Golang Http Server源码阅读

yjf512 · 2014-11-05 09:42:19 · 1943 次点击 · 预计阅读时间 7 分钟 · 大约12小时之前 开始浏览

这是一个创建于 2014-11-05 09:42:19 的文章，其中的信息可能已经有所发展或是发生改变。

这篇文章出现的理由是业务上需要创建一个Web Server。创建web是所有语言出现必须实现的功能之一了。在nginx+fastcgi+php广为使用的今天，这里我们不妨使用Go来进行web服务器的搭建。

前言

使用Go搭建Web服务器的包有很多，大致有下面几种方法，直接使用net包，使用net.http包，使用第三方包（比如gorilla）。使用net包就需要从tcp层开始封装，耗费人力物力极大，果断舍弃。直接使用封装好的net.http和第三方包才是上策。这里我们就选择了使用官方提供的net.http包来搭建web服务。另外附带一句，gorilla的第三方包现在使用还是非常广的，文档也是比较全的，有兴趣的同学可以考虑使用一下。

建议看这篇文章前先看一下net/http文档 **http://golang.org/pkg/net/http/**

net.http包里面有很多文件，都是和http协议相关的，比如设置cookie，header等。其中最重要的一个文件就是server.go了，这里我们阅读的就是这个文件。

几个重要概念

ResponseWriter： 生成Response的接口

Handler： 处理请求和生成返回的接口

ServeMux： 路由，后面会说到ServeMux也是一种Handler

Conn : 网络连接

具体分析

（具体的说明直接以注释形式放在代码中）

几个接口：

Handler

```
type Handler interface {
        ServeHTTP(ResponseWriter, *Request)  // 具体的逻辑函数
}
```

实现了handler接口的对象就意味着往server端添加了处理请求的逻辑。

下面是三个接口(ResponseWriter, Flusher, Hijacker):

ResponseWriter, Flusher, Hijacker

```
// ResponseWriter的作用是被Handler调用来组装返回的Response的
type ResponseWriter interface {
        // 这个方法返回Response返回的Header供读写
        Header() Header

        // 这个方法写Response的Body
        Write([]byte) (int, error)

        // 这个方法根据HTTP State Code来写Response的Header
        WriteHeader(int)
}

// Flusher的作用是被Handler调用来将写缓存中的数据推给客户端
type Flusher interface {
        // 这个方法将写缓存中数据推送给客户端
        Flush()
}

// Hijacker的作用是被Handler调用来关闭连接的
type Hijacker interface {
        // 这个方法让调用者主动管理连接
        Hijack() (net.Conn, *bufio.ReadWriter, error)

}
```

response

实现这三个接口的结构是response(这个结构是http包私有的，在文档中并没有显示，需要去看源码)

```
// response包含了所有server端的http返回信息
type response struct {
        conn          *conn          // 保存此次HTTP连接的信息
        req           *Request // 对应请求信息
        chunking      bool     // 是否使用chunk
        wroteHeader   bool     // header是否已经执行过写操作
        wroteContinue bool     // 100 Continue response was written
        header        Header   // 返回的http的Header
        written       int64    // Body的字节数
        contentLength int64    // Content长度
        status        int      // HTTP状态
        needSniff     bool     // 是否需要使用sniff。（当没有设置Content-Type的时候，开启sniff能根据HTTP body来

        closeAfterReply bool     //是否保持长链接。如果客户端发送的请求中connection有keep-alive，这个字段就设置为

        requestBodyLimitHit bool //是否requestBody太大了（当requestBody太大的时候，response是会返回411状态的，并

}
```

在response中是可以看到

```
func (w *response) Header() Header
func (w *response) WriteHeader(code int)
func (w *response) Write(data []byte) (n int, err error)
func (w *response) Flush()
func (w *response) Hijack() (rwc net.Conn, buf *bufio.ReadWriter, err error)
```

这么几个方法。所以说response实现了ResponseWriter,Flusher,Hijacker这三个接口

HandlerFunc

handlerFunc是经常使用到的一个type

```
// 这里将HandlerFunc定义为一个函数类型，因此以后当调用a = HandlerFunc(f)之后，调用a的ServeHttp实际上就是调用f的对
type HandlerFunc func(ResponseWriter, *Request)

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

这里需要多回味一下了，这个HandlerFunc定义和ServeHTTP合起来是说明了什么？说明HandlerFunc的所有实例是实现了ServeHttp方法的。另，实现了ServeHttp方法就是什么？实现了接口Handler!

所以你以后会看到很多这样的句子：

