WEB 开发与应用

设计报告

基于 gin 的 chatgpt 聊天室系统设计与实现

专业名称 _____

班级学号 _____

学生姓名 _____

指导教师 _____

设计时间 2023.6.2-2023.6.16

目录

1　需求分析

该程序的需求是实现一个简单的聊天室应用，用户可以通过网页界面进行注册、登录、发送消息、接收消息等操作。主要提供聊天室服务和集成 OpenAI GPT-3 模型实现 AI 自动聊天回复功能，聊天室应用包括以下具体功能：

1) 用户注册和登录功能：用户可以通过注册页面进行注册，或者通过登录页面登录到聊天室。
2) 实时聊天功能：已登录的用户可以在聊天室内发送消息，其他用户可以实时接收到消息。在发送消息时，用户可以输入文本信息并发送，也可以与 AI 进行聊天，AI 会返回回答。聊天记录将保存在数据库中。
3) 在线用户列表：聊天室界面将显示当前在线的所有用户的列表。
4) 聊天室界面：聊天室的界面需要具有良好的用户体验，包括输入框、发送按钮、消息显示框、在线用户列表等元素。
5) 数据库支持：程序需要使用 MySQL 数据库来存储用户信息和聊天记录。
6) WebSocket 支持：程序需要使用 WebSocket 技术来实现实时聊天功能。
7) AI 对话支持：用户可以选择与 AI 进行聊天，AI 返回的回答将显示在聊天室中。
8) 前端页面：前端永远走在后端之前，要想写一个好程序，首先要看起来像一个好程序。聊天室需要一个漂亮的前端页面来增强用户体验。

2　设计方案

本系统拟采用 Gin 和 websocket 作为后台开发工具，使用 html、css 和 javascript 开发前台，MySQL 作为数据存储的工具，目的是实现一个基于 WebSocket 和 OpenAI GPT-3.5-turbo API 的聊天室。

1) 用户认证：这个聊天室提供了用户注册和登录的功能。用户可以输入用户名和密码进行注册。注册成功后，用户可以使用注册的用户名和密码进行登录。
2) WebSocket 通信：这个聊天室使用 WebSocket 协议来实现实时通信。当用户登录后，服务器会为其创建一个 WebSocket 连接。用户可以通过这个连接发送消息，服务器也可以通过这个连接向用户发送消息。
3) 消息广播：当服务器接收到一个用户的消息后，会将这个消息广播给所有在线的用户。这是通过遍历所有的 WebSocket 连接并向每个连接发送消息实现的。
4) 与 OpenAI GPT-3.5-turbo API 的交互：每当用户发送一条消息，服务器都会将这条消息发送给 OpenAI GPT-3.5-turbo API，并获取到 AI 的回复。然后，服务器会将 AI 的回复也广播给所有在线的用户。
5) 用户和消息的存储：这个聊天室使用 MySQL 数据库来存储用户信息和消息。当用户注册时，会将用户的信息保存到数据库中。当用户发送消息时，也会将消息保存到数据库中。同时，AI 的回复也会自动保存到数据库中。
6) 在线用户列表：服务器会维护一个在线用户列表，当有用户连接或断开连接时，都会更新这个列表，并将更新后的列表发送给所有在线的用户。

3　数据库设计

在系统开发中，数据库设计是不可或缺的核心技术。一个良好的数据库设计模式，可以为开发者提供友好的开发环境，还能够提供高效的数据存储和操作效率，简化开发过程，提高系统的安全性和性能。由于 MySQL 具有高速处理数据、支持并发访问、占用资源较小等特点，本系统使用 MySQL 作为开发中的数据库。

在这个聊天室系统中，至少需要两个数据库表：用户表（users）和消息表（messages）。

1) 用户表（users）：该表用于存储用户的登录信息。在代码中，用户表应该包含以下字段：

   - `username`：用户名，用于登录，应该设置为唯一键，不允许重复。
   - `password`：用户的密码，用于登录验证。
   - `ip`（ip 地址）。
   - `email`（用户邮箱）。
   - `created_at`（用户创建时间）。

