Go 语言制作 Markdown 预览器

一、实验介绍

1.1 实验内容

我们将使用 Go 语言完成一个在线 Markdown 解析器。

1.2 实验知识点 【实验中的核心知识点，完成该课程的收获】

- **1.3 实验环境【实验使用的实验环境及核心开发及部署软件简单介绍】**

- Go 1.2.1
- Xfce终端

1.4 适合人群

本课程难度为一般，属于初级级别课程，适合具有Python基础的用户，熟悉python基础知识加深巩固。

1.5 代码获取

学习本项目课前，需要先学习Go 语言编程 (http://www.shiyanlou.com/courses/11)。

本项目课的所有源代码可以通过 wget 的方式下载。

```
$ cd ~
$ wget http://labfile.oss.aliyuncs.com/courses/56/golang-markdown-previewer.zip
$ unzip golang-markdown-previewer.zip
```

1.6 代码结构

克隆完成以后 golang-markdown-previewer 的目录结构如下：

```
golang-markdown-previewer
├── README.md
└── src
    ├── previewer
    │   ├── http_server.go
    │   ├── md_converter.go
    │   ├── previewer.go
    │   ├── template.go
    │   ├── watcher.go
    │   └── websocket.go
    └── sysm
        ├── http_server.go
        ├── sysm.go
        ├── template.go
        ├── watcher.go
        └── websocket.go
```

二、实验原理

2.1 Markdown 预览器的设计

本项目课中，我们将使用 go 语言 (http://www.shiyanlou.com/courses/11)编写一个 markdown 文件的实时预览器，它可以在浏览器实时预览使用任何文本编辑器正在编辑的 markdown 文件。

什么是 markdown 呢?

> Markdown 是一种轻量级标记语言，创始人为约翰·格鲁伯（John Gruber）。它允许人们“使用易读易写的纯文本格式编写文档，然后转换成有效的 XHTML(或者 HTML)文档”。

我们可以使用 markdown 编写纯文本的文件，然后通过软件将这些文本格式化为排版优美的 HTML 页面。实际上，本课程就是使用 markdown 进行编写。

预览器的工作原理是什么呢？其实非常简单：预览器会监控 markdown 文件的状态，如果检测到发生变化就将 markdown 文件格式化为 html 页面重新显示到浏览器上。所以，我们的预览器将包含：

- http 服务器：用于显示文本
- markdown 转换器： 将 markdown 文件转换为 html 页面

除此以外，为了实时预览，我们将使用 websocket 技术。

2.2 Websocket 浅析

我们的预览器会在浏览器中实时预览我们编辑的 markdown 文件。在浏览器中实现实时的响应，有两种方式，第一种是通过浏览器的轮询方式，浏览器端不断的向服务端请求数据（主要通过客户端 javascript），然后更新页面上的数据。不过如今我们可以使用 websocket 解决这类问题了，利用 websocket 可以在浏览器和服务器建立一个全双工的通道，这样服务端可以直接将新的数据发送给浏览器，浏览器在页面上更新这些数据即可。

什么是 Websocket?

> WebSocket 是 HTML5 开始提供的一种浏览器与服务器间进行全双工通讯的网络技术。WebSocket 通信协议于 2011 年被 IETF 定为标准 RFC 6455，WebSocketAPI 被 W3C 定为标准。在 WebSocket API 中，浏览器和服务器只需要做一个握手的动作，然后，浏览器和服务器之间就形成了一条快速通道。两者之间就直接可以数据互相传送。

2.2.1 Websocket 的协议转换

Websocket 是工作在 http 协议之上的，我们都知道 http 协议是无状态的，那浏览器和服务器是怎么样知道将 http 协议转换为 websocket 协议的呢?

