[toc]

demo

```
// demo
pingHandler := func(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintf(w, "pong")
}

http.HandleFunc("/ping", pingHandler)
http.ListenAndServe(":8060", nil)
```

struct

最最核心的接口，实现了ServeHTTP方法

```
// A Handler responds to an HTTP request.
//
// ServeHTTP should write reply headers and data to the ResponseWriter
// and then return. Returning signals that the request is finished; it
// is not valid to use the ResponseWriter or read from the
// Request.Body after or concurrently with the completion of the
// ServeHTTP call.
//
// Depending on the HTTP client software, HTTP protocol version, and
// any intermediaries between the client and the Go server, it may not
// be possible to read from the Request.Body after writing to the
// ResponseWriter. Cautious handlers should read the Request.Body
// first, and then reply.
//
// Except for reading the body, handlers should not modify the
// provided Request.
//
// If ServeHTTP panics, the server (the caller of ServeHTTP) assumes
// that the effect of the panic was isolated to the active request.
// It recovers the panic, logs a stack trace to the server error log,
// and either closes the network connection or sends an HTTP/2
// RST_STREAM, depending on the HTTP protocol. To abort a handler so
// the client sees an interrupted response but the server doesn't log
// an error, panic with the value ErrAbortHandler.
type Handler interface {
        ServeHTTP(ResponseWriter, *Request)
}
```

net/http默认的函数签名，实现了ServeHTTP接口，自己实现ServeHTTP接口也可以的

```
// 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)
}
```

ServeMux，HandleFunc Handle注册，Handler路由，内部是一个map

```
// ServeMux is an HTTP request multiplexer.
// It matches the URL of each incoming request against a list of
registered
// patterns and calls the handler for the pattern that
// most closely matches the URL.
//
// Patterns name fixed, rooted paths, like "/favicon.ico",
// or rooted subtrees, like "/images/" (note the trailing slash).
// Longer patterns take precedence over shorter ones, so that
// if there are handlers registered for both "/images/"
// and "/images/thumbnails/", the latter handler will be
// called for paths beginning "/images/thumbnails/" and the
// former will receive requests for any other paths in the
// "/images/" subtree.
//
// Note that since a pattern ending in a slash names a rooted subtree,
// the pattern "/" matches all paths not matched by other registered
// patterns, not just the URL with Path == "/".
//
// If a subtree has been registered and a request is received naming the
// subtree root without its trailing slash, ServeMux redirects that
// request to the subtree root (adding the trailing slash). This behavior
can
// be overridden with a separate registration for the path without
// the trailing slash. For example, registering "/images/" causes ServeMux
// to redirect a request for "/images" to "/images/", unless "/images" has
// been registered separately.
//
// Patterns may optionally begin with a host name, restricting matches to
// URLs on that host only. Host-specific patterns take precedence over
// general patterns, so that a handler might register for the two patterns
// "/codesearch" and "codesearch.google.com/" without also taking over
// requests for "http://www.google.com/".
//
// ServeMux also takes care of sanitizing the URL request path and the
Host
// header, stripping the port number and redirecting any request
containing . or
```

```
// .. elements or repeated slashes to an equivalent, cleaner URL.
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
}
```

