n]))  
 }  
}  
func main() {  
 fmt.Println("服务器开始监听...")  
 //"tcp": 表示使用的网络协议是tcp  
 //"0.0.0.0:8888": 表示在本地进行监听端口8888  
 //127.0.0.1 => ipv4, 0.0.0.0 => ipv4/ipv6  
 listen, err := net.Listen("tcp", "0.0.0.0:8888")  
 if err != nil {  
 fmt.Printf("listen err, err = %v\n", err)  
 return  
 }  
  
 fmt.Printf("listen success, success = %v\n", listen)  
  
 defer listen.Close() // 延时关闭listen  
 //循环等待客户端来连接  
 for {  
 //等待客户端连接  
 fmt.Println("等待客户端连接...")  
 conn, err := listen.Accept()  
 if err != nil {  
 fmt.Printf("listen accept err, err=%v\n", err)  
 } else {  
 fmt.Printf("listen success, success=%v, client addr = %v\n", conn, conn.RemoteAddr().String())  
 }  
 //准备一个协程,为客户端服务  
 go process(conn)  
 }  
  
}

客户端功能：

1.编写一个客户端端程序，能链接到服务器端的8888端口

2.客户端可以发送单行数据，然后就退出

3.能通过终端输入数据（输入一行发送一行），并发送给服务器端口

4.在终端输入exit，表示退出程序

package main  
  
import (  
 "bufio"  
 "fmt"  
 "net" //做网络socket开发时,net包含需要的所有方法和函数  
 "os"  
 "strings"  
)  
  
func main() {  
 conn, err := net.Dial("tcp", "127.0.0.1:8888")  
 if err != nil {  
 fmt.Printf("dial err, err = %v\n", err)  
 return  
 }  
 fmt.Printf("dial success, success = %v\n", conn)  
 //功能一: 客户端发送单行数据,然后退出  
 reader := bufio.NewReader(os.Stdin) // os.Stdin 表示标准输入[终端]  
  
 for {  
 //从终端读取一行用户输入,并发送给服务器  
 line, err := reader.ReadString('\n')  
 if err != nil {  
 fmt.Printf("readString err = %v\n", err)  
 }  
 //如果输入的是exit,则退出  
 line = strings.Trim(line, "\r\n")  
 if line == "exit" {  
 fmt.Println("客户端退出")  
 break  
 }  
 //将line发送给服务器  
 n, err := conn.Write([]byte(line + "\n"))  
 if err != nil {  
 fmt.Printf("conn err =%v\n", err)  
 }  
 fmt.Printf("客户端发送了%d字节给服务器,并退出", n)  
 }  
}

# net/http源码分析

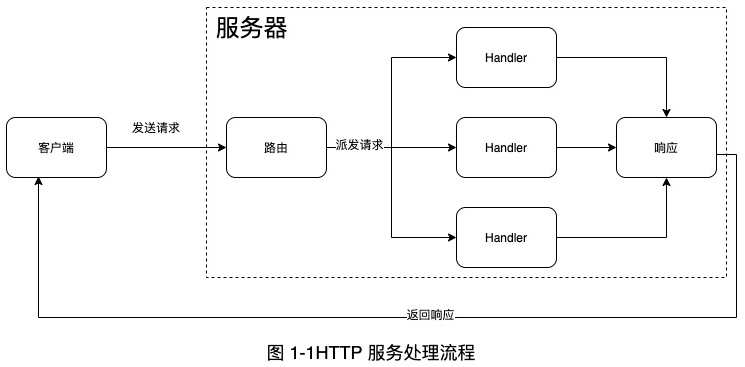
## 前言

Go 是一门通用的编程语言，想要学习 Go 语言的 Web 开发，就必须知道如何用 Go 启动一个 HTTP 服务器用于接收和响应来自客户端的 HTTP 请求。用 Go 实现一个 http server 非常容易，Go 语言标准库 net/http 自带了一系列结构和方法来帮助开发者简化 HTTP 服务开发的相关流程。因此，我们不需要依赖任何第三方组件就能构建并启动一个高并发的 HTTP 服务器。这篇文章会学习如何用 net/http 自己编写实现一个 HTTP Server 并探究其实现原理，以此来学习了解网络编程的常见范式以及设计思路。