在使用 wbesocket 的时候，浏览器向服务发送一个请求，表明它要将协议由 http 转为 websocket。客户端通过 http 头中的 Upgrade 属性来表达这一请求，下面是一个请求 http 头的示例:

```
GET ws://localhost:6060/1.md HTTP/1.1
Host: localhost:6060
Connection: Upgrade
Upgrade: websocket
Origin: http://localhost:6060
Sec-WebSocket-Version: 13
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/38.0.2125.122 Safar
i/537.36
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8,zh-CN;q=0.6,zh;q=0.4,zh-TW;q=0.2,ja;q=0.2
Sec-WebSocket-Key: 2DbWGFVjcauSVjY1+/2neQ==
Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
```

如果服务器支持 websocket 协议，同样通过 http 头中的 Upgrade 属性来表示同意进行协议的转换，下面是一个响应 http 头的示例：

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: w6rXmjUVLpQxK3rHk25Va9h7Y2w=
```

2.2.2 基于 Websocket 协议的实时系统监控工具

在 Go 语言中，我们可以使用 `github.com/gorilla/websocket` 来实现 websocket 协议。在实际应用中，我们判断每一个请求的 header 中 `Upgrade` 是否包含 “`websocket`” 字符串， `Connection` 字段中是否包含 “`Upgrade`” 字符串，如果都包含的话这就是一个 websocket 请求。

下面让我们通过一个练习来学习下 go 语言中的 websocket 的实现，在这个练习中，我将开发一个简单的基于 websocket 的服务器监控软件 `sysm` ，它将系统的 CPU 和内存使用率实时显示在浏览器页面上，浏览器端的绘图我将使用 Highcharts 进行绘图。关于 HighCharts 的更多信息可以参考：HighCharts 官方文档 (http://www.highcharts.com/)。

总的来说 `sysm` 的工作流程如下： 动手实践是学习 IT 技术最有效的方式! 开始实验

1. sysm 程序启动，开始监听端口；
2. sysm 程序判断 http 请求是否是 websocket 请求，如果不是则发送页面资源，页面中包含使用 HighCharts 绘图的代码以及 websocket 连接代码；
3. 当浏览器加载完毕 sysm 发送的静态页面后，浏览器开始执行 websocket 连接代码；
4. sysm 服务端检测到 http 请求头是 websocket 连接，然后开始发送 CPU、内存使用率数据；
5. 此时浏览器页面中的 websocket 连接，收到数据，开始进行绘图工作；

可以看到整个逻辑比较简单，针对每一个功能模块，我们可以将源代码划分为以下几个部分：

- **http_server.go**

  实现了一个功能简单的 http 服务器，该服务器针对请求是否是 websocket 请求做出相应的处理；

- **websocket.go**

  基于 github.com/gorilla/websocket 模块实现了基本的 websocket 操作；

- **template.go**

  主要是实现了对页面资源的操作，在浏览器对服务器第一次发起请求的时候，我们可以使用 Template 将页面资源发送给浏览器；

- **watcher.go**

  主要实现了系统 CPU 和内存使用率的监控；

- **sysm.go**

  封装了以上代码以方便对外使用。

2.2.3 sysm 的代码实现

在这一节中，我们主要以源代码注释的方式讲解 sysm 的核心代码，我们并没有列出所有的代码，省略的代码使用 `//...` 表示， 所有的源代码可以通过 git 克隆本课程 github 项目的方式获得，具体方法请查看本课程开头说明。

- http_server.go

```
//...
// 定义一个 HTTPServer 结构体
type HTTPServer struct {
    port     int
    // 嵌入了一个 net.Listener 接口，任何满足该接口的类型都可以嵌入该字段
    listener net.Listener
}

// 构造方法
func NewHTTPServer(port int) *HTTPServer {
    return &HTTPServer{port, nil}
}

//...
// 使用 http.Server 类型建立了一个 web 服务器，并默认使用 ServeHTTP 方法作为默认 handler
func (s *HTTPServer) ListenAndServe() {
    var err error
    server := &http.Server{
        Addr:           s.Addr(),
        Handler:        s,
        ReadTimeout:    10 * time.Second,
        WriteTimeout:   10 * time.Second,
        MaxHeaderBytes: 1 << 20,
    }
    // 使用 net.Listen 进行端口监听
    s.listener, err = net.Listen("tcp", s.Addr())
    if err != nil {
        panic(err)
    }
    // 服务器可以说接收并处理请求
    server.Serve(s.listener)
}

// 默认的请求处理函数
func (s *HTTPServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    path := r.URL.Path[1:] // remove '/'
    if path == "ping" {
        w.Write([]byte("pong"))
        fmt.Println("accept connection")
    } else if isWebsocketRequest(r) { //判断是否是 websocket 请求
        fmt.Println("websocket connect..")
        NewWebsocket().Serve(w, r) // 创建 websocket 连接，发送数据
    } else {
        Template(w, s.port)
    }
}
```

- watcher.go

