



北京邮电大学

Beijing University of Posts and Telecommunications

程序设计实践 上机实验

[一种领域特定脚本语言的解释器的设计与实现——DSL]

学院：计算机学院
2020211376 马天成
2022 年 12 月 1 日

北京邮电大学《程序设计课程设计》课程实验报告

[illegible]

注：评语要体现每个学生的工作情况，可以加页。

目录

1. 实验目的和要求.....	4
1.1 实验目的.....	4
1.2 实验要求.....	4
2. 程序设计.....	4
2.1 程序层级设计.....	4
2.2 程序模块设计.....	5
2.3 mian.go.....	6
2.4 src.....	6
2.4.1 router.....	6
2.4.2 controller.....	7
2.4.3 service.....	9
2.4.4 dao.....	11
2.4.5 output.....	14
2.4.6 test.....	14
2.5 templates.....	16
2.6 static.....	17
3. 综合分析.....	17
3.1 风格.....	17
3.2 设计和实现.....	18
3.3 接口.....	20
3.4 测试.....	22
3.5 记法.....	22
4. 改进方案.....	22

1. 实验目的和要求

1.1 实验目的

描述：领域特定语言（Domain Specific Language, DSL）可以提供一种相对简单的文法，用于特定领域的业务流程定制。本作业要求定义一个领域特定脚本语言，这个语言能够描述在线客服机器人（机器人客服是目前提升客服效率的重要技术，在银行、通信和商务等领域的复杂信息系统中有广泛的应用）的自动应答逻辑，并设计实现一个解释器解释执行这个脚本，可以根据用户的不同输入，根据脚本的逻辑设计给出相应的应答。

1.2 实验要求

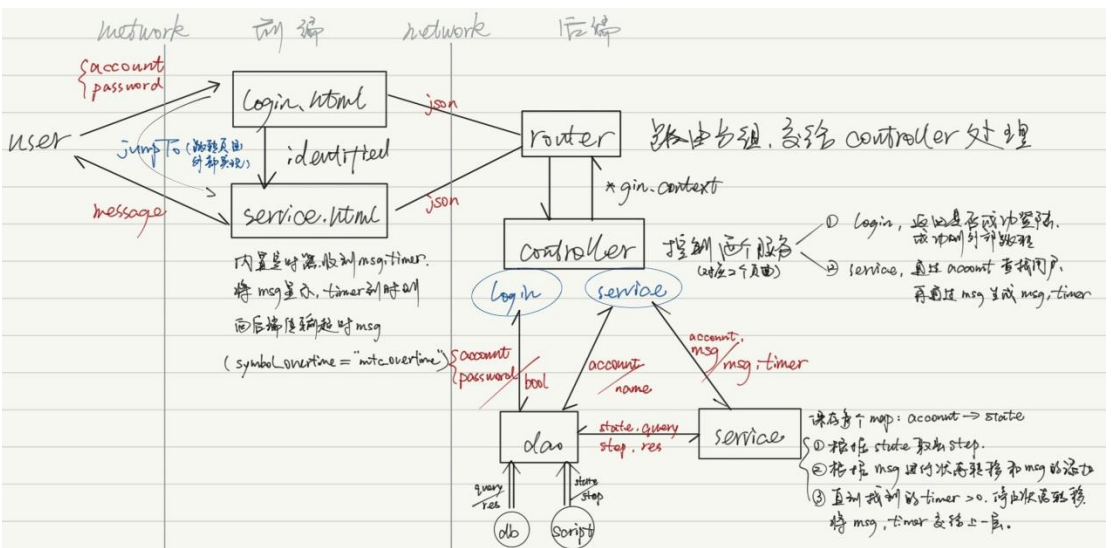
基本要求：

1. 脚本语言的语法可以自由定义，只要语义上满足描述客服机器人自动应答逻辑的要求。
2. 程序输入输出形式不限，可以简化为纯命令行界面。
3. 应该给出几种不同的脚本范例，对不同脚本范例解释器执行之后会有不同的行为表现。