## HTTP 服务处理流程

基于 HTTP 构建的服务标准模型包括两个端，客户端 (Client) 和服务端 (Server)。HTTP 请求从客户端发出，服务端接受到请求后进行处理然后将响应返回给客户端。所以 http 服务器的工作就在于如何接受来自客户端的请求，并向客户端返回响应。

典型的 HTTP 服务的处理流程如下图所示：



服务器在接收到请求时，首先会进入路由 (router)，也称为服务复用器（Multiplexe），路由的工作在于让请求找到对应的处理器 (handler)，处理器对接收到的请求进行相应处理后构建响应并返回给客户端。Go 实现的 http server 同样遵循这样的处理流程。

我们先看看 Go 如何实现一个简单的返回 "Hello World" 的 http server：

package main  
  
import (  
 "fmt"  
 "net/http"  
)  
  
func HelloHandler(w http.ResponseWriter, r \*http.Request) {  
 fmt.Fprintf(w, "Hello World")  
}  
  
func main () {  
 http.HandleFunc("/", HelloHandler)  
 http.ListenAndServe(":8000", nil)  
}

运行代码之后，在浏览器中打开 localhost:8000 就可以看到 Hello World。这段代码先利用 http.HandleFunc 在根路由 / 上注册了一个 HelloHandler, 然后利用 http.ListenAndServe 启动服务器并监听本地的 8000 端口。当有请求过来时，则根据路由执行对应的 handler 函数。

注意：http.ListenAndServe(":8000", nil) 的第一个参数本来应该是 ip: 端口号 的形式，但是这里省略了ip，那么默认为 0.0.0.0。

我们再看一下另外一种常见的实现方式：

package main  
  
import (  
 "fmt"  
 "net/http"  
)  
  
type HelloHandlerStruct struct {  
 content string  
}  
  
func (handler \*HelloHandlerStruct) ServeHTTP(w http.ResponseWriter, r \*http.Request) {  
 fmt.Fprintf(w, handler.content)  
}  
  
func main() {  
 http.Handle("/", &HelloHandlerStruct{content: "Hello World"})  
 http.ListenAndServe(":8000", nil)  
}

这段代码不再使用 http.HandleFunc 函数，取而代之的是直接调用 http.Handle 并传入我们自定义的 http.Handler 接口的实例。

Go 实现的 http 服务步骤非常简单，首先注册路由，然后创建服务并开启监听即可。下文我们将从注册路由、开启服务、处理请求，以及关闭服务这几个步骤了解 Go 如何实现 http 服务。

## 路由注册

http.HandleFunc 和 http.Handle 都是用于给路由规则指定处理器，http.HandleFunc 的第一个参数为路由的匹配规则 (pattern) 第二个参数是一个签名为 func(w http.ResponseWriter, r \*http.Requests) 的函数。而 http.Handle 的第二个参数为实现了 http.Handler 接口的类型的实例。

http.HandleFunc 和 http.Handle 的源码如下：

func HandleFunc(pattern string, handler func(ResponseWriter, \*Request)) {  
 DefaultServeMux.HandleFunc(pattern, handler)  
}

func (mux \*ServeMux) HandleFunc(pattern string, handler func(ResponseWriter, \*Request)) {  
 if handler == nil {  
 panic("http: nil handler")  
 }  
 mux.Handle(pattern, HandlerFunc(handler))  
}

func Handle(pattern string, handler Handler) { DefaultServeMux.Handle(pattern, handler) }

可以看到这两个函数最终都由 DefaultServeMux 调用 Handle 方法来完成路由处理器的注册。  
这里我们遇到两种类型的对象：ServeMux 和 Handler。