2) 消息表（messages）：该表用于存储用户在聊天室中发送的所有消息。在代码中，消息表应该包含以下字段：

   - `sender`：消息的发送者，应该与用户表的 `username` 字段相对应。
   - `message`：消息内容。
   - `sent_at`（消息发送时间）等。

4　主要功能模块实现

1. WebSocket Server：这是整个应用的核心部分。它管理所有客户端的连接，并负责向客户端广播消息。当一个新的 WebSocket 连接被接受时，该连接将被添加到 `clients` map 中。然后，一个新的 goroutine 将开始监听来自这个连接的消息。这些消息将被发送到全局消息通道 `broadcast`。主线程持续监听 `broadcast` 通道，并将每个收到的消息广播到所有客户端。

2. User and AI Messaging：这部分代码处理用户的消息，并通过 GPT-3 生成 AI 的回应。每当收到一个用户消息时，应用将该消息添加到全局消息通道 `broadcast`，然后调用 `chatWithGPT` 函数获取 AI 的回应。这个函数创建一个请求，发送到 OpenAI 的 API，然后返回 AI 的消息。这个消息也会被添加到 `broadcast`。

3. Database Integration：这部分代码用于将用户和 AI 的对话存储在数据库中。每当 `broadcast` 收到一个消息，这个消息就会被添加到数据库中。这是通过调用 `addMessageToDB` 函数实现的，该函数接受一个 `UserMessage` 结构体，并将其添加到数据库中。

4. Web Server and Routing：这部分代码实现了 HTTP 服务器和路由。服务器监听特定的端口，并使用一组路由处理 HTTP 请求。这些路由允许用户建立 WebSocket 连接（`/ws` 路由），查看聊天记录（`/history` 路由），以及获取静态文件（`/` 路由）。

这就是这个聊天应用的主要功能模块。整个应用是在 Go 语言中实现的，使用了 Gorilla WebSocket 库、OpenAI API、以及 MySQL 数据库。

5　软件测试

为了高效、高质量的完成一个软件项目，测试要求必须完善且适度，否则，如果将系统软件投入到运行中会让使用者承担很大的危险，付出很大的代价。系统开发中的测试只能够向开发者证明错误的存在，但是不是能够表明隐藏的错误是否会造成很大的影响。

5.1　数据库测试

测试数据库时需要考虑以下多个方面：

(1) 数据库设计的测试

对于数据库表结构设计，要确保表结构合理、规范，以及正确地使用了索引。

(2) SQL 语句效率测试

对于 SQL 查询性能，通过执行计划分析查询性能，优化查询语句。可以针对数据库访问层编写测试用例，确保查询和修改数据的逻辑是正确的。还可以针对关键的 SQL 语句和数据库操作，进行性能基准测试，以及不同数据量下的性能测试。

(3) 数据库并发性能测试

在高并发场景下，数据库性能是否满足需求。可以进行压力测试，模拟高并发场景，测试数据库是否能够在高并发下正常工作，性能是否满足需求。

5.2　功能测试

在功能测试的部分主要是对系统的实际操作进行测试，主要作用是对各个功能的使用情况的完善性和可操作性进行检查，使用这种方法便于测试人员及时更改和完善代码。

(1) 单元测试：使用适当的测试框架（如 Go's testing package）来编写和运行单元测试。在程序的 main_test.go 中编写测试函数来测试数据库连接、用户登录和注册逻辑等。用到 httpmock 和 sqlmock 等库，模拟用户输入、HTTP 请求和响应、数据库连接，以及 WebSocket 连接。

(2) 集成测试：编写集成测试来测试整个系统的功能。在 main_test.go 中编写

测试用例来模拟用户登录、发送消息、接收 AI 回复等操作，并验证系统的行为是否符合预期。

5.3　测试过程中遇到的问题及解决方法

(1) 界面无法显示消息

在进行消息广播测试时，无法显示消息，而客户端服务器是可以正常接受消息的，经过排查确定是由于前后端定义的消息格式不同造成的，同意消息的数据结构后，可以将消息正确地发送给所有连接的客户端，并在页面上正常显示。

(2) 无法与OpenAI API的进行交互

首先是因为所用的 chatgpt 模型已经不在维护，于是查看官方文档，改代码适配其他模型。

随后在请求 http 链接时报错：

2023/05/09 19:47:12 HTTP request failed: Post "api.openai.com dial tcp [2a03:2880:f127:283:face:b00c:0:25de]:443: connectex: A connection attempt failed because the connected party did not properly respond after a period of time, or established connection failed because connected host has failed to respond.