HTTP server的封装

```
// A Server defines parameters for running an HTTP server.
// The zero value for Server is a valid configuration.
type Server struct {
        Addr    string  // TCP address to listen on, ":http" if empty
        Handler Handler // handler to invoke, http.DefaultServeMux if nil

        // TLSConfig optionally provides a TLS configuration for use
        // by ServeTLS and ListenAndServeTLS. Note that this value is
        // cloned by ServeTLS and ListenAndServeTLS, so it's not
        // possible to modify the configuration with methods like
        // tls.Config.SetSessionTicketKeys. To use
        // SetSessionTicketKeys, use Server.Serve with a TLS Listener
        // instead.
        TLSConfig *tls.Config

        // ReadTimeout is the maximum duration for reading the entire
        // request, including the body.
        //
        // Because ReadTimeout does not let Handlers make per-request
        // decisions on each request body's acceptable deadline or
        // upload rate, most users will prefer to use
        // ReadHeaderTimeout. It is valid to use them both.
        ReadTimeout time.Duration

        // ReadHeaderTimeout is the amount of time allowed to read
        // request headers. The connection's read deadline is reset
        // after reading the headers and the Handler can decide what
        // is considered too slow for the body.
        ReadHeaderTimeout time.Duration

        // WriteTimeout is the maximum duration before timing out
        // writes of the response. It is reset whenever a new
        // request's header is read. Like ReadTimeout, it does not
        // let Handlers make decisions on a per-request basis.
        WriteTimeout time.Duration

        // IdleTimeout is the maximum amount of time to wait for the
```

```
        // next request when keep-alives are enabled. If IdleTimeout
        // is zero, the value of ReadTimeout is used. If both are
        // zero, ReadHeaderTimeout is used.
        IdleTimeout time.Duration

        // MaxHeaderBytes controls the maximum number of bytes the
        // server will read parsing the request header's keys and
        // values, including the request line. It does not limit the
        // size of the request body.
        // If zero, DefaultMaxHeaderBytes is used.
        MaxHeaderBytes int

        // TLSNextProto optionally specifies a function to take over
        // ownership of the provided TLS connection when an NPN/ALPN
        // protocol upgrade has occurred. The map key is the protocol
        // name negotiated. The Handler argument should be used to
        // handle HTTP requests and will initialize the Request's TLS
        // and RemoteAddr if not already set. The connection is
        // automatically closed when the function returns.
        // If TLSNextProto is not nil, HTTP/2 support is not enabled
        // automatically.
        TLSNextProto map[string]func(*Server, *tls.Conn, Handler)

        // ConnState specifies an optional callback function that is
        // called when a client connection changes state. See the
        // ConnState type and associated constants for details.
        ConnState func(net.Conn, ConnState)

        // ErrorLog specifies an optional logger for errors accepting
        // connections, unexpected behavior from handlers, and
        // underlying FileSystem errors.
        // If nil, logging is done via the log package's standard logger.
        ErrorLog *log.Logger

        disableKeepAlives int32     // accessed atomically.
        inShutdown        int32     // accessed atomically (non-zero means
we're in Shutdown)
        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()
}
```

对net.TCPListener的封装，Accept之后设置TCP keep-alive，默认3分钟

```
// tcpKeepAliveListener sets TCP keep-alive timeouts on accepted
// connections. It's used by ListenAndServe and ListenAndServeTLS so
```

```
// dead TCP connections (e.g. closing laptop mid-download) eventually
// go away.
type tcpKeepAliveListener struct {
        *net.TCPListener
}
func (ln tcpKeepAliveListener) Accept() (net.Conn, error) {
        tc, err := ln.AcceptTCP()
        if err != nil {
                return nil, err
        }
        tc.SetKeepAlive(true)
        tc.SetKeepAlivePeriod(3 * time.Minute)
        return tc, nil
}
```

sync.Once对net.Listener封装，保证只关闭一次

```
// onceCloseListener wraps a net.Listener, protecting it from
// multiple Close calls.
type onceCloseListener struct {
        net.Listener
        once     sync.Once
        closeErr error
}
func (oc *onceCloseListener) Close() error {
        oc.once.Do(oc.close)
        return oc.closeErr
}
func (oc *onceCloseListener) close() { oc.closeErr = oc.Listener.Close() }
```

一个http连接的封装

```
// A conn represents the server side of an HTTP connection.
type conn struct {
        // server is the server on which the connection arrived.
        // Immutable; never nil.
        server *Server

        // cancelCtx cancels the connection-level context.
        cancelCtx context.CancelFunc

        // rwc is the underlying network connection.
        // This is never wrapped by other types and is the value given out
        // to CloseNotifier callers. It is usually of type *net.TCPConn or
        // *tls.Conn.
        rwc net.Conn

        // remoteAddr is rwc.RemoteAddr().String(). It is not populated
synchronously
```

```
        // inside the Listener's Accept goroutine, as some implementations
block.
        // It is populated immediately inside the (*conn).serve goroutine.
        // This is the value of a Handler's (*Request).RemoteAddr.
        remoteAddr string

        // tlsState is the TLS connection state when using TLS.
        // nil means not TLS.
        tlsState *tls.ConnectionState

        // werr is set to the first write error to rwc.
        // It is set via checkConnErrorWriter{w}, where bufw writes.
        werr error

        // r is bufr's read source. It's a wrapper around rwc that
provides
        // io.LimitedReader-style limiting (while reading request headers)
        // and functionality to support CloseNotifier. See *connReader
docs.
        r *connReader

        // bufr reads from r.
        bufr *bufio.Reader

        // bufw writes to checkConnErrorWriter{c}, which populates werr on
error.
        bufw *bufio.Writer

        // lastMethod is the method of the most recent request
        // on this connection, if any.
        lastMethod string

        curReq atomic.Value // of *response (which has a Request in it)

        curState struct{ atomic uint64 } // packed
(unixtime<<8|uint8(ConnState))

        // mu guards hijackedv
        mu sync.Mutex

        // hijackedv is whether this connection has been hijacked
        // by a Handler with the Hijacker interface.
        // It is guarded by mu.
        hijackedv bool
}
```