2. 程序设计

这里介绍了程序层级设计，程序模块设计，各模块详细设计

2.1 程序层级设计



这里是应用整个的层级服务。用户端到前端到后端，典型的 BS 架构。

这里各个层级间的数据交换我也写在图表中了。具体的细节会在后面程序模块设计中详细介绍。

2.2 程序模块设计

首先程序的结构如下

目录结构描述

```
├── bin
│   ├── release.exe    // 发行版
│   ├── release_1.exe  // 女仆版
│   ├── release_2.exe  // 强尼银手版
│   └── script_test.exe // 脚本本地测试版
├── src
│   ├── router          // 路由层
│   │   └── router.go
│   ├── controller      // 控制层
│   │   └── controller.go
│   ├── service          // 服务层
│   │   └── service.go
│   ├── dao              // 数据访问层
│   │   ├── dao.go
│   │   ├── db.go
│   │   └── script.go
│   ├── output           // 输出
│   │   ├── print.go
│   │   └── log.go
│   ├── test             // 脚本本地测试
│   │   └── test.go
│   ├── main.go          // 源程序入口
│   ├── go.mod
│   ├── go.sum
│   └── scripts           // 脚本相关
│       ├── script.json
│       ├── script_1.json
│       └── script_2.json
├── script.json
├── script_1.json
├── script_2.json
├── test                  // 测试数据
│   ├── input            // 输入数据
│   │   ├── input_script.txt
│   │   ├── input_script_1.txt
│   │   └── input_script_2.txt
│   └── reference         // 对比数据
│       ├── input_script.txt
│       ├── input_script_1.txt
│       └── input_script_2.txt
├── static
│   ├── css
│   │   ├── login.css
│   │   └── service.css
│   ├── js
│   │   └── jquery-3.6.1.min.js
│   ├── pics
│   │   ├── Spike.jpg
│   │   └── 02.jpg
├── templates
│   ├── login.html
│   └── service.html
└── docs
    ├── README.md
    ├── Notation description.md
    ├── design.jpg
    └── Lab Report.pdf
```

可见，有以下几个文件夹：

- **bin**：所有可执行文件
- **src**：源码
- **main.go, go.mod, go.sum**：对应 src，管理程序和程序入口
- **scripts**：所有脚本文件和脚本测试文件
- **static**：所有前端静态资源
- **templates**：前端 html 模板
- **docs**：说明文档

2.3 mian.go

```
13
14 // main -----
15 func main() {
16     if len(os.Args) > 1 {
17         if os.Args[1] == "script_test" {
18             // Test mode: local test all scripts
19             fmt.Println( a...: "####SCRIPT_TEST MODE")
20             test.Test_scripts()
21         } else {
22             // init -----
23             output.Log_init()
24             // init service layer data(states)
25             output.Print( info: "Init service layer.")
26             service.Init()
27             // init dao layer data(script and db)
28             output.Print( info: "Init dao layer.")
29             dao.Init_db()
30             dao.Init_script(os.Args[1])
31
32             // Release mode: create router layer
33             fmt.Println( a...: "####RELEASE MODE")
34             router.CreateServer()
35         }
36     }
37 }
```

这里规定运行参数必定大于 1，而且第二个参数（下标 1）规定运行模式：

- **script_test**: 本地测试所有的脚本逻辑和脚本应答正确性；
- **脚本名称**: 用 **release mode** 运行该脚本。脚本不存在则会退出。此时这里会先进行 **service**，**dao** 层的初始化，然后进行相应的 **router** 服务。

2.4 src

这里是代码的核心。

这里极好的体现了分层设计，也很好体现了代码的复用性，以及分布式的思想。

其主要有四个层级：**router**，**controller**，**service**，**dao**。此外还有 **test** 进行测试脚本和 **output** 进行 **io** 调用。

2.4.1 router

▼ router
router.go

很简单的，**router** 就是利用 **gin** 框架进行路由创建。然后规定相应域名的 **get** 或者 **post** 提交给 **controller** 层处理。

```

9 func CreateServer() {
10     // engine
11     // create router engine
12     gin.SetMode(gin.ReleaseMode)
13     router := gin.Default()
14     // init assets for router( mainly two pages, discribed below )
15     router.LoadHTMLGlob( pattern: "./templates/*")
16     router.StaticFS( relativePath: "/static", http.Dir("./static"))
17
18     // login page
19     group_login := router.Group( relativePath: "/login")
20     group_login.GET( relativePath: "/", func(c *gin.Context) {
21         c.HTML(http.StatusOK, name: "login.html", gin.H{
22             "title": "login.html",
23         })
24     })
25     group_login.POST( relativePath: "/", controller.Call_login)
26
27     // service page
28     group_service := router.Group( relativePath: "/service")
29     group_service.GET( relativePath: "/", func(c *gin.Context) {
30         c.HTML(http.StatusOK, name: "service.html", gin.H{
31             "title": "service.html",
32         })
33     })
34     group_service.POST( relativePath: "/", controller.Call_service)
35
36     // port
37     http.ListenAndServe( addr: ":8848", router)
38 }