```
//...
// 用于存储系统 CPU 和内存使用率
type Info struct {
    Cpu  float64 `json:"cpu"`
    Mem  float64 `json:"mem"`
    Time int64   `json:"time"`
}

type Watcher struct {
    ticker *time.Ticker
    stop   chan bool
    Data   chan *[]byte
}

// 构造函数
func NewWatcher() *Watcher {
    return &Watcher{nil, nil, make(chan *[]byte)}
}

func (w *Watcher) Start() {
    go func() {
        // 定时器
        w.ticker = time.NewTicker(time.Millisecond * WatcherInterval)
        defer w.ticker.Stop()
        w.stop = make(chan bool)
        for {
            select {
            case <-w.stop:
                return
            // 时间周期到达
            case <-w.ticker.C:
                // 使用 github.com/cloudfoundry/gosigar 进行 CPU 和内存数据采集
                var info Info
                cpu := sigar.Cpu{}
                cpu.Get()
                info.Cpu = float64(100) - float64(cpu.Idle*100)/float64(cpu.Total())

                mem := sigar.Mem{}
                mem.Get()
                info.Mem = float64(100) - float64(mem.Free)/float64(mem.Total)

                info.Time = time.Now().UnixNano() / 1000000

                // 数据转换为 json 后，发送到数据 channel
                data, _ := json.Marshal(info)
                w.Data <- &data
            }
        }
    }()
}
```

- websocket.go

```
// 将从`watcher`中获取到的数据，通过 websocket 发送到浏览器端
func (ws *Websocket) Writer(c *goWs.Conn, closed <-chan bool) {
    ws.watcher.Start()
    defer ws.watcher.Stop()
    defer c.Close()
    for {
        select {
        // 接收系统 CPU 和内存数据
        case data := <-ws.watcher.Data:
            c.SetWriteDeadline(time.Now().Add(WriteTimeout))
            // 发送数据
            err := c.WriteMessage(goWs.TextMessage, *data)
            if err != nil {
                return
            }
        case <-closed:
            return
        }
    }
}

func (ws *Websocket) Serve(w http.ResponseWriter, r *http.Request) {
    if r.Method != "GET" {
        http.Error(w, "Method not allowed", 405)
        return
    }
    // 将 http 协议转换成 websocket 协议， 并开始数据读写操作
    sock, err := upgrader.Upgrade(w, r, nil)
    if err != nil {
        fmt.Println("Can't connect to websocket")
        return
    }

    closed := make(chan bool)

    go ws.Reader(sock, closed)
    ws.Writer(sock, closed)
}
```

- sysm.go

```
package sysm

type Sysm struct {
    port       int
    httpServer *HTTPServer
    stop       chan bool
}
// 构造函数
func NewSysm(port int) *Sysm {
    return &Sysm{port, nil, make(chan bool)}
}
// 开始运行
func (s *Sysm) Run() {
    s.httpServer = NewHTTPServer(s.port)
    s.httpServer.Listen()
    <-s.stop
}

func (s *Sysm) Stop() {
    s.httpServer.Stop()
    s.stop <- true
}
```

如果已经克隆了本课程的源代码，这可以通过以下方式运行 sysm 程序（假如本课程源码克隆到 /home/shiyanlou/golang-markdown-previewer 目录）：

1. 设置环境变量和安装依赖包