代理Server Handler，DefaultServeMux or handler

```
// serverHandler delegates to either the server's Handler or
// DefaultServeMux and also handles "OPTIONS *" requests.
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)
}
```

regist

直接调用http.HandleFunc注册在DefaultServeMux上，ServeMux.HandleFunc，ServeMux.Handle

```
// HandleFunc registers the handler function for the given pattern
// in the DefaultServeMux.
// The documentation for ServeMux explains how patterns are matched.
func HandleFunc(pattern string, handler func(ResponseWriter, *Request)) {
        DefaultServeMux.HandleFunc(pattern, handler)
}
// 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))
}
```

或者直接调用http.Handle注册在DefaultServeMux上，ServeMux.Handle

```
// Handle registers the handler for the given pattern
// in the DefaultServeMux.
// The documentation for ServeMux explains how patterns are matched.
func Handle(pattern string, handler Handler) {
DefaultServeMux.Handle(pattern, handler) }
```

或者NewServeMux生成一个ServeMux，然后通过ServeMux.Handle注册

```
// NewServeMux allocates and returns a new ServeMux.
func NewServeMux() *ServeMux { return new(ServeMux) }
```

最终的方法注册实现，map操作加锁 对于以/结尾的path，加入es数组，es数组按path长度递减排序 如果不以/开头，则包含host

```
// Handle registers the handler for the given pattern.
// If a handler already exists for pattern, Handle panics.
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")
        }
        if _, exist := mux.m[pattern]; exist {
                panic("http: multiple registrations for " + pattern)
        }

        if mux.m == nil {
                mux.m = make(map[string]muxEntry)
        }
        e := muxEntry{h: handler, pattern: pattern}
        mux.m[pattern] = e
        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
}
```

run

http.ListenAndServe启动服务，生成了一个默认Server，然后Server.ListenAndServe，自己构造Server可以做一些设置

```
// ListenAndServe listens on the TCP network address addr and then calls
// Serve with handler to handle requests on incoming connections.
// Accepted connections are configured to enable TCP keep-alives.
//
// The handler is typically nil, in which case the DefaultServeMux is
used.
//
// ListenAndServe always returns a non-nil error.
func ListenAndServe(addr string, handler Handler) error {
        server := &Server{Addr: addr, Handler: handler}
        return server.ListenAndServe()
}
```

监听端口，tcpKeepAliveListener封装listener，进入Server.Serve

```
// ListenAndServe listens on the TCP network address srv.Addr and then
// calls Serve to handle requests on incoming connections.
// Accepted connections are configured to enable TCP keep-alives.
//
// If srv.Addr is blank, ":http" is used.
//
// ListenAndServe always returns a non-nil error. After Shutdown or Close,
// the returned error is ErrServerClosed.
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(tcpKeepAliveListener{ln.(*net.TCPListener)})
}
```

事件循环主体，生成了一个所有请求共享的context，http-server->当前Server 开始for循环，accept，如果发生错误，判断是不是已经关闭了，或者开始重试；如果正常，conn封装一次连接请求，设置为StateNew，启动一个goroutine处理请求conn.serve

```
// Serve accepts incoming connections on the Listener l, creating a
// new service goroutine for each. The service goroutines read requests
and
```

```
// then call srv.Handler to reply to them.
//
// HTTP/2 support is only enabled if the Listener returns *tls.Conn
// connections and they were configured with "h2" in the TLS
// Config.NextProtos.
//
// Serve always returns a non-nil error and closes l.
// After Shutdown or Close, the returned error is ErrServerClosed.
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)
        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)
        }
}
```