解决方案：在程序代码中配置代理服务器，成功链接；

然而在成功链接之后，仍然无法显示消息，报错显示：

```
Response Status: 429 Too Many Requests
Response Body: {
    "error": {
        "message": "You exceeded your current quota, please check your plan and billing details.",
        "type": "insufficient_quota",
        "param": null,
        "code": null
    }
```

由于openai api key额度到期，无法再进行服务请求，只得进行充值，终于在网页中显示了openai 的回答。

(3) 无法显示多行内容

错误描述：在服务器和数据库中多行文本确实被完整保存，然而在前端却只能显示文本的第一行。

问题分析与解决：未解决

5.4　系统尚存在的不足

1. 目前的程序仍然只是一个最小可行性产品，有待完善和优化。未来打算加入错误处理、日志记录、安全性方面的改进等。

2. 对于数据库设计应考虑更多的因素，比如数据的一致性、完整性、安全性等，需要设计更复杂的表结构和更完善的字段。

3. 密码的存储应使用加密方法以保证安全性。

4. 代码重构和优化。

6 结论与心得体会

编写代码是一个复杂且富有挑战性的过程，同时为我带来了许多满足感和成就感，在这个过程中我学到了很多有用的方法论：

1. 测试驱动开发（TDD）：编写单元测试和集成测试可以帮助你确保代码的功能正常，也可以防止未来的更改破坏已有的功能。

2. 设计优先：在开始编码之前，花时间设计解决方案，确保你的代码组织有序，并使其更容易被他人理解和维护。

3. 持续学习和改进：技术和最佳实践总是在不断变化和发展。要养成阅读英文文档、阅读他人的代码等方式不断提高你的技能和知识。

4. 从 bug 中学习：在编写代码时，你可能会遇到各种问题和挑战。这些可能是编程错误，也可能是理解问题的困难，或者是设计决策的困扰。不管是哪种情况，你都可以把它们看作是学习和提高的机会。

5. 重构的技巧：在编写代码时时刻践行代码简洁之道。

附录：部分程序清单

以下是后端核心代码，前端、测试代码及其他请访问 github 仓库：https://github.com/ElandWoo/chatroom。

1. Mysql 服务

```sql
CREATE DATABASE chatroom;

USE chatroom;
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL
);

CREATE TABLE messages (
    id INT AUTO_INCREMENT PRIMARY KEY,
    sender VARCHAR(255) NOT NULL,
    message TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

USE chatroom;
SHOW TABLES;
SELECT * FROM users;
SELECT * FROM messages;
```

2. 广播

```go
func wshandler(w http.ResponseWriter, r *http.Request) {
    // 升级连接为 WebSocket
    conn, err := upgrader.Upgrade(w, r, nil)
    if err != nil {
        fmt.Printf("Failed to set websocket upgrade: %+v\n", err)
        return
    }

    // 在这个位置获取客户端IP地址和用户名
    ip := r.RemoteAddr
    cookie, err := r.Cookie("username")
    if err != nil {
        fmt.Println("Error getting username from cookie:", err)
        return
    }
    username := cookie.Value

    // 将新的 WebSocket 连接添加到映射中
    conns.Lock()
    conns.m[conn] = username
    conns.Unlock()

    // 在连接建立后发送在线用户列表
    sendUserList()
```

```
    defer func() {
        // 当连接断开时，从映射中移除
        conns.Lock()
        delete(conns.m, conn)
        conns.Unlock()

        // 在连接断开后发送在线用户列表
        sendUserList()
    }()

    // 读取消息信息
    for {
        _, msg, err := conn.ReadMessage()
        if err != nil {
            break
        }

        message := UserMessage{
            Sender:  username,
            Content: fmt.Sprintf("%s (%s): %s", username, ip, string(msg)),
        }

        // 将消息发送到全局消息通道
        msgChan <- message

        //fmt.Println(message.Content)
        //fmt.Println(chatWithGPT(message.Content).Content)
        chatWithGPT(string(msg))

    }

}
```