```
$ cd ~
$ export GOPATH=/home/shiyanlou/golang-markdown-previewer:$GOPATH
$ go get github.com/shiyanlou/gosigar
$ go get github.com/shiyanlou/websocket
$ mkdir /home/shiyanlou/code
$ cd code
```

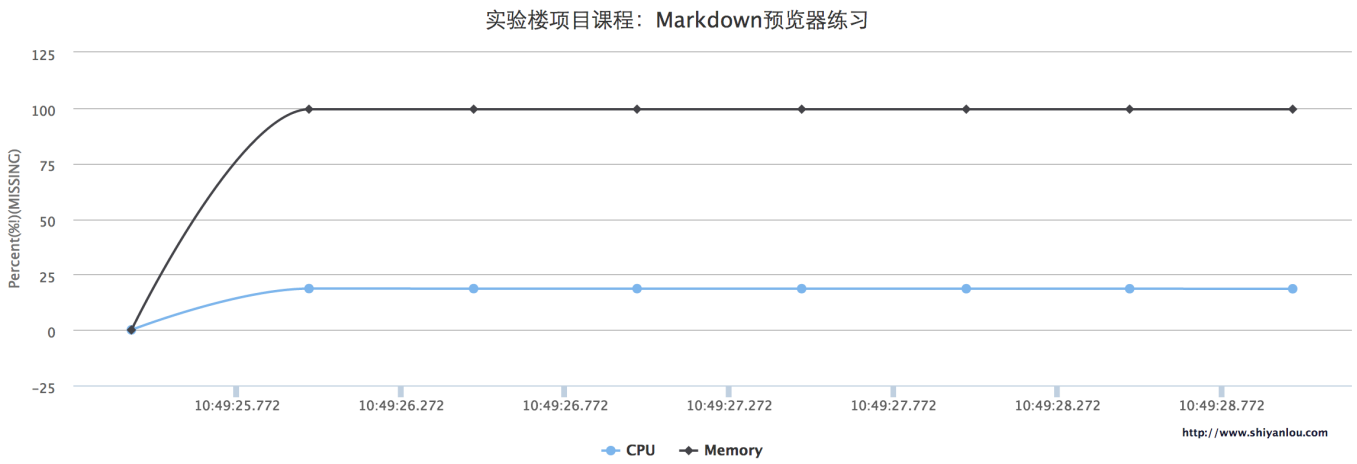

2. 创建源文件 sysm-main.go ，输入以下代码：

```
package main

import (
    "sysm"
)

func main() {

    sysm := sysm.NewSysm(8088)
    sysm.Run()
}
```

3. 通过 `go run` 命令运行 `sysm` 程序

```
$ go run sysm-main.go
```

1. 在浏览器地址栏中，输入 `http://localhost:8088` ，查看查看程序运行效果，如下图：



四. Markdown 预览器的实现

1. 实现

经过上一节中的练习，markdown 预览器的实现就非常简单了，预览器大部分核心功能我们已经在上一节中实现，现在我们只需要其他代码判断 markdown 文件是否发生变化，如果发生变化则转换其内容发送到浏览器即可。那怎么样判断文件是否发生变化呢，只需要判断文件的时间就行了。所以相比较于 `sysm` 的实现，我们只需增加一部分处理 markdown 文件的源代码就可以实现 markdown 预览器了。

同样的，这里我们直接列出一些变化和增加的核心代码，以注释的方式进行讲解：

- `md_converter.go`： 用于将 markdown 内容转换为 html，标签

```
package previewer

// 使用 blackfriday 包 对 markdown 进行编译
import (
    "github.com/russross/blackfriday"
)

// 全局的 markdown 编译器
var MdConverter = NewMarkdownConverter()

type MarkdownConverter struct {
    convert func([]byte) []byte
}

// 构造函数
func NewMarkdownConverter() *MarkdownConverter {
    return &MarkdownConverter{blackfriday.MarkdownCommon}
}

// 使用 blackfriday 基础编译器
func (md *MarkdownConverter) UseBasic() {
    md.convert = blackfriday.MarkdownBasic
}

func (md *MarkdownConverter) Convert(raw []byte) []byte {
    return md.convert(raw)
}
```

相对于 sysm 程序，我们需要修改其他文件中的部分代码，部分代码说明如下：

watcher.go：需要对 **markdown** 文件内容是否变化做出判断

```
package previewer

//...

type DataChan struct {
    Raw chan *[]byte
    Req chan bool
}

type Watcher struct {
    path   string
    ticker *time.Ticker
    stop   chan bool
    C      *DataChan
}

func (w *Watcher) Start() {
    go func() {
        w.ticker = time.NewTicker(time.Millisecond * WatcherInterval)
        defer w.ticker.Stop()
        w.stop = make(chan bool)
        var currentTimestamp int64
        for {
            select {
            // 如果接收到退出消息，则退出
            case <-w.stop:
                return
            case <-w.ticker.C:
                reload := false
                select {
                case <-w.C.Req:
                    reload = true
                default:
                }
                // 获取文件状态
                info, err := os.Stat(w.path)
                if err != nil {
                    continue
                }
                // 获取文件变化的时间戳
                timestamp := info.ModTime().Unix()
                if currentTimestamp < timestamp || reload {
                    currentTimestamp = timestamp
                    // 如果上一次的时间戳小于当前时间戳，则刷新数据
                    raw, err := ioutil.ReadFile(w.path)
                    if err != nil {
                        continue
                    }

                    w.C.Raw <- &raw
                }
            }
        }
    }()
}
```

websocket.go： 需要将读取的 **markdown** 内容转换为 **html** 内容，然后发送到浏览器。