### Handler

http.Handler 是 net/http 中定义的接口用来表示 HTTP 请求：

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

Handler 接口中声明了名为 ServeHTTP 的函数签名，也就是说任何结构或自定义类型只要实现了这个 ServeHTTP 方法，那么这个结构体或自定义类型就是一个 Handler 对象。其实 go 的 http 服务都是基于 Handler 进行处理，而 Handler 对象的 ServeHTTP 方法会读取 Request 进行逻辑处理然后向 ResponseWriter 中写入响应的头部信息和响应内容。

回到上面的 HandleFunc 函数，它调用了 \*ServeMux.HandleFunc 将处理器注册到指定路由规则上：

// HandleFunc registers the handler function for the given pattern.  
func (mux \*ServeMux) HandleFunc(pattern string, handler func(ResponseWriter, \*Request)) {  
 if handler == nil {  
 panic("http: nil handler")  
 }  
 mux.Handle(pattern, HandlerFunc(handler))  
}

注意一下这行代码：

mux.Handle(pattern, HandlerFunc(handler))

这里 HandlerFunc 实际上是将 handler 函数做了一个类型转换，将函数转换为了 http.HandlerFunc 类型（注意：注册路由时调用的是 http.HandleFunc，这里类型是 http.HandlerFunc）。看一下 HandlerFunc 的定义：

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

HandlerFunc 类型表示的是一个具有 func(ResponseWriter, \*Request) 签名的函数类型，并且这种类型实现了 ServeHTTP 方法（在其实现的 ServeHTTP 方法中又调用了被转换的函数自身）。也就是说这个类型的函数其实就是一个 Handler 接口类型的对象。利用这种类型转换，我们可以将将具有 func(ResponseWriter, \*Request) 签名的普通函数转换为一个 Handler 对象，而不需要定义一个结构体，再让这个结构实现 ServeHTTP 方法。

### ServeMux (服务复用器)

上面的代码中可以看到不论是使用 http.HandleFunc 还是 http.Handle 注册路由的处理函数时最后都会用到 ServerMux 结构的 Handle 方法去注册路由处理函数。

我们先来看一下 ServeMux 的定义

type ServeMux struct {  
 mu sync.RWMutex  
 m map[string]muxEntry  
 es []muxEntry // slice of entries sorted from longest to shortest.  
 hosts bool // whether any patterns contain hostnames  
}  
  
type muxEntry struct {  
 h Handler  
 pattern string  
}

ServeMux 中的字段 m，是一个 map，key 是路由表达式，value 是一个 muxEntry 结构体，muxEntry 结构体存储了路由表达式和对应的 handler。字段 m 对应的 map 用于路由的精确匹配而 es 字段的 slice 会用于路由的部分匹配，这个到了路由匹配部分再细讲。

ServeMux 也实现了 ServeHTTP 方法：

func (mux \*ServeMux) ServeHTTP(w ResponseWriter, r \*Request) {  
 if r.RequestURI == "\*" {  
 if r.ProtoAtLeast(1, 1) {  
 w.Header().Set("Connection", "close")  
 }  
 w.WriteHeader(*StatusBadRequest*)  
 return  
 }  
 h, \_ := mux.Handler(r)  
 h.ServeHTTP(w, r)  
}

也就是说 ServeMux 结构体也是 Handler 对象，只不过 ServeMux 的 ServeHTTP 方法不是用来处理具体的 request 和构建 response，而是用来通过路由查找对应的路由处理器 Handler 对象，再去调用路由处理器的 ServeHTTP 方法去处理 request 和构建 reponse。

### 注册路由

搞明白 Handler 和 ServeMux 之后，我们再回到之前的代码：

DefaultServeMux.Handle(pattern, handler) }

var DefaultServeMux = &defaultServeMux  
var defaultServeMux ServeMux

这里的 DefaultServeMux 表示一个默认的 ServeMux 实例，在上面的例子中我们没有创建自定义的 ServeMux，所以会自动使用 DefaultServeMux

然后再看一下 ServeMux 的 Handle 方法是怎么注册路由的处理函数的：

func (mux \*ServeMux) Handle(pattern string, handler Handler) {  
 mux.mu.Lock()  
 defer mux.mu.Unlock()  
  
 if pattern == "" {  
 panic("http: invalid pattern")  
 }  
 if handler == nil {  
 panic("http: nil handler")  
 }