3. Websocket

```
func main() {
    // 打开一个数据库连接
    db, err := sql.Open("mysql", "user:password@tcp(localhost:3306)/chatroom")
    if err != nil {
        fmt.Println(err)
        return
    }

    // Close the database connection
    defer db.Close()

    // Test the database connection
    err = db.Ping()
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println("Connected to MySQL database!")
    // 打开一个路由
    router := gin.Default()
    router.LoadHTMLGlob("templates/*.html")
    router.Static("/static", "./static")

    router.GET("/", func(c *gin.Context) {
        c.HTML(http.StatusOK, "index.html", nil)
    })
```

```
    router.POST("/login", func(c *gin.Context) {
        username := c.PostForm("username")
        password := c.PostForm("password")

        if username == "" || password == "" {
            c.JSON(http.StatusBadRequest, gin.H{"error": "Username and password are required"})
            return
        }

        // 查询用户信息
        var storedPassword string
        query := "SELECT password FROM users WHERE username = ?"
        err := db.QueryRow(query, username).Scan(&storedPassword)
        if err != nil {
            if err == sql.ErrNoRows {
                c.JSON(http.StatusUnauthorized, gin.H{"error": "Invalid username or password"})
            } else {
                c.JSON(http.StatusInternalServerError, gin.H{"error": "Failed to query user information"})
            }
            return
        }

        // 验证用户密码
        if password != storedPassword {
            c.JSON(http.StatusUnauthorized, gin.H{"error": "Invalid username or password"})
            return
        }
```

```
        // 设置Cookie
        usernameCookie := http.Cookie{
            Name:  "username",
            Value: username,
            Path:  "/",
        }
        http.SetCookie(c.Writer, &usernameCookie)

        c.Redirect(http.StatusMovedPermanently, "/chat")
    })
```

```
    // sign up  Interface
    router.GET("/register", func(c *gin.Context) {
        c.HTML(http.StatusOK, "register.html", nil)
    })

    router.POST("/register", func(c *gin.Context) {
        username := c.PostForm("username")
        password := c.PostForm("password")
        confirm_password := c.PostForm("confirm_password")
```

```
        // 判断用户名和密码是否为空
        if username == "" || password == "" || confirm_password == "" {
            c.JSON(http.StatusBadRequest, gin.H{"error": "Username and password are required"})
            return
        }
        // 判断密码是否一致
        if password != confirm_password {
            c.JSON(http.StatusBadRequest, gin.H{"error": "Passwords do not match"})
            return
        }

        // 将用户注册信息保存到数据库中
        insertQuery := "INSERT INTO users (username, password) VALUES (?, ?);"
        _, err := db.Exec(insertQuery, username, password)
        if err != nil {
            c.JSON(http.StatusInternalServerError, gin.H{"error": "Failed to register user"})
            return
        }
```

```
        // 设置Cookie
        usernameCookie := http.Cookie{
            Name:  "username",
            Value: username,
            Path:  "/",
        }
        http.SetCookie(c.Writer, &usernameCookie)
        // 注册成功，跳转到聊天室页面
        c.Redirect(http.StatusMovedPermanently, "/chat")
    })

    // chat Interface
    router.GET("/chat", func(c *gin.Context) {
        c.HTML(http.StatusOK, "chat.html", nil)
    })

    //
    router.GET("/ws", func(c *gin.Context) {
        wshandler(c.Writer, c.Request)
    })
```

```
    // 在 main 函数中启动消息广播和消息保存到数据库的协程
    go func() {
        for {
            msg := <-msgChan

            // 将消息存储到数据库中
            insertQuery := "INSERT INTO messages (sender, message) VALUES (?, ?)"
            _, err := db.Exec(insertQuery, msg.Sender, msg.Content)
            if err != nil {
                fmt.Printf("Failed to insert message to database: %v", err)
                continue
            }

            // 广播消息给所有连接的 WebSocket 客户端
            conns.RLock()
            for conn := range conns.m {
                if err := conn.WriteMessage(websocket.TextMessage, []byte(msg.Content));
err != nil {
                    fmt.Printf("Failed to broadcast message: %+v\n", err)
                }
            }
            conns.RUnlock()
        }
    }()

    router.Run(":8080")
}
```

