Programming Assignment #8

Introduction to Computer Networks

The Assignment

`PA8.go` should work like a Web (text/html) file server that receives and interprets the HTTP request messages from `curl`, as well as a regular Web browser such as Google Chrome and Mozilla Firefox and returns the requested file. More specifically, your `PA8.go` needs to:

(1) listen at <your port#> until there's an HTTP request
(2) read from the socket
(3) find the path and name of the text/html file requested
(4) In case the file exists, return the file so the Web browser displays the text/html file on screen
(5) In case the file doesn't exist, return a short message so the Web browser displays "File not found" on screen.
(6) close the connections and go back to (1)

To prepare you for the task, follow through the 3 examples below.

1. HTTP Response as a `string`

Let's start with the native string approach this time. Just like the 2nd example in PA7, one needs to review the message format to interpret an HTTP request message. To generate an HTTP Response from scratch, one will need to review HTTP response message format as well. This is so the Web browser will be able to interpret and render on the screen accordingly. This approach, seemingly tedious, is in fact straightforward. Start a file `string-Response.go` and type up the following code.

```
package main

import "fmt"
import "bufio"
import "net"
import "net/http"


func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {
    fmt.Println("Launching server...")
    ln, _ := net.Listen("tcp", ":<your port#>")
    defer ln.Close()
    conn, _ := ln.Accept()
    defer conn.Close()

    reader := bufio.NewReader(conn)
    req, err := http.ReadRequest(reader)
    check(err)
    fmt.Printf("Method: %s\n", req.Method)

    fmt.Fprintf(conn, "HTTP/1.1 404 Not Found\r\n")
    fmt.Fprintf(conn, "Date: ...\r\n")
    fmt.Fprintf(conn, "\r\n")
    fmt.Fprintf(conn, "File not found\r\n")
    fmt.Fprintf(conn, "\r\n")
}
```

Replace `<your port#>` with the port number assigned to your team. Start the `string-Response.go` first. Then, start a Chrome or Firefox browser and request for

http://127.0.0.1:<your port#>/. Alternatively, `curl` an arbitrary Web object on 127.0.0.1:<your port#>.

```
$ curl 127.0.0.1:<your port#>/
File not found

$
```

If you are using a Web browser for testing, the browser should also display “File not found”. The output is as expected given the example is returning 404 Not Found no matter what.

Code walk-through:

- The first part of the example is identical to `simple-Request.go`.
- What’s new are the last 5 lines. `fmt.Fprintf()` has been introduced in PA2 (in the `hello-whoever.go` example). It writes 5 lines to `conn`, the network socket connecting to the Web browser.
- The Web browser will be receiving a (minimal) HTTP response message, only 5 lines. First is the status line, second a lame header line, and third an empty line that signals the end of the header lines. Fourth is the real data. The empty line in the end signals the end of the HTTP response message.
- Most Web browsers receiving the first three lines (status line and at least 1 header line) will consider the message legitimate and displays the text/html content in the data field.
- You go ahead and try other Web browsers out. Please let polly know if any Web browser rejects this HTTP response message.

2. HTTP Response by the built-in `http.FileServer()`

The alternative is to send HTTP response messages exploiting the built-in file server – `http.FileServer()`. It interprets the incoming HTTP request messages and generates HTTP response messages accordingly. The functionality of a Web file server is already implemented. `simple-Response.go` is super brief.

```
package main

import "fmt"
import “net/http”

func main() {
    fmt.Println("Launching server...")

    http.ListenAndServe(":<your port#>", \
    http.FileServer(http.Dir(".")))
}
```

Replace `<your port#>` and start `simple-Response.go`. Try `curl` an existing text/html file on 127.0.0.1:<your port#>, say `string-Response.go`.

```
$ curl 127.0.0.1:<your port#>/string-Response.go
package main

import "fmt"
import "bufio"
…                    <- (rest of string-Response.go)
$
```

We see the exact content of `string-Response.go` on the Web browser screen, which means the text object is downloaded to the Web browser.

Perhaps copying `simple-Response.go` to `PA8.go` will do… Now `curl` a non-existing text/html file, say `qwerty.htm`.

```
$ curl 127.0.0.1:<your port#>/qwerty.htm
404 page not found
$
```

Ouch. `404 page not found` is not quite what we are hoping to display – "File not found".

Code walk-through:

- `http.ListenAndServe()` meant to call the `ListenAndServe()` API defined in the `http` package (short for `net/http`).
- `ListenAndServe()` takes in 2 parameters – (1) a port number (e.g., `":8080"`) and (2) a function handling the incoming HTTP request message. In the example, `http.FileServer()` is called.
- What the API does is to (1) start a server listening at the port number and (2) pass the HTTP request message to the handling function, or just "handler".
- `http.FileServer()` meant to call the `FileServer()` API defined in the `http` package, and `FileServer()` implements the Golang built-in Web file server.
- `FileServer()` takes in 1 parameter, the home directory of the Web file server. In the example, it's specified by `http.Dir(".")`.
- `Dir` is a data type defined in the `http` package. It stores a directory in the file system as a string. The string `"."` meant the directory from which the server code is started, and `"/"` meant the root directory of the file system.
- `FileServer(http.Dir("."))` meant the built-in server will look from the server's home directory for the file being requested.
- If the file is not found, `FileServer()` returns all files in the directory specified by `http.Dir()`, as a way to suggest alternative files to request.
- If the file is found, `FileServer()` returns the file to the requested.
- Taking this approach, one will need to find a way around `FileServer()` to send back the a customized 404 response that says "File not found" instead. This is not trivial and does require hacking and some googling.