```
//...
func (ws *Websocket) Writer(c *goWs.Conn, closed <-chan bool) {
    ws.watcher.Start()
    defer ws.watcher.Stop()
    defer c.Close()
    for {
        select {
        // 等待接收 markdown 文件原始内容
        case data := <-ws.watcher.C.Raw:
            c.SetWriteDeadline(time.Now().Add(WriteTimeout))
            // 将 markdown 内容转换为 html 内容，然后发送到浏览器
            err := c.WriteMessage(goWs.TextMessage, MdConverter.Convert(*data))
            if err != nil {
                return
            }
        case <-closed:
            return
        }
    }
}
//...
```

template.go： 用于将 **html** 页面到浏览器，发生在浏览器第一次发起请求的时候。

```
func Template(w http.ResponseWriter, filepath string, port int) {
    //...
    templateStr := fmt.Sprintf(`
<!DOCTYPE html>
<html>
<head>
  <meta charset='UTF-8' />
  <title>%[1]s</title>
  %[3]s
</head>
<body>
  <div id='md' class='markdown-body'></div>
  <script>
     <!-- 浏览器执行以下代码，发起 websocket 请求，获取数据-->
    (function () {
      var markdown = document.getElementById("md");
      var conn = new WebSocket("ws://localhost:%[2]d/%[1]s");
      conn.onmessage = function (evt) {
        markdown.innerHTML = evt.data;
      };
    })();
  </script>
</body>`, filepath, port, style)

    var (
        t   *template.Template
        err error
    )

    if t, err = template.New("template").Parse(templateStr); err != nil {
        panic(err)
    }
    // 将页面内容发送给浏览器
    if err = t.Execute(w, nil); err != nil {
        panic(err)
    }
}
```

同样的我们将参照 sysm.go 源文件，编写 previewer.go 代码：

- previewer.go

```
package previewer

import (
    "fmt"
    "github.com/skratchdot/open-golang/open"
)

const (
    MarkdownChanSize = 3
    Version          = "0.1"
)

func NewPreviewer(port int) *Previewer {
    return &Previewer{port, nil, make(chan bool)}
}

type Previewer struct {
    port       int
    httpServer *HTTPServer
    stop       chan bool
}

func (p *Previewer) Run(files ...string) {
    p.httpServer = NewHTTPServer(p.port)
    p.httpServer.Listen()
    // 使用 open 包，打开相应的链接
    for _, file := range files {
        addr := fmt.Sprintf("http://localhost:%d/%s", p.port, file)
        open.Run(addr)
    }

    <-p.stop
}

func (p *Previewer) UseBasic() {
    MdConverter.UseBasic()
}
```

到这里，我们对整个预览器的代码就比较熟悉啦，更多的代码细节请查看源码源码。

2. 运行程序

我们终于可以运行程序啦。当然运行之前我们需要做一些工作。

1. 设置环境变量和安装依赖包

```
$ cd ~
$ export GOPATH=/home/shiyanlou/golang-markdown-previewer
$ go get github.com/shiyanlou/open-golang/
$ go get github.com/shiyanlou/open-golang/open
$ go get github.com/russross/blackfriday
$ mkdir /shiyanlou/home/previewer
$ cd previewer
```

在 `$ go get github.com/shiyanlou/open-golang/open` 这一步有报错是正常的。

1. 在 `/home/shiyanlou/previewer` 目录中创建源文件 `previewer-main.go` ，输入以下代码：

