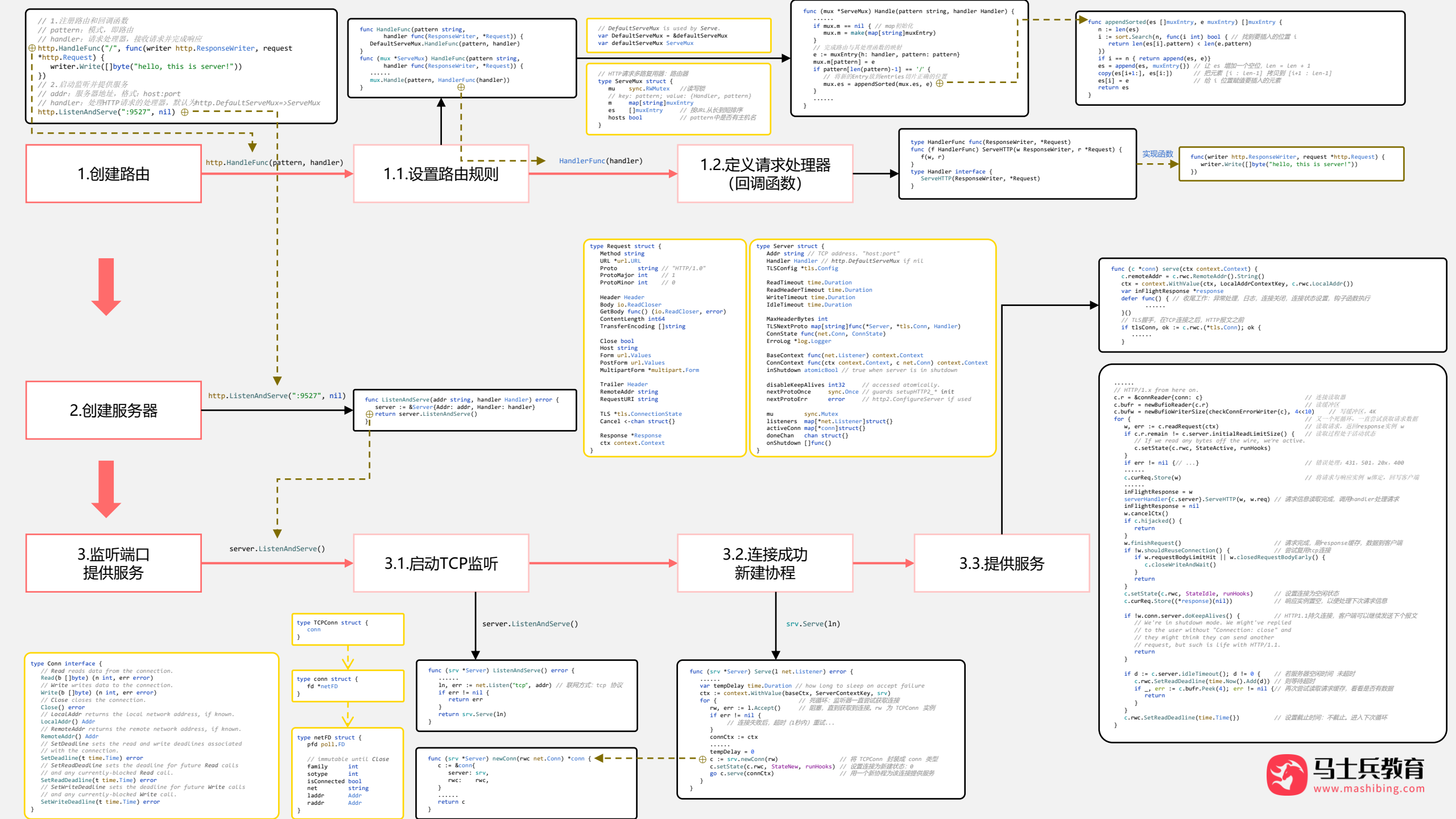


```go
// 1.注册路由和回调函数
// pattern：模式，即路由
// handler：请求处理器，接收请求并完成响应
http.HandleFunc("/", func(writer http.ResponseWriter, request
*http.Request) {
    writer.Write([]byte("hello, this is server!"))
})
// 2.启动监听并提供服务
// addr：服务器地址，格式：host:port
// handler：处理HTTP请求的处理器，默认为http.DefaultServeMux=>ServeMux
http.ListenAndServe(":9527", nil)
```

```go
func HandleFunc(pattern string,
        handler func(ResponseWriter, *Request)) {
    DefaultServeMux.HandleFunc(pattern, handler)
}
func (mux *ServeMux) HandleFunc(pattern string,
        handler func(ResponseWriter, *Request)) {
    ......
    mux.Handle(pattern, HandlerFunc(handler))
}
```

```go
// DefaultServeMux is used by Serve.
var DefaultServeMux = &defaultServeMux
var defaultServeMux ServeMux
```

```go
// HTTP请求多路复用器：路由器
type ServeMux struct {
    mu    sync.RWMutex   //读写锁
    // key: pattern; value: {Handler, pattern}
    m     map[string]muxEntry
    es    []muxEntry     // 按URL从长到短排序
    hosts bool           // pattern中是否有主机名
}
```

```go
func (mux *ServeMux) Handle(pattern string, handler Handler) {
    ......
    if mux.m == nil { // map初始化
        mux.m = make(map[string]muxEntry)
    }
    // 完成路由与其处理函数的映射
    e := muxEntry{h: handler, pattern: pattern}
    mux.m[pattern] = e
    if pattern[len(pattern)-1] == '/' {
        // 将新的Entry放到entries切片正确的位置
        mux.es = appendSorted(mux.es, e)
    }
    ......
}
```

```go
func appendSorted(es []muxEntry, e muxEntry) []muxEntry {
    n := len(es)
    i := sort.Search(n, func(i int) bool { // 找到要插入的位置 i
        return len(es[i].pattern) < len(e.pattern)
    })
    if i == n { return append(es, e)}
    es = append(es, muxEntry{}) // 让 es 增加一个空位，Len = Len + 1
    copy(es[i+1:], es[i:])      // 把元素 [i : Len-1] 拷贝到 [i+1 : Len-1]
    es[i] = e                   // 给 i 位置赋值要插入的元素
    return es
}
```

```go
type HandlerFunc func(ResponseWriter, *Request)
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