3. HTTP Response by the custom `http.FileServer()`

Your first impression might be that the Golang built-in Web server is very specific. It is quite the opposite. The file server part of it is quite specific alright, but there are ways to customize the Web server and the file server as well. The Web server is designed so that a client can send HTTP requests with specific URL prefixes for custom functionalities. This example showcases the "customizability" of the built-in Web server. `handler-Response.go` below implements a Web server that allows the `/hello` command and file download at the same time.

```
package main

import "fmt"
import "net/http"



func helloHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello, world!")
}

func main() {
    fmt.Println("Launching server...")

    hh := http.HandlerFunc(helloHandler)
    http.Handle("/hello", hh)
    fs := http.FileServer(http.Dir("."))
    http.Handle("/", http.StripPrefix("/", fs))
    http.ListenAndServe(":<your port#>", nil)
}
```

Start `handler-Response.go` first. Then, `curl` from another terminal.

```
$ curl 127.0.0.1:<your port#>/hello
Hello, world!
$
```

The `handler-Response.go` server replies to the request a string “Hello, world!”. In the meantime, the outputs of the following two `curl` commands remain the same as the outputs from `simple-Response.go`.

```
$ curl 127.0.0.1:<your port#>/string-Response.go
$ curl 127.0.0.1:<your port#>/qwerty.htm
```

Code walk-through:

- The `handler-Response.go` server handles the `/hello` command and works as well as a Web file server.
- The `http.ListenAndServe()` this time sets no handler. Note the 2nd parameter is `nil`. No worries. There’s a default handler the HTTP request messages will be forwarded to.
- The default handler, `DefaultServeMux`, is a demultiplexer in fact. The built-in Web server allows multiple handlers and each is associated with a URL prefix. The demultiplexer checks the URL prefix in the incoming HTTP request message and determines which handler to forward the message for processing.
- `http.Handle()` is the key API that associates a prefix to its handler and inserts the prefix-handler entry to the `DefaultServeMux`. `http.Handle("/hello", hh)` associates `/hello` with `hh`.
- `hh` is obtained by adapting a programmer-defined function to a handler function – `http.HandlerFunc(helloHandler)`.
- As a result, if the URL starts with `/hello` , `helloHandler` will be called and `"Hello, world!"` will be sent through the data socket `w` back to the client.
- `http.Handle("/", http.StripPrefix("/", fs))` associates `/` with `StripPrefix("/", fs)`.
- As a result, if the URL starts with `/` , `StripPrefix("/", fs)` will be called.
- `StripPrefix()` is a special API defined in `net/http`‘s source code, particularly in `server.go`, line 2040-2056.
- We can see in the code that if the prefix is not empty, it returns the following code block as the handler.

```
func(w ResponseWriter, r *Request) {
  if p := strings.TrimPrefix(r.URL.Path, prefix); len(p) <
len(r.URL.Path) {
```

```
        r2 := new(Request)
        *r2 = *r
        r2.URL = new(url.URL)
        *r2.URL = *r.URL
        r2.URL.Path = p
        h.ServeHTTP(w, r2)
    } else {
        NotFound(w, r)
    }
}
```

- It looks the prefix is trimmed off the URL.Path and then used to call the server's `ServeHTTP()` function. That is, the original URL.Path in the HTTP request are `/string-Response.go` and `/qwerty.htm` after `StripPrefix("/", fs)` will be simply `string-Response.go` and `qwerty.htm`. `ServeHTTP()` looks like the main code looking for `string-Response.go` and `qwerty.htm` in the file system.
- When the new path is not shorter than the original path, the handler function branches to `NotFound()`, which is where `404 page not found` is returned to the client (`server.go`, line 2029). Perhaps one can build a custom `StripPrefix()` and change the returned handler function to allow calling to a custom `NotFound()` when the object requested does not exist.
- If tracing the source code does not quite inspire, try google and see if other programmers have sought to send back custom 404 messages before.

4. PA8.go

Make sure your `PA8.go` is listening on the port number you are assigned to. To test your `PA8.go`, use `curl` and at least Google Chrome and Mozilla Firefox to request a text/html file existing and non-existing from the server's home directory.

To help you verify your implementation, polly has made the compiled byte code of her `PA8.go` (plain string solution) and `PA8-http.go` (`FileServer()` wrapper approach) available here: http://homepage.ntu.edu.tw/~pollyhuang/teach/intro-cn-pa/PA8/PA8 and http://homepage.ntu.edu.tw/~pollyhuang/teach/intro-cn-pa/PA8/PA8-http. Again, polly's `PA8` and `PA8-http` are configured to run on port# 11999.

Things get a bit complicated when the index.html file exists in the server's home directory. Assume that your `PA8.go` will not be tested from a directory containing the index.html. It is perfectly fine to see either the "File not found" response or to see the files and subdirectories in the server's home directory when the client requests to 127.0.0.1:<your port#>/.

5. Submit your PA8

`ssh` to the `140.112.42.221` workstation. At the team account's home directory, create a directory `PA8`. Upload your `PA8.go` to directory `PA8`. Test your `PA8.go` again on the workstation just to make sure it's working as expected.