```
package main

import (
    "os"
    "previewer"
)

func main() {

    previewer := previewer.NewPreviewer(8089)
    previewer.UseBasic()
    previewer.Run(os.Args...)
}
```

2. 通过 `go build` 编译 `previewer` 程序

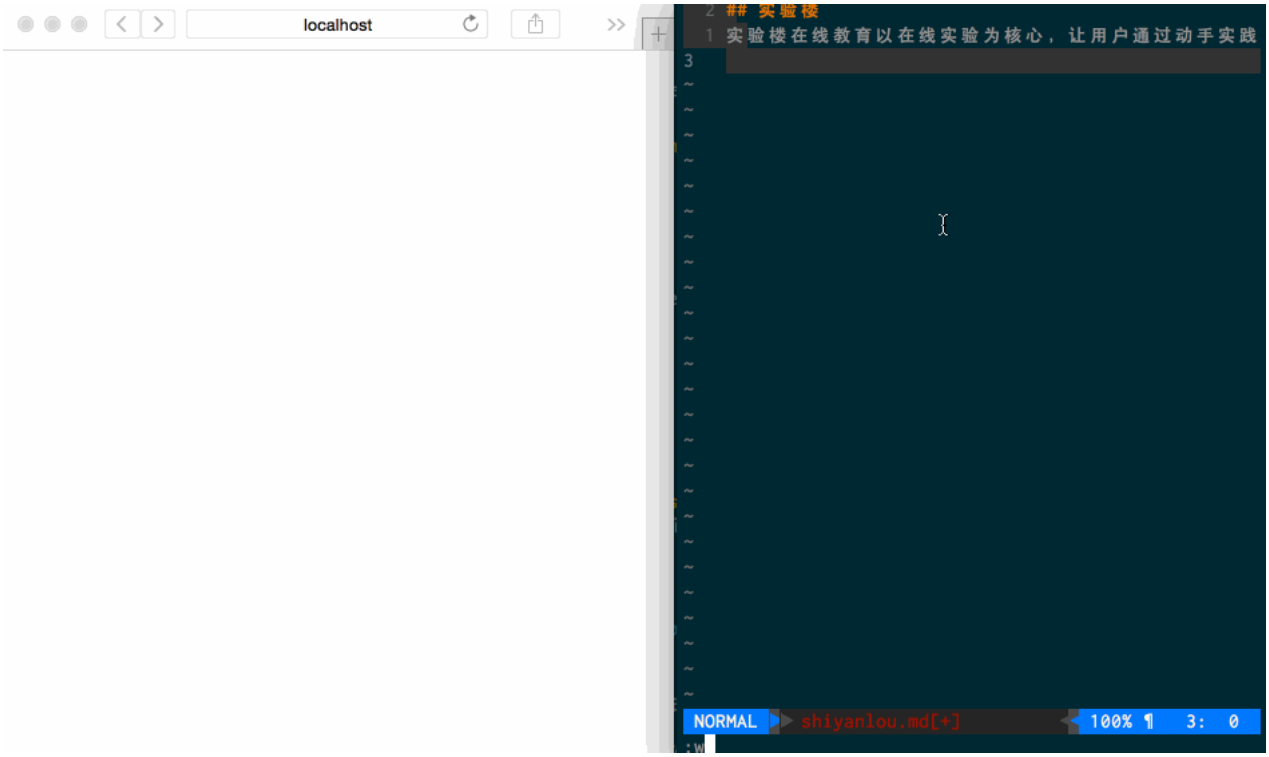
```
$ go build -o previewer previewer-main.go
```

3. 以上命令运行后会再当前目录下生产 previewer 程序，假如当前目录下有 shiyanlou.md ， 我们正在其他编辑器中编辑该文件，那么我们可以通过以下方式进行实时预览：

```
$ ./previewer shiyanlou.md
```

以上命令运行后，会再浏览器中打开 shiyanlou.md 文件，如果我们编辑该文件，就可以在浏览器中看到实时预览了。

如下图：



到目前为止，我们就实现了一个简单的 markdown 预览器啦，也是非常实用的一个工具。其实我们再添加一些使用的功能，比如支持自定义 markdown 显示样式（css 文件），实现对 markdown 文件的管理等，这些功能就等着大家去实践了。

参考

[1]. Wiki 百科 Markdown (http://zh.wikipedia.org/wiki/Markdown)

[2]. Wiki 百科 Websocket (http://zh.wikipedia.org/wiki/WebSocket)

[3]. RFC6455 Websocket 协议标准 (http://tools.ietf.org/html/rfc6455)

[4]. orange-cat markdown previewer (https://github.com/noraesae/orange-cat.git)