A multi-threaded web server

Introduction

In this project, we built a multithreaded web server in Golang with Goroutine.

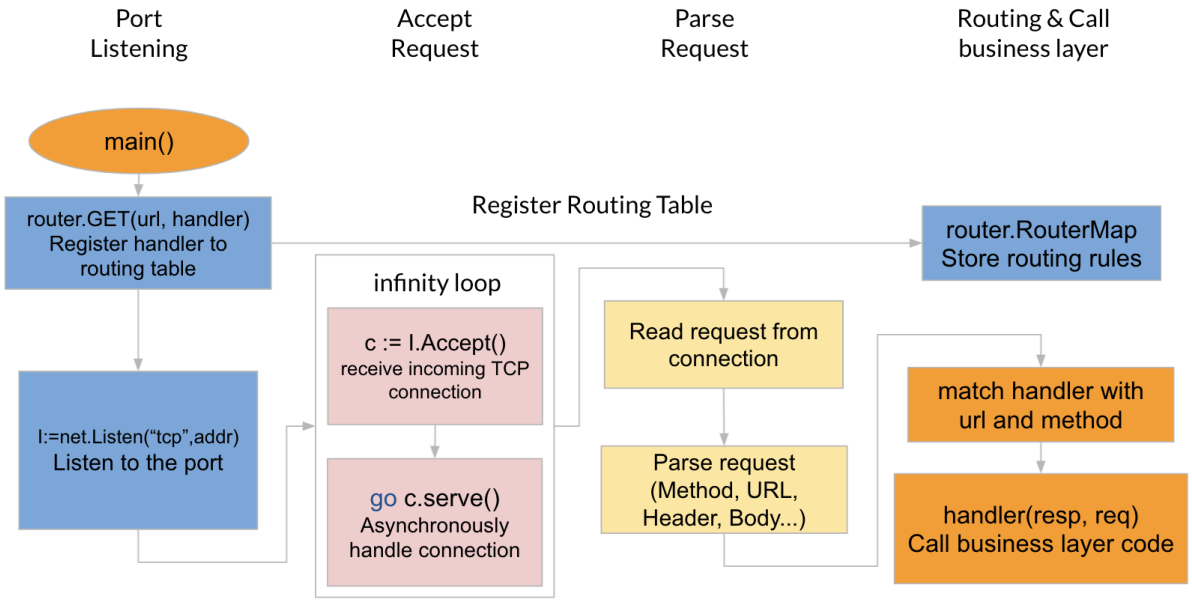
Goals

1. Handle HTTP requests with Goroutine concurrently
    a. Read and parse HTTP request data from TCP connection
    b. Write response data and send it back to the client
2. Implement Route Matching (Router)

HTTP Server Workflow

In our implementation, we:

1. Generate, bind and listen to the socket.
2. Waiting for connections.
3. Use Goroutine to handle each connection:
    a. Read data from connections.
    b. Parse HTTP header information.
    c. Generate HTTP responses.
    d. Write responses and return them to clients.



Implementations

Concurrency

To achieve concurrency, one of the methods is to use multi-threading. In each connection, we can start a new thread to handle it. However, spawning and destroying a thread is resource-consuming. One of the solutions is to use a thread pool to avoid frequently spawning and destroying threads.

In Golang, we can achieve concurrency with Goroutines. A Goroutine is a lightweight thread managed by the Go runtime. The cost of creating a Goroutine is very small, so it is suitable for handling concurrency HTTP requests.

In our implementation, we set up an infinite loop to wait for incoming TCP connections. After it receives a connection, we start a Goroutine to handle the connection asynchronously.

```
for {
   rw, err:= l.Accept()
   if err != nil {
      return err
   }
   c := &conn{
      server: srv,
      rwc:    rw,
```

```
    }
    go c.serve()
}
```

Router

Router or routing in web development is a mechanism where HTTP requests are routed to the code (handler) that handles them. In our implementation, we use a HashMap to store the corresponding business layer handler with the HTTP Method and URL as the key.

```
type Router struct {
    RouterMap map[routerKey]func(resp *Response, req *Request)
}
```

Our implementation can only exact match a handler. In other implementations, they will also use fuzzy matching to find the handler.

It is very easy to use our package. Initially, developers can new a router. Then register the business layer handler to the router. After that, developers can start to call `ListenAndServe` to boot the HTTP service.

```
r:= myhttp.NewRouter()
r.GET("/hello", bizLayerHandler)
myhttp.ListenAndServe(":8000", r)
```

HTTP Request Parsing

HTTP Message is human-readable. The request follow the following format:

```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: en

// Request Body
```

The first line of the request includes the HTTP Request method, path, and version of the protocol. The following lines indicate the header information in the HTTP request. After that, there is a request body.

In our method, we read data from the connection line by line and parse it into a struct for further use.

```
type Request struct {
   Method        string
   URL           *url.URL
   Header        Header
   Body          io.Reader
   ContentLength int64
   Host          string
   RemoteAddr    string
}
```

HTTP Response Generation

After calling the business layer handler, the result should return and send back to client via HTTP response. An HTTP response example:

```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Server: Apache
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
Content-Length: 29769
Content-Type: text/html

<!DOCTYPE HTML... (here come the 29769 bytes of the requested web page)
```

It consists of the following elements:

- The version of the HTTP protocol
- A status code
- A status message
- HTTP headers

- Response body

To generate response data, we define our response struct like following:

```
type Response struct {
   w          *bufio.Writer
   statusCode int
   conn       *conn
   header     Header
}
```

and format the response struct as a string and send it back to the client.