```

1. 初始化了 html 文件和 static 静态资源
2. 规定了 login/service page 对应的 Get 要给的页面，和 Post 要调用的 controller
3. 规定了 router 监听 8848 端口。

2.4.2 controller

● Controller.go

- 第一部分是 login 页面的服务：

```

13 // Login page: send to dao and generate response
14 func Call_login(c *gin.Context) {
15     // get login answer from dao layer
16     account := c.PostForm( key: "account")
17     password := c.PostForm( key: "password")
18     output.Print("Controller- recv: account=" + account + ", password=" + password)
19
20     // pre detect for account
21     if len(account) != 10 {
22         // account len = 10
23         c.JSON(http.StatusUnauthorized, gin.H{
24             "Identify": "Login fail",
25             "account": account,
26             "info": "Account.length should be equal to 10!",
27             "jumpTo": "/login",
28         })

```

```

29     } else {
30         // analyze answer
31         name := dao.Login(account, password)
32         if name == "" {
33             // identify error, return to login
34             output.Print( info: "Controller- send: Login fail!")
35             c.JSON(http.StatusUnauthorized, gin.H{
36                 "Identify": "Login fail",
37                 "account":  account,
38                 "info":     "Account or password is wrong!",
39                 "jumpTo":   "/login",
40             })
41
42         } else {
43             // identify success, send data and jump to new page
44             output.Print( info: "Controller- send: Login success!")
45             c.JSON(http.StatusOK, gin.H{
46                 "Identify": "Login success",
47                 "account":  account,
48                 "info":     name,
49                 "jumpTo":   "/service",
50             })
51         }
52     }

```

1. 账号不为 10 位，返回格式不正确
2. 账号为 10 位，dao 层查询数据库，但密码不匹配，返回验证错误
3. 密码匹配，返回姓名作为子段，以及跳转页面。

返回的内容即为 JSON 格式：

返回值内容也如上图所示。

```

"Identify":
"account":
"info":
"jumpTo":

```

- 第二部分是 service（客服）的服务。

```

54 // Service page: generate msg for return
55 func Call_service(c *gin.Context) {
56     text := c.PostForm( key: "text")
57     account := c.PostForm( key: "account")
58     output.Print("\nController- recv: " + account + " " + text)
59
60     // get name for current account(if this account exist)
61     name := dao.Query(account, query: "name")
62     if name == "" {
63         // illegal account
64         c.JSON(http.StatusOK, gin.H{"msg": "This account has no record.", "timer": 0})
65         return
66     } else {
67         // get msg & timer from script
68         msg, timer := service.Service(account, text)
69         output.Print("Controller- send: " + msg)
70         c.JSON(http.StatusOK, gin.H{"msg": msg, "timer": timer})
71     }
72 }

```

这里假定前提：登陆跳转后 service.html 的页面能正确接收到账号和姓名参数。随意篡改 url 中的参数，账号不能找到用户，会使得这里不提供服务。

而当用户合法，则会调用 service 层进行 msg 和 timer。

2.4.3 service

- Service.go

宏定义，全局变量 `states` 和 `service` 的初始化

很多宏定义：根据脚本文件获得的常量。全局变量：`state map[string]string`，账户到状态。

`Init` 给全局变量 `states` 一个 `map` 的创建。加入需要多次覆盖 `states`。`states` 他保存了账户和 `state` 字符串的映射关系。

`Clear` 提供给外部一个删除特定用户状态的接口。此函数仅供 `test` 模块使用。

因为测试账户为 2020211376，每次测试完一个脚本需要删除掉特定用户的状态。（因为测试没有退出指令的话会影响下一个脚本的起始状态，所以一定要清零）

`Refresh` 进行状态转移和更新。如果状态到 `exit` 则返回 `exit`。

```
9 // consts
10 const NULL = 0 // const timer 0
11 const symbol_devide = "&"
12 const symbol_name = "$name"
13 const symbol_amount = "$amount"
14 const symbol_balance = "$balance"
15 const symbol_overtime = "mtc_Overtime"
16 const state_start = "Start"
17 const state_exit = "Exit"
18
19 // init variable states(map)
20 func Init() {
21     states = make(map[string]string)
22 }
23
24 // clear user(for test currently)
25 func Clear_User(account string) {
26     delete(states, account)
27 }
28
29 // variable for store users' states, and change
30 var states map[string]string // when get users' post, generate response(msg) depend on this state
31 func Refresh_state(account string, new_state string) string {
32     delete(states, account)
33     // Exit, delete record
34     if new_state == state_exit : state_exit {
35         states[account] = new_state
36     }
37     output.Print("-----Service- User states: " + states[account])
38     return ""
39 }
40 }
```