// 路由已经注册过处理器函数，直接panic  
 if \_, exist := mux.m[pattern]; exist {  
 panic("http: multiple registrations for " + pattern)  
 }  
  
 if mux.m == nil {  
 mux.m = make(map[string]muxEntry)  
 }

// 用路由的pattern和处理器函数创建muxEntry对象  
 e := muxEntry{h: handler, pattern: pattern}

// 向ServeMux的m字段增加新的路有匹配规则  
 mux.m[pattern] = e

// 如果路由pattern以’/’结尾，则将对应的muxEntry对象加入到[]muxEntry中，路由pattern长的位于es的的前面  
 if pattern[len(pattern)-1] == '/' {  
 mux.es = appendSorted(mux.es, e)  
 }  
  
 if pattern[0] != '/' {  
 mux.hosts = true  
 }  
}

func appendSorted(es []muxEntry, e muxEntry) []muxEntry {  
 n := len(es)  
 i := sort.Search(n, func(i int) bool {  
 return len(es[i].pattern) < len(e.pattern)  
 })  
 if i == n {  
 return append(es, e)  
 }  
 // we now know that i points at where we want to insert  
 es = append(es, muxEntry{}) // try to grow the slice in place, any entry works.  
 copy(es[i+1:], es[i:]) // Move shorter entries down  
 es[i] = e  
 return es  
}

Handle 方法注册路由时主要做了两件事情：一个就是向 ServeMux 的 map[string]muxEntry 增加给定的路由匹配规则；然后如果路由表达式以'/' 结尾，则将对应的 muxEntry 对象加入到 []muxEntry 中，按照路由pattern的长度倒序排列，pattern长的位于前面。前者用于路由精确匹配，后者用于部分匹配，具体怎么匹配的后面再看。

### 自定义 ServeMux

通过 http.NewServeMux() 可以创建一个 ServeMux 实例取代默认的 DefaultServeMux

我们把上面输出 Hello World 的 http server 再次改造一下，使用自定义的 ServeMux 实例作为 ListenAndServe() 方法的第二个参数，并且增加一个 /welcome 路由（下面的代码主要是展示用 Handle 和 HandleFunc 注册路由，实际使用的时候不必这么麻烦，选一种就好）：

//package main  
//  
//import (  
// "fmt"  
// "net/http"  
//)  
//  
//func HelloHandler(w http.ResponseWriter, r \*http.Request) {  
// fmt.Fprintf(w, "Hello World")  
//}  
//  
//func main () {  
// http.HandleFunc("/index/", HelloHandler)  
// http.HandleFunc("/", HelloHandler)  
// http.ListenAndServe(":8000", nil)  
//}  
  
package main  
  
import (  
 "fmt"  
 "net/http"  
)  
  
type WelcomeHandlerStruct struct {  
  
}  
  
func HelloHandler(w http.ResponseWriter, r \*http.Request) {  
 fmt.Fprintf(w, "Hello World")  
}  
  
func (\*WelcomeHandlerStruct) ServeHTTP(w http.ResponseWriter, r \*http.Request) {  
 fmt.Fprintf(w, "Welcome")  
}  
  
func main () {  
 mux := http.NewServeMux()  
 mux.HandleFunc("/", HelloHandler)  
 mux.Handle("/welcome", &WelcomeHandlerStruct{})  
 http.ListenAndServe(":8080", mux)  
}

之前提到 ServeMux 也实现了 ServeHTTP 方法，因此 mux 也是一个 Handler 对象。对于 ListenAndServe() 方法，如果第二个参数是自定义 ServeMux 实例，那么 Server 实例接收到的 ServeMux 服务复用器对象将不再是 DefaultServeMux 而是 mux。