```
func AdminHandler(w ResponseWriter, r *Request) {
    ...
}
handler := HandlerFunc(AdminHandler)
handler.ServeHttp(w,r)
```

请不要讶异，你明明没有写ServeHttp，怎么能调用呢？ 实际上调用ServeHttp就是调用AdminHandler。

好吧，理解这个也花了我较长时间，附带上一个play.google写的一个小例子

http://play.golang.org/p/nSt_wcjc2u

有兴趣继续研究的同学可以继续试验下去

如果你理解了HandlerFunc，你对下面两个句子一定不会讶异了

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

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

下面接着看Server.go

ServerMux结构

它就是http包中的路由规则器。你可以在ServerMux中注册你的路由规则，当有请求到来的时候，根据这些路由规则来判断将请求分发到哪个处理器（Handler）。

它的结构如下：

```
type ServeMux struct {
        mu sync.RWMutex   //锁，由于请求设计到并发处理，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就是我注册的路由表达式
}
```

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
}
```

看到这两个结构就应该对请求是**如何路由**的有思路了：

当一个请求request进来的时候，server会依次根据ServeMux.m中的string（路由表达式）来一个一个匹配，如果找到了可以匹配的muxEntry,就取出muxEntry.h,这是个handler，调用handler中的ServeHTTP（ResponseWriter, *Request）来组装Response，并返回。

ServeMux定义的方法有：

```
func (mux *ServeMux) match(path string) Handler   //根据path获取Handler
func (mux *ServeMux) handler(r *Request) Handler  //根据Request获取Handler，内部实现调用match
func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request) //！！这个说明，ServeHttp也实现了Handler接口，它实
func (mux *ServeMux) Handle(pattern string, handler Handler) //注册handler方法

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

在godoc文档中经常见到的**DefaultServeMux是http默认使用的ServeMux**

var DefaultServeMux = NewServeMux()

如果我们没有自定义ServeMux，系统默认使用这个ServeMux。

换句话说，http包外层（非ServeMux）中提供的几个方法：

```
func Handle(pattern string, handler Handler) { DefaultServeMux.Handle(pattern, handler) }
func HandleFunc(pattern string, handler func(ResponseWriter, *Request)) {
        DefaultServeMux.HandleFunc(pattern, handler)
}
```

实际上就是调用ServeMux结构内部对应的方法。

Server

下面还剩下一个Server结构

```
type Server struct {
        Addr           string        // 监听的地址和端口
        Handler        Handler       // 所有请求需要调用的Handler（实际上这里说是ServeMux更确切）如果为空则设置为
        ReadTimeout    time.Duration // 读的最大Timeout时间
        WriteTimeout   time.Duration // 写的最大Timeout时间
        MaxHeaderBytes int           // 请求头的最大长度
        TLSConfig      *tls.Config   // 配置TLS
}
```

Server提供的方法有：

```
func (srv *Server) Serve(l net.Listener) error   //对某个端口进行监听，里面就是调用for进行accept的处理了
func (srv *Server) ListenAndServe() error  //开启http server服务，内部调用Serve
func (srv *Server) ListenAndServeTLS(certFile, keyFile string) error //开启https server服务，内部调用Serve
```

当然Http包也直接提供了方法供外部使用，实际上内部就是实例化一个Server，然后调用ListenAndServe方法

```
func ListenAndServe(addr string, handler Handler) error   //开启Http服务
func ListenAndServeTLS(addr string, certFile string, keyFile string, handler Handler) error //开启HTTPs服务
```

具体例子分析

下面根据上面的分析，我们对一个例子我们进行阅读。这个例子搭建了一个最简易的Server服务。当调用**http://XXXX:12345/hello**的时候页面会返回"hello world"

```
func HelloServer(w http.ResponseWriter, req *http.Request) {
        io.WriteString(w, "hello, world!\n")
}

func main() {
        http.HandleFunc("/hello", HelloServer)
        err := http.ListenAndServe(":12345", nil)
        if err != nil {
                log.Fatal("ListenAndServe: ", err)
        }

}
```

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(":12345", 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header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

后记

对于net.http包中server的理解是非常重要的。理清serverMux, responseWriter, Handler, HandlerFunc等常用结构和函数是使用go web的重要一步。个人感觉由于go中文档较少，像这样有点复杂的包，看godoc的效果就远不如直接看代码来的快和清晰了。实际上在理解了http包后，才会对godoc中出现的句子有所理解。后续还会写一些文章关于使用net.http构建web server的。请期待之。

本文来自：博客园

感谢作者：yjf512

查看原文：Golang Http Server源码阅读

上一篇：编译更小的 Golang 程序 下一篇：Golang中如何让html/template不转义html标签

1 回复 | 直到 2017-10-10 19:05:35 http net web 信息

itfanr · #1 · 8月之前

```
    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)
    }
```