service 函数

```
43 func Service(account string, input string) (string, int) {
44     // select user state record-----
45     _, ok := states[account]
46
47     // service block-----
48     var curStep dao.Step
49     msg := ""
50     timer := NULL
51
52     // error: no state for this account-----
53     if !ok {
54         output.Print( info: "-----Service- This account has no state, start serve.")
55         states[account] = state_start
56         output.Print("-----Service- User states: " + states[account])
57     }
```

```

59 // get current step-----
60 curStep = dao.Get_step(states[account])
61 output.Print("-----Service- Current: " + states[account])
62
63 // overtime-----
64 if strings.Contains(input, symbol_overtime) {
65     // convert state previously depend on silence
66     if Refresh_state(account, curStep.S_silence) == state_exit {
67         msg = Msg_append(account, msg, dao.Get_step(state_exit).S_speak)
68         return msg, timer
69     }
70 } else
71 // receive-----
72 {
73     // convert state previously depend on branch&input
74     flag_find := false
75     for k, v := range curStep.S_branch {
76         if strings.Contains(input, k) {
77             flag_find = true
78             if Refresh_state(account, v) == state_exit {
79                 msg = Msg_append(account, msg, dao.Get_step(state_exit).S_speak)
80                 return msg, timer
81             }
82         }
83     }
84     // don't find in branch, should jump to default
85     if !flag_find {
86         if Refresh_state(account, curStep.S_default) == state_exit {
87             msg = Msg_append(account, msg, dao.Get_step(state_exit).S_speak)
88             return msg, timer
89         }
90     }
91 }
92
93 // work until listen exists, stop-----
94 for true {
95     // get step and inner attributes(speak and listen)
96     curStep = dao.Get_step(states[account])
97     msg = Msg_append(account, msg, curStep.S_speak)
98     timer = curStep.S_listen
99     // if listen is NULL, convert current states
100    if timer != NULL {
101        break
102    }
103    // listen is NULL, convert current states
104    if Refresh_state(account, curStep.S_default) == state_exit {
105        msg = Msg_append(account, msg, dao.Get_step(state_exit).S_speak)
106        return msg, timer
107    }
108 }
109
110 return msg, timer
111 }

```

这里就是状态转移逻辑核心。

第一步先找这个 `account` 的状态是否存在。如果不存在，那么我们会返回错误提示，`timer=0`。

第二步我们会看是否是超市信号。如果是超时信号那么会先进行 `silence` 的转移。

第三步进行正常工作：如果当前步骤 `step` 的 `listen`（等待值为 0），那么拼接当前的 `speak` 到 `msg`，转移到 `default` 步骤；如果 `listen` 大于 0，则拼接当前的 `speak` 到 `msg`，发送当前 `step` 的 `timer`。

msg_append 函数

```
113 // msg_append
114 func Msg_append(account string, msg string, add string) string {
115     // convert add
116     output.Print("-----Service- Add: " + add)
117     if strings.Contains(add, symbol_name) {
118         output.Print( info: "-----Service- Find msg_add has $name, replace it.")
119         add = strings.Replace(add, old: "$name", dao.Query(account, query: "name"), n: -1)
120     }
121     if strings.Contains(add, symbol_amount) {
122         output.Print( info: "-----Service- Find msg_add has $name, replace it.")
123         add = strings.Replace(add, old: "$amount", dao.Query(account, query: "amount"), n: 1)
124     }
125     if strings.Contains(add, symbol_balance) {
126         output.Print( info: "-----Service- Find msg_add has $name, replace it.")
127         add = strings.Replace(add, old: "$balance", dao.Query(account, query: "balance"), n: 1)
128     }
129
130     if msg == "" {
131         msg += add
132     } else {
133         msg += symbol_divide + add
134     }
135     return msg
136 }
```

