链滴

Go net/http 浅析

作者：Allenxuxu
原文链接：https://ld246.com/article/1568526295750
来源网站：链滴

GO HTTP Server

使用标准库构建 HTTP 服务

Go 语言标准库自带一个完善的 net/http 包，可以很方便编写一个可以直接运行的 Web 服务。

```
package main

import (
    "log"
    "net/http"
)

func hello(w http.ResponseWriter, r *http.Request) {
    log.Println(r.Method, r.Host, r.RequestURI)
    w.Write([]byte("hello"))
}

func main() {
    http.HandleFunc("/hello", hello)            //设置访问的路由
    // http.Handle("/hello", http.HandlerFunc(hello)) // 和上面写法等价

    err := http.ListenAndServe(":9090", nil)    //设置监听的端口并启动 HTTP 服务
    if err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}

$  curl -v 127.0.0.1:9090/hello
*   Trying 127.0.0.1...
```

原文链接：Go net/http 浅析

```
* TCP_NODELAY set
* Connected to 127.0.0.1 (127.0.0.1) port 9090 (#0)
> GET /hello HTTP/1.1
> Host: 127.0.0.1:9090
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Tue, 10 Sep 2019 10:52:07 GMT
< Content-Length: 5
< Content-Type: text/plain; charset=utf-8
<
* Connection #0 to host 127.0.0.1 left intact
hello
```

上面短短几行代码，已经启动了一个 HTTP 服务。 在浏览输入 127.0.0.1:9090/hello 或者执行 curl -v 127.0.0.1:9090/hello 可以验证。

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

http.HandleFunc("/hello", hello) 会在 net/http 的默认路由中注册 hello 处理函数，这也是我们为么在 http.ListenAndServe(":9090", nil) 中传入 nil，传入 nil 意味着使用默认的路由器。

上面的 main 函数和如下其实是一样的：

```
func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/hello", hello)            //设置访问的路由
    // mux.Handle("/hello", http.HandlerFunc(hello)) // 和上面的写法等价

    err := http.ListenAndServe(":9090", mux)    //设置监听的端口并启动 HTTP 服务
    if err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}
```

Go 自带的 http.ServerMux 实现比较简单，只支持路径匹配，不支持按照 Method 等信息匹配，没直接实现 RESTful 接口，所有有很多其他优秀的路由器和 HTTP 库实现，后面的文章中会介绍。

Go net/http 库浅析

Go 的标准库 net/http 内部处理了 TCP 连接和 HTTP 报文解析的等繁琐的细节，仅仅对外提供 HTTP 处理的相关接口。

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

开发者只需实现对应的 Handler 接口并注册， 在处理函数中和 http.request 、 http.ResponseWrite 交互读取请求信息，设置返回信息即可，就像文章开头的例子那样。

- 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

- ResponseWriter： 服务器需要返回给客户端的信息

mux.HandleFunc("/hello", hello) 第一个参数是 URL 路径，第二个参数就是设置的 Handler。这里 et/http 做了一个适配器，让我们可以不用每次都定义一个结构体去实现 ServeHTTP(ResponseWriter *Request) 。

第二个参数传入一个函数，并其函数签名为 func(ResponseWriter, *Request)，内部通过适配器将封装，主要代码如下：

```
// 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))
}

// The HandlerFunc type is an adapter to allow the use of
// ordinary functions as HTTP handlers. If f is a function
// with the appropriate signature, HandlerFunc(f) is a
// Handler that calls f.
type HandlerFunc func(ResponseWriter, *Request)

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

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

net/http 库中会去调用 ServeHTTP 方法，这也是接口规定我们实现的方法。HandlerFunc 适配器封了它，在其内部调用我们传入的函数 f(w, r) 。

我们一步步查看最后启动 Web 服务的 ListenAndServe 实现：

```
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)      //创建一个 TCP listener
    if err != nil {
        return err
    }
    return srv.Serve(tcpKeepAliveListener{ln.(*net.TCPListener)})
```

```
}
```

上面两层封装，主要是保存了 HTTP Server 的运行参数，并且创建了 TCP Listener ，最后 Serve 方会进入真正的循环。

```
func (srv *Server) Serve(l net.Listener) error {
    if fn := testHookServerServe; fn != nil {
        fn(srv, l) // call hook with unwrapped listener
    }

    l = &onceCloseListener{Listener: l}
    defer l.Close()