## 启动服务

路由注册完成后，使用 http.ListenAndServe 方法就能启动服务器开始监听指定端口过来的请求。

func ListenAndServe(addr string, handler Handler) error {  
 server := &Server{Addr: addr, Handler: handler}  
 return server.ListenAndServe()  
}

func (srv \*Server) ListenAndServe() error {  
 if srv.shuttingDown() {  
 return ErrServerClosed  
 }  
 addr := srv.Addr  
 if addr == "" {  
 addr = ":http"  
 }  
 ln, err := net.Listen("tcp", addr)  
 if err != nil {  
 return err  
 }  
 return srv.Serve(ln)  
}

这先创建了一个 Server 对象，传入了地址和 handler 参数（这里的 handler 参数时 ServeMux 实例），然后调用 Server 对象 ListenAndServe() 方法。

### Server（服务器对象）

先看一下 Server 这个结构体的定义，字段比较多，可以先大致了解一下：

type Server struct {  
 Addr string  
  
 Handler Handler

TLSConfig \*tls.Config  
 ReadTimeout time.Duration  
 ReadHeaderTimeout time.Duration  
 WriteTimeout time.Duration  
 IdleTimeout time.Duration  
 MaxHeaderBytes int  
 TLSNextProto map[string]func(\*Server, \*tls.Conn, Handler)  
 ConnState func(net.Conn, ConnState)  
 ErrorLog \*log.Logger  
 BaseContext func(net.Listener) context.Context  
 ConnContext func(ctx context.Context, c net.Conn) context.Context  
 inShutdown atomicBool // true when when server is in shutdown  
 disableKeepAlives int32 // accessed atomically.  
 nextProtoOnce sync.Once // guards setupHTTP2\_\* init  
 nextProtoErr error // result of http2.ConfigureServer if used  
 mu sync.Mutex  
 listeners map[\*net.Listener]struct{}  
 activeConn map[\*conn]struct{}  
 doneChan chan struct{}  
 onShutdown []func()  
}

在 Server 的 ListenAndServe 方法中，会初始化监听地址 Addr，同时调用 Listen 方法设置监听。最后将监听的 TCP 对象传入其 Serve 方法。Server 对象的 Serve 方法会接收 Listener 中过来的连接，为每个连接创建一个 goroutine，在 goroutine 中会用路由处理 Handler 对请求进行处理并构建响应。

func (srv \*Server) ListenAndServe() error {  
 if srv.shuttingDown() {  
 return ErrServerClosed  
 }  
 addr := srv.Addr  
 if addr == "" {  
 addr = ":http"  
 }  
 ln, err := net.Listen("tcp", addr) // 监听  
 if err != nil {  
 return err  
 }  
 return srv.Serve(ln)  
}

func (srv \*Server) Serve(l net.Listener) error {

......

baseCtx := context.Background() // base is always background, per Issue 16220

ctx := context.WithValue(baseCtx, ServerContextKey, srv)

for {

rw, e := l.Accept()// 接收 listener 过来的网络连接请求

......

c := srv.newConn(rw)

c.setState(c.rwc, StateNew) // 将连接放在 Server.activeConn这个 map 中

go c.serve(ctx)// 创建协程处理请求

}

}

这里隐去了一些细节，以便了解 Serve 方法的主要逻辑。首先创建一个上下文对象，然后调用 Listener 的 Accept() 接收监听到的网络连接；一旦有新的连接建立，则调用 Server 的 newConn() 创建新的连接对象，并将连接的状态标志为 StateNew，然后开启一个 goroutine 处理连接请求。

## 处理连接

在开启的 goroutine 中 conn 的 serve() 会进行路由匹配找到路由处理函数然后调用处理函数。这个方法很长，我们保留关键逻辑。

func (c \*conn) serve(ctx context.Context) {

...

for {

w, err := c.readRequest(ctx)

if c.r.remain != c.server.initialReadLimitSize() {

// If we read any bytes off the wire, we're active.

c.setState(c.rwc, StateActive)

}

...

serverHandler{c.server}.ServeHTTP(w, w.req)

w.cancelCtx()

if c.hijacked() {

return

}

w.finishRequest()

if !w.shouldReuseConnection() {

if w.requestBodyLimitHit || w.closedRequestBodyEarly() {

c.closeWriteAndWait()

}

return

}

c.setState(c.rwc, StateIdle)

c.curReq.Store((\*response)(nil))

...