```go
func(writer http.ResponseWriter, request *http.Request) {
    writer.Write([]byte("hello, this is server!"))
})
```

```go
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
}
```

```go
type Server struct {
    Addr string // TCP address. "host:port"
    Handler Handler // http.DefaultServeMux if nil
    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 server is in shutdown

    disableKeepAlives int32     // accessed atomically.
    nextProtoOnce     sync.Once // guards setupHTTP2_* init
    nextProtoErr      error     // http2.ConfigureServer if used

    mu         sync.Mutex
    listeners  map[*net.Listener]struct{}
    activeConn map[*conn]struct{}
    doneChan   chan struct{}
    onShutdown []func()
}
```

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

```go
func (srv *Server) ListenAndServe() error {
    ......
    ln, err := net.Listen("tcp", addr) // 联网方式：tcp 协议
    if err != nil {
        return err
    }
    return srv.Serve(ln)
}
```

```go
type TCPConn struct {
    conn
}
```

```go
type conn struct {
    fd *netFD
}
```

```go
type netFD struct {
    pfd poll.FD

    // immutable until Close
    family      int
    sotype      int
    isConnected bool
    net         string
    laddr       Addr
    raddr       Addr
}
```

```go
type Conn interface {
    // Read reads data from the connection.
    Read(b []byte) (n int, err error)
    // Write writes data to the connection.
    Write(b []byte) (n int, err error)
    // Close closes the connection.
    Close() error
    // LocalAddr returns the local network address, if known.
    LocalAddr() Addr
    // RemoteAddr returns the remote network address, if known.
    RemoteAddr() Addr
    // SetDeadline sets the read and write deadlines associated
    // with the connection.
    SetDeadline(t time.Time) error
    // SetReadDeadline sets the deadline for future Read calls
    // and any currently-blocked Read call.
    SetReadDeadline(t time.Time) error
    // SetWriteDeadline sets the deadline for future Write calls
    // and any currently-blocked Write call.
    SetWriteDeadline(t time.Time) error
}
```

```go
func (srv *Server) newConn(rwc net.Conn) *conn {
    c := &conn{
        server: srv,
        rwc:    rwc,
    }
    ......
    return c
}
```

```go
func (srv *Server) Serve(l net.Listener) error {
    ......
    var tempDelay time.Duration // how long to sleep on accept failure
    ctx := context.WithValue(baseCtx, ServerContextKey, srv)
    for {                       // 死循环：监听器一直尝试获取连接
        rw, err := l.Accept()   // 阻塞，直到获取到连接，rw 为 TCPConn 实例
        if err != nil {
            // 连接失败后，超时（1秒内）重试...
        }
        connCtx := ctx
        ......
        tempDelay = 0
        c := srv.newConn(rw)            // 将 TCPConn 封装成 conn 类型
        c.setState(c.rwc, StateNew, runHooks) // 设置连接为新建状态：0
        go c.serve(connCtx)             // 用一个新协程为该连接提供服务
    }
}
```

```go
func (c *conn) serve(ctx context.Context) {
    c.remoteAddr = c.rwc.RemoteAddr().String()
    ctx = context.WithValue(ctx, LocalAddrContextKey, c.rwc.LocalAddr())
    var inFlightResponse *response
    defer func() { // 收尾工作：异常处理，日志，连接关闭，连接状态设置，钩子函数执行
        ......
    }()
    // TLS握手，在TCP连接之后，HTTP报文之前
    if tlsConn, ok := c.rwc.(*tls.Conn); ok {
        ......
    }
```

```go
......
// HTTP/1.x from here on.
c.r = &connReader{conn: c}                          // 连接读取器
c.bufr = newBufioReader(c.r)                        // 读缓冲区
c.bufw = newBufioWriterSize(checkConnErrorWriter{c}, 4<<10)     // 写缓冲区，4K
for {                                               // 又一个死循环，一直尝试获取请求数据
    w, err := c.readRequest(ctx)                    // 读取请求，返回response实例 w
    if c.r.remain != c.server.initialReadLimitSize() {  // 读取过程处于活动状态
        // If we read any bytes off the wire, we're active.
        c.setState(c.rwc, StateActive, runHooks)
    }
    if err != nil {// ...}                           // 错误处理：431，501，20x，400
    ......
    c.curReq.Store(w)                               // 将请求与响应实例 w绑定，回写客户端
    ......
    inFlightResponse = w
    serverHandler{c.server}.ServeHTTP(w, w.req) // 请求信息读取完成，调用handler处理请求
    inFlightResponse = nil
    w.cancelCtx()
    if c.hijacked() {
        return
    }
    w.finishRequest()                           // 请求完成，刷response缓存，数据到客户端
    if !w.shouldReuseConnection() {             // 尝试复用tcp连接
        if w.requestBodyLimitHit || w.closedRequestBodyEarly() {
            c.closeWriteAndWait()
        }
        return
    }
    c.setState(c.rwc, StateIdle, runHooks)      // 设置连接为空闲状态
    c.curReq.Store((*response)(nil))            // 响应实例置空，以便处理下次请求信息

    if !w.conn.server.doKeepAlives() {          // HTTP1.1持久连接，客户端可以继续发送下个报文
        // 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 {    // 若服务器空闲时间 未超时
        c.rwc.SetReadDeadline(time.Now().Add(d)) // 则等待超时
        if _, err := c.bufr.Peek(4); err != nil {// 再次尝试读取请求缓存，看看是否有数据
            return
        }
    }
    c.rwc.SetReadDeadline(time.Time{})          // 设置截止时间：不截止，进入下次循环
}
```