一个请求的处理逻辑 获取remote addr 把local addr写入context，local-addr recover context WithCancel serverHandler{c.server}.ServeHTTP(w, w.req)，通过serverHandler调用DefaultServeMux or handler的ServeHTTP方法

```
// Serve a new connection.
func (c *conn) serve(ctx context.Context) {
        c.remoteAddr = c.rwc.RemoteAddr().String()
        ctx = context.WithValue(ctx, LocalAddrContextKey,
c.rwc.LocalAddr())
        defer func() {
                if err := recover(); err != nil && err != ErrAbortHandler
{
                        const size = 64 << 10
                        buf := make([]byte, size)
                        buf = buf[:runtime.Stack(buf, false)]
                        c.server.logf("http: panic serving %v: %v\n%s",
c.remoteAddr, err, buf)
                }
                if !c.hijacked() {
                        c.close()
                        c.setState(c.rwc, StateClosed)
                }
        }()

        if tlsConn, ok := c.rwc.(*tls.Conn); ok {
                if d := c.server.ReadTimeout; d != 0 {
                        c.rwc.SetReadDeadline(time.Now().Add(d))
                }
                if d := c.server.WriteTimeout; d != 0 {
                        c.rwc.SetWriteDeadline(time.Now().Add(d))
                }
                if err := tlsConn.Handshake(); err != nil {
                        // If the handshake failed due to the client not
speaking
                        // TLS, assume they're speaking plaintext HTTP and
write a
                        // 400 response on the TLS conn's underlying
net.Conn.
                        if re, ok := err.(tls.RecordHeaderError); ok &&
re.Conn != nil && tlsRecordHeaderLooksLikeHTTP(re.RecordHeader) {
                                io.WriteString(re.Conn, "HTTP/1.0 400 Bad
Request\r\n\r\nClient sent an HTTP request to an HTTPS server.\n")
                                re.Conn.Close()
```

```
                                return
                        }
                        c.server.logf("http: TLS handshake error from %s:
%v", c.rwc.RemoteAddr(), err)
                        return
                }
                c.tlsState = new(tls.ConnectionState)
                *c.tlsState = tlsConn.ConnectionState()
                if proto := c.tlsState.NegotiatedProtocol; validNPN(proto)
{
                        if fn := c.server.TLSNextProto[proto]; fn != nil {
                                h := initNPNRequest{tlsConn,
serverHandler{c.server}}
                                fn(c.server, tlsConn, h)
                        }
                        return
                }
        }

        // HTTP/1.x from here on.

        ctx, cancelCtx := context.WithCancel(ctx)
        c.cancelCtx = cancelCtx
        defer cancelCtx()

        c.r = &connReader{conn: c}
        c.bufr = newBufioReader(c.r)
        c.bufw = newBufioWriterSize(checkConnErrorWriter{c}, 4<<10)

        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)
                }
                if err != nil {
                        const errorHeaders = "\r\nContent-Type:
text/plain; charset=utf-8\r\nConnection: close\r\n\r\n"

                        if err == errTooLarge {
                                // Their HTTP client may or may not be
                                // able to read this if we're
                                // responding to them and hanging up
                                // while they're still writing their
                                // request. Undefined behavior.
                                const publicErr = "431 Request Header
Fields Too Large"
                                fmt.Fprintf(c.rwc, "HTTP/1.1
"+publicErr+errorHeaders+publicErr)
                                c.closeWriteAndWait()
                                return
                        }
                        if isCommonNetReadError(err) {
```

```
                                return // don't reply
                        }

                        publicErr := "400 Bad Request"
                        if v, ok := err.(badRequestError); ok {
                                publicErr = publicErr + ": " + string(v)
                        }

                        fmt.Fprintf(c.rwc, "HTTP/1.1
"+publicErr+errorHeaders+publicErr)
                        return
                }

                // Expect 100 Continue support
                req := w.req
                if req.expectsContinue() {
                        if req.ProtoAtLeast(1, 1) && req.ContentLength !=
0 {
                                // Wrap the Body reader with one that
replies on the connection
                                req.Body =
&expectContinueReader{readCloser: req.Body, resp: w}
                        }
                } else if req.Header.get("Expect") != "" {
                        w.sendExpectationFailed()
                        return
                }

                c.curReq.Store(w)

                if requestBodyRemains(req.Body) {
                        registerOnHitEOF(req.Body,
w.conn.r.startBackgroundRead)
                } else {
                        w.conn.r.startBackgroundRead()
                }

                // 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.
                // But we're not going to implement HTTP pipelining
because it
                // was never deployed in the wild and the answer is
HTTP/2.
                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))

                if !w.conn.server.doKeepAlives() {
                        // We're in shutdown mode. We might've replied
                        // to the user without "Connection: close" and
                        // they might think they can send another
                        // request, but such is life with HTTP/1.1.
                        return
                }

                if d := c.server.idleTimeout(); d != 0 {
                        c.rwc.SetReadDeadline(time.Now().Add(d))
                        if _, err := c.bufr.Peek(4); err != nil {
                                return
                        }
                }
                c.rwc.SetReadDeadline(time.Time{})
        }
}

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)
}
```

ServeMux.Handler获取对应handler function，然后调用方法

```
// ServeHTTP dispatches the request to the handler whose
// pattern most closely matches the request URL.
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)
}
```

router

```
// Handler returns the handler to use for the given request,
// consulting r.Method, r.Host, and r.URL.Path. It always returns
// a non-nil handler. If the path is not in its canonical form, the
// handler will be an internally-generated handler that redirects
// to the canonical path. If the host contains a port, it is ignored
// when matching handlers.
//
// The path and host are used unchanged for CONNECT requests.
//
// Handler also returns the registered pattern that matches the
// request or, in the case of internally-generated redirects,
// the pattern that will match after following the redirect.
//
// If there is no registered handler that applies to the request,
// Handler returns a ``page not found'' handler and an empty pattern.
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)
}
// 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
}
```