4. 显示在线列表

```
func sendUserList() {
    conns.RLock()
    defer conns.RUnlock()

    var userList []string
    for _, username := range conns.m {
        userList = append(userList, username)
    }

    userListMessage := strings.Join(userList, ", ")
    for conn := range conns.m {
        if err := conn.WriteMessage(websocket.TextMessage, []byte("Online users:
"+userListMessage)); err != nil {
            fmt.Printf("Failed to send user list: %+v\n", err)
        }
    }
}
```

5. Chatgpt 服务

使用 OpenAI API 进行聊天，并利用 Golang 编写代码。使用 OpenAI API 时，需要提供 API 密钥和请求体数据。请求体数据包含一些关键信息，例如要使用的模型、聊天的消息等等。在这个示例中，将使用`gpt-3.5-turbo`模型进行聊天，请求体数据包含了一条来自用户的消息`"Hello!"`。

config.go

```go
package main

const (
    OpenAIURL    = "https://api.openai.com/v1/chat/completions"
    OpenAIAPIKey = ""                     // Fill with your OpenAI API Key
    ProxyURL     = "http://localhost:7890" // Replace with your proxy server URL
)
```

main.go

```go
package main

import (
    "bytes"
    "encoding/json"
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
    "net/url"
)

type Message struct {
    Role    string `json:"role"`
    Content string `json:"content"`
}

type RequestBody struct {
    Model    string    `json:"model"`
    Messages []Message `json:"messages"`
}
```

```go
type ResponseBody struct {
    ID      string `json:"id"`
    Object  string `json:"object"`
    Created int    `json:"created"`
    Choices []struct {
        Index        int    `json:"index"`
        FinishReason string `json:"finish_reason"`
        Message      Message
    } `json:"choices"`
    Usage struct {
        PromptTokens     int `json:"prompt_tokens"`
        CompletionTokens int `json:"completion_tokens"`
        TotalTokens      int `json:"total_tokens"`
    } `json:"usage"`
}

func createRequestBody() RequestBody {
    return RequestBody{"gpt-3.5-turbo", []Message{{"user", "Hello!"}}}
}

func createRequest(payload []byte) *http.Request {
    req, err := http.NewRequest("POST", OpenAIURL, bytes.NewBuffer(payload))
    if err != nil {
        log.Fatal(err)
    }
    req.Header.Set("Content-Type", "application/json")
    req.Header.Set("Authorization", "Bearer "+OpenAIAPIKey)
    return req
}
```

```go

func createClient() *http.Client {
    proxyURL, err := url.Parse(ProxyURL)
    if err != nil {
        log.Fatal(err)
    }
    return &http.Client{Transport: &http.Transport{Proxy: http.ProxyURL(proxyURL)}}
}

func printResponseBody(respBody ResponseBody) {
    fmt.Printf("ID: %s\nObject: %s\nCreated: %d\n", respBody.ID, respBody.Object, respBody.Created)
    for i, choice := range respBody.Choices {
        fmt.Printf("Choice %d:\n  Index: %d\n  Finish Reason: %s\n  Message Role: %s\n Message Content: %s\n", i, choice.Index, choice.FinishReason, choice.Message.Role, choice.Message.Content)
    }
    fmt.Printf("Usage:\n  Prompt Tokens: %d\n  Completion Tokens: %d\n  Total Tokens: %d\n", respBody.Usage.PromptTokens, respBody.Usage.CompletionTokens, respBody.Usage.TotalTokens)
}
```

```go

func main() {
    reqBody := createRequestBody()
    payload, err := json.Marshal(reqBody)
    if err != nil {
        log.Fatal(err)
    }

    client := createClient()
    resp, err := client.Do(createRequest(payload))
    if err != nil {
        log.Fatal(err)
    }
    defer resp.Body.Close()

    body, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        log.Fatal(err)
    }

    var respBody ResponseBody
    err = json.Unmarshal(body, &respBody)
    if err != nil {
        log.Fatal(err)
    }

    printResponseBody(respBody)
}
```