}

}

当一个连接建立之后，该连接中所有的请求都将在这个协程中进行处理，直到连接被关闭。在 serve() 方法中会循环调用 readRequest() 方法读取下一个请求进行处理，其中最关键的逻辑是下面行代码：

serverHandler{c.server}.ServeHTTP(w, w.req)

serverHandler 是一个结构体类型，它会代理 Server 对象：

type serverHandler struct {  
 srv \*Server  
}  
  
func (sh serverHandler) ServeHTTP(rw ResponseWriter, req \*Request) {  
 handler := sh.srv.Handler  
 if handler == nil {  
 handler = DefaultServeMux  
 }  
 if req.RequestURI == "\*" && req.Method == "OPTIONS" {  
 handler = globalOptionsHandler{}  
 }  
 handler.ServeHTTP(rw, req)  
}

在 serverHandler 实现的 ServeHTTP() 方法里的 sh.srv.Handler 就是我们最初在 http.ListenAndServe() 中传入的 Handler 参数，也就是我们自定义的 ServeMux 对象。如果该 Handler 对象为 nil，则会使用默认的 DefaultServeMux。最后调用 ServeMux 的 ServeHTTP() 方法匹配当前路由对应的 handler 方法。

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

if r.RequestURI == "\*" {

if r.ProtoAtLeast(1, 1) {

w.Header().Set("Connection", "close")

}

w.WriteHeader(StatusBadRequest)

return

}

h, \_ := mux.Handler(r)

h.ServeHTTP(w, r)

}

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

if r.Method == "CONNECT" {

if u, ok := mux.redirectToPathSlash(r.URL.Host, r.URL.Path, r.URL); ok {

return RedirectHandler(u.String(), StatusMovedPermanently), u.Path

}

return mux.handler(r.Host, r.URL.Path)

}

// All other requests have any port stripped and path cleaned

// before passing to mux.handler.

host := stripHostPort(r.Host)

path := cleanPath(r.URL.Path)

// If the given path is /tree and its handler is not registered,

// redirect for /tree/.

if u, ok := mux.redirectToPathSlash(host, path, r.URL); ok {

return RedirectHandler(u.String(), StatusMovedPermanently), u.Path

}

if path != r.URL.Path {

\_, pattern = mux.handler(host, path)

url := \*r.URL

url.Path = path

return RedirectHandler(url.String(), StatusMovedPermanently), pattern

}

return mux.handler(host, r.URL.Path)

}

// handler is the main implementation of Handler.

// The path is known to be in canonical form, except for CONNECT methods.

func (mux \*ServeMux) handler(host, path string) (h Handler, pattern string) {

mux.mu.RLock()

defer mux.mu.RUnlock()

// Host-specific pattern takes precedence over generic ones

if mux.hosts {

h, pattern = mux.match(host + path)

}

if h == nil {

h, pattern = mux.match(path)

}

if h == nil {

h, pattern = NotFoundHandler(), ""

}

return

}

// Find a handler on a handler map given a path string.

// Most-specific (longest) pattern wins.

func (mux \*ServeMux) match(path string) (h Handler, pattern string) {

// Check for exact match first.

v, ok := mux.m[path]

if ok {

return v.h, v.pattern

}

// Check for longest valid match. mux.es contains all patterns

// that end in / sorted from longest to shortest.

for \_, e := range mux.es {

if strings.HasPrefix(path, e.pattern) {

return e.h, e.pattern

}

}

return nil, ""

}

在 match 方法里我们看到之前提到的 mux 的 m 字段 (类型为 map[string]muxEntry) 和 es(类型为 []muxEntry)。这个方法里首先会进行精确匹配，在 map[string]muxEntry 中查找是否有对应的路由规则存在；如果没有匹配的路由规则，则会利用 es 进行近似匹配。