这里的函数是将新的字符串插到 `msg` 后面。分隔符 `symbol_divide` 用的是 `&`。

这里的每一条新加入的 `msg` 会在数据库中查找相应的记法词汇替换掉数据。

2.4.4 dao

db.go

```
9 // table struction
10 type USER struct {
11     account string `db:"account"`
12     password string `db:"password"`
13     name string `db:"name"`
14     amount float32 `db:"change"`
15     balance float32 `db:"change"`
16 }
```

首先规定一下数据库的 `schema`。

这里很显然就是一个说明性的结构体。

这个结构体可以再查询数据库时，将每一条记录的数据录入到这个结构体中，然后进行后续的判断及返回操作。

```

18 // database init
19 func Init_db() {
20     var err error
21
22     // link
23     db, err = sql.Open( driverName: "mysql", dataSourceName: "user_practice_of_programming:password@tcp(127.0.0.1:3306)/project_practice_of_program
24     if err != nil {
25         fmt.Println(err)
26         fmt.Println( a...: "-----Dao- Create sql.DB fail.")
27     }
28
29     // detect link state
30     err = db.Ping()
31     if err != nil {
32         fmt.Println( a...: "-----Dao- Database link fail.")
33         log.Fatal(err)
34     } else {
35         fmt.Println( a...: "-----Dao- Database link successfully.")
36     }
37 }
38
39 func Close_db() {
40     db.Close()
41 }

```

数据库的创建和关闭，简而言之就是一些固定操作。

● script.go

```

9     const script_path = "./scripts/"
10
11     // script struct
12     type Step struct {
13         S_speak string      `json:"speak"`
14         S_listen int           `json:"listen"`
15         S_branch map[string]string `json:"branch"`
16         S_silence string         `json:"silence"`
17         S_default string         `json:"default"`
18     }
19     type Script struct {
20         Steps map[string]Step `json:"Steps"`
21     }

```

const 和结构体。我们会直接读取 json，然后 encode 到相应的结构体中。非常方便。所以我们的语法树就直接是 json 存储的结构体。无需后续转换。

```

23 // functions for service
24 func Init_script(filename string) {
25     // read script json
26     bytes, err := ioutil.ReadFile(script_path + filename)
27     if err != nil {
28         fmt.Println(err)
29     }
30     var script Script
31     err = json.Unmarshal(bytes, &script)
32     if err != nil {
33         fmt.Println( a...: "-----Dao- Script parse fail!", err)
34         return
35     }
36
37     // store into steps( map: easy to use )
38     steps = script.Steps
39 }
40
41 // (for test currently)
42 func Return_script_setps() map[string]Step {
43     return steps
44 }

```

在陆劲中根据名称加载相应的 script。并且提供 script 根据 state 获取 step 的接口。

● dao.go

提供 login 服务。直接在数据库中查询。假如不符合则返回空串，符合则返回名称。

```
9 // Database block-----
10 var db *sql.DB
11
12 // Login for user
13 func Login(account string, password string) string {
14     // query all users
15     rows, err := db.Query( query: "select * from users where account='" + account + "' and password='" + password + "'" )
16     output.Print("-----Dao- Login check: " + account + " " + password)
17     if err != nil {
18         fmt.Println(err)
19     }
20
21     // judge login state
22     if rows.Next() {
23         // store a instance into user
24         var user USER
25         err = rows.Scan(&user.account, &user.password, &user.name, &user.amount, &user.balance)
26         if err != nil {
27             fmt.Println(err)
28         }
29         return user.name
30     }
31     return ""
32 }
```

提供 query 的接口，通过 account 和 query 的字段来进行查询，返回结果为 string

```
34 // query for user
35 func Query(account string, query string) string {
36     // query
37     rows, err := db.Query( query: "select " + query + " from users where account='" + account + "'" )
38     output.Print( info: "-----Dao- Select " + query + " from users where account='" + account + "'" )
39     if err != nil {
40         fmt.Println(err)
41     }
42
43     ans := ""
44     // store ans
45     if rows.Next() {
46         // store a instance into user
47         var tmp string
48         err = rows.Scan( dest...: &(tmp))
49         if err != nil {
50             fmt.Println(err)
51         }
52         if ans == "" {
53             ans += tmp
54         } else {
55             ans += ("&" + tmp)
56         }
57     }
58
59     output.Print("-----Dao- Query " + query + ":" + ans)
60     return ans
61 }
```