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

    if !srv.trackListener(&l, true) {
        return ErrServerClosed
    }
    defer srv.trackListener(&l, false)

    var tempDelay time.Duration     // how long to sleep on accept failure
    baseCtx := context.Background() // base is always background, per Issue 16220
    ctx := context.WithValue(baseCtx, ServerContextKey, srv)
    // 死循环，不断接受客户端连接处理
    for {
        rw, e := l.Accept()                         // 接受客户端连接
        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)             // 启动一个协程来执行处理逻辑
    }
```

```
}
```

这个函数内部有一个无限循环会不断接受新的客户断连接，并且启动一个协程来处理它。

```
func (c *conn) serve() {

    ...

  for {
      w, err := c.readRequest()
      if c.lr.N != c.server.initialLimitedReaderSize() {
          // If we read any bytes off the wire, we're active.
          c.setState(c.rwc, StateActive)
      }

       ...

      // HTTP cannot have multiple simultaneous active requests.[*]
      // Until the server replies to this request, it can't read another,
      // so we might as well run the handler in this goroutine.
      // [*] Not strictly true: HTTP pipelining.  We could let them all process
      // in parallel even if their responses need to be serialized.
      serverHandler{c.server}.ServeHTTP(w, w.req)

      w.finishRequest()
      if w.closeAfterReply {
          if w.requestBodyLimitHit {
              c.closeWriteAndWait()
          }
          break
      }
      c.setState(c.rwc, StateIdle)
  }
}
```

对客户端的请求处理，会执行 serverHandler{c.server}.ServeHTTP(w, w.req) ，这里面会调用我们册的路由器 ServeHTTP 方法，继而根据路由判断，调用我们注册的 Handler 。

```
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)
}
```

接下来我们看看默认的路由器 ServeMux 的实现：

```
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
}
```

内部通过一个 map 来实现路由映射，这也是它只支持路径匹配，不支持按照 Method 等信息匹配的因。我们知道在对客户端的请求处理中会首先调用其 ServeHTTP 方法，我们先来看看其实现：

```
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)
}
```

这个函数非常短小，主要是首先执行 h, _ := mux.Handler(r) 来匹配路由，然后再调用其 ServeHTTP 也就是我们注册的 Handler。

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

    // CONNECT requests are not canonicalized.
    if r.Method == "CONNECT" {
        // If r.URL.Path is /tree and its handler is not registered,
        // the /tree -> /tree/ redirect applies to CONNECT requests
        // but the path canonicalization does not.
        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)
}
```

ServeMux 的 Handler 方法内部主要就是根据用户请求的 URL 来找到其对应的 Handler ，也就是 mux.HandleFunc("/hello", hello) 中注册的路由和 Handler 。

我们梳理一下 Go Web 的主要执行流程：

- 启动 TCP Server 监听指定端口，等待客户端连接
- 接受客户端连接，并启动一个协程单独处理客户端逻辑
- 在新启动的协程中，默认路由器根据 URL 匹配对应的用户处理函数并执行

接下来，我们看下业务开发时接触最多的 Request 和 ResponseWriter 。

```
type Request struct {
    Method string
    URL *url.URL
    Proto      string // "HTTP/1.0"
    ProtoMajor int    // 1
    ProtoMinor int    // 0
    Header Header
    Body io.ReadCloser
    GetBody func() (io.ReadCloser, error)
    ContentLength int64
    TransferEncoding []string
    Close bool
    Host string
    Form url.Values
    PostForm url.Values
    MultipartForm *multipart.Form
    Trailer Header
    RemoteAddr string
    RequestURI string
    TLS *tls.ConnectionState
    Cancel <-chan struct{}
    Response *Response
    ctx context.Context
}
```

从 Request 结构体中，可以看出，我们在 Handler 需要的 HTTP 请求相关信息都在这个结构体中，实际开发中通过 Request 的公开方法或者直接读取公开变量获取。

```
type ResponseWriter interface {
    Header() Header
    Write([]byte) (int, error)
    WriteHeader(statusCode int)
}
```

ResponseWriter 的实现更加简洁，主要就是通过 Header 来设置返回头，Write 来设置返回 body WriteHeader 来设置返回状态码。

关于 Request 和 ResponseWriter 更多的使用方法这里就不细说，可以查阅其他相关资料。

至此，我们已经大概清楚 net/http 的大概工作流程了。得益于 Go 协程的轻量，net/http 库采用 per request per goroutine ，这使得 Go 的 HTTP 请求处理非常快速。同时 net/http 内部封装大量细，让开发者通过简单的 API 调用就可以搭建 HTTP 服务。

原文链接：Go net/http 浅析