之前提到在注册路由时会把以'/' 结尾的路由（可称为节点路由）加入到 es 字段的 []muxEntry 中。对于类似 /path1/path2/path3 这样的路由，如果不能找到精确匹配的路由规则，那么则会去匹配和当前路由最接近的已注册的父节点路由，所以如果路由 /path1/path2/ 已注册，那么该路由会被匹配，否则继续匹配下一个父节点路由，直到根路由 /。

由于 []muxEntry 中的 muxEntry 按照路由表达式从长到短排序，所以进行近似匹配时匹配到的节点路由一定是已注册父节点路由中最相近的。

查找到路由实际的处理器 Handler 对象返回给调用者 ServerMux.ServeHTTP 方法后，最后在方法里就会调用处理器 Handler 的 ServeHTTP 方法处理请求、构建写入响应：

h.ServeHTTP(w, r)

实际上如果根据路由查找不到处理器 Handler 那么也会返回 NotFoundHandler:

func NotFound(w ResponseWriter, r \*Request) { Error(w, "404 page not found", StatusNotFound) }

func NotFoundHandler() Handler { return HandlerFunc(NotFound) }

这样标准统一，在调用 h.ServeHTTP(w, r) 后则会想响应中写入 404 的错误信息。

## 停止服务

我们写的 http server 已经能监听网络连接、把请求路由到处理器函数处理请求并返回响应了，但是还需要能优雅的关停服务，在生产环境中，当需要更新服务端程序时需要重启服务，但此时可能有一部分请求进行到一半，如果强行中断这些请求可能会导致意外的结果。

从 Go 1.8 版本开始，net/http 原生支持使用 http.ShutDown 来优雅的关停 HTTP 服务。这种方案同样要求用户创建自定义的 http.Server 对象，因为 Shutdown 方法无法通过其它途径调用。

我们来看下面的代码，这段代码通过结合捕捉系统信号（Signal）、goroutine 和管道（Channel）来实现服务器的优雅停止：

package main  
  
import (  
 "context"  
 "fmt"  
 "log"  
 "net/http"  
 "os"  
 "os/signal"  
 "syscall"  
)  
  
func main() {  
 mux := http.NewServeMux()  
 mux.Handle("/", &helloHandler{})  
  
 server := &http.Server{  
 Addr: ":8081",  
 Handler: mux,  
 }  
  
 // 创建系统信号接收器  
 done := make(chan os.Signal)  
 signal.Notify(done, os.Interrupt, syscall.*SIGINT*, syscall.*SIGTERM*)  
 go func() {  
 <-done  
  
 if err := server.Shutdown(context.Background()); err != nil {  
 log.Fatal("Shutdown server:", err)  
 }  
 }()  
  
 log.Println("Starting HTTP server...")  
 err := server.ListenAndServe()  
 if err != nil {  
 if err == http.ErrServerClosed {  
 log.Print("Server closed under request")  
 } else {  
 log.Fatal("Server closed unexpected")  
 }  
 }  
}  
  
type helloHandler struct{}  
  
func (\*helloHandler) ServeHTTP(w http.ResponseWriter, r \*http.Request) {  
 fmt.Fprintf(w, "Hello World")  
}

这段代码通过捕捉 os.Interrupt 信号（Ctrl+C）和 syscall,SIGTERM 信号（kill 进程时传递给进程的信号）然后调用 server.Shutdown 方法告知服务器应停止接受新的请求并在处理完当前已接受的请求后关闭服务器。为了与普通错误相区别，标准库提供了一个特定的错误类型 http.ErrServerClosed，我们可以在代码中通过判断是否为该错误类型来确定服务器是正常关闭的还是意外关闭的。

用 Go 编写 http server 的流程就大致学习完了，当然要写出一个高性能的服务器还有很多要学习的地方，net/http 标准库里还有很多结构和方法来完善 http server，学会这些最基本的方法后再看其他 Web 框架的代码时就清晰很多。甚至熟练了觉得框架用着太复杂也能自己封装一个 HTTP 服务的脚手架。