此处规定了全局变量 steps，存储脚本的内容。同时提供了根据状态在脚本中获取单元 step 的接口。

```
69 // Script block-----
70 var steps map[string]Step
71
72 // get step depend on state
73 func Get_step(state string) Step {
74     output.Print("-----Dao- Get step for " + state)
75     return steps[state]
76 }
```

2.4.5 output

● Print.go

在正式发行版本中，将需要输出的字符串打印在终端和 Log 文件中。

```
8 func Print(info string) {
9     if os.Args[1] != "script_test" {
10         Log_insert(info + "\n")
11         fmt.Println(info)
12     }
13 }
```

● Log.go

根据当前时间新建 Log 文件。并且根据输入字符串将其输入到文件中。

```
12 var log_file *os.File
13
14 // init
15 func Log_init() {
16     // get log_file name
17     cur := strings.Split(time.Now().Format(layout: "2006-01-02 15:04:05"), sep: " ")
18     date := cur[0]
19     time := strings.Replace(cur[1], old: ":", new: "-", n: -1)
20     log_file_name := date + "_" + time + ".log"
21
22     var err error
23     log_file, err = os.OpenFile(log_path+log_file_name, os.O_CREATE|os.O_APPEND, perm: 6)
24     if err != nil {
25         fmt.Println(a...: "Open " + log_file_name + " fail!")
26         os.Exit(code: 1)
27     }
28 }
29
30 // insert an info
31 func Log_insert(info string) {
32     log_file.WriteString(info)
33 }
34
35 // close
36 func Log_close() {
37     log_file.Close()
38 }
```

2.4.6 test

在程序设计中，本地逻辑的测试是很重要的。本地测试往往时程序上线前的一道机器重要的步骤。

我们这里集中于脚本测试。我们围绕两个检测机制展开：

1. 脚本的逻辑检测（测脚本本身性质）
2. 脚本的应答检测（测 service 和 dao 层结合提供 response 的功能正确性）

1. 首先是一些 const。这些都是后续会用到的常量。

```
14 const script_path = "./scripts"
15 const prefixe_input = "./scripts/test/input/input_"
16 const prefixe_reference = "./scripts/test/reference/reference_"
17 const test_account = "2020211376"
18 const symbol_Start = "Start"
19 const symbol_Exit = "Exit"
20 const symbol_End = "End"
```

2. 其次是进行所有脚本的检测和调用。

先加载当前脚本到 service。在此基础上进行：逻辑和应答的正确性检验。输出检验信息。

```
22 func Test_scripts() {
23     // init
24     service.Init()
25
26     // init dao layer data(script and db)
27     output.Print( info: "Init dao layer.")
28     dao.Init_db()
29
30     // detect all scripts
31     files, err := ioutil.ReadDir(script_path)
32     if err != nil {
33         fmt.Println(err)
34     }
35     for _, item := range files {
36         // select all scripts
37         if !item.IsDir() {
38             // load this script into dao.script
39             fmt.Println( a...: "\nTest:", item.Name())
40             dao.Init_script(item.Name())
41
42             // test logic & response
43             if Test_script_logic() {
44                 fmt.Println( a...: "Logic correct!")
45             } else {
46                 fmt.Println( a...: "Logic error!")
47             }
48
49             if Test_script_response(item.Name()) {
50                 fmt.Println( a...: "Response correct!")
51             } else {
52                 fmt.Println( a...: "Response error!")
53             }
54         }
55     }
```

3. 逻辑检验函数

```
57 func Test_script_logic() bool {
58     // get current script steps
59     steps := dao.Return_script_setps()
60
61     // judge Start
62     if _, ok := steps[symbol_Start]; !ok {...}
63
64     // judge Exit
65     if _, ok := steps[symbol_Exit]; !ok {
66         fmt.Println( a...: "No Exit step")
67         return false
68     }
69
70     for _, value := range steps {
71         // judge listen
72         if value.S_listen < 0 {...}
73
74         // judge branch
75         for _, v := range value.S_branch {...}
76
77         // judge silence
78         if _, ok := steps[value.S_silence]; value.S_listen > 0 && !ok {...}
79
80         // judge default
81         if _, ok := steps[value.S_default]; value.S_listen == 0 && !ok && value.S_default != symbol_End {...}
82     }
83
84     return true
85 }
```

这个函数解释如注释所说，检验 6 个正确性：

- 是否存在 Start 的步骤，
- 是否存在 Exit 的步骤
- branch 步骤是否合法（下一步存在）
- listen 是否合法（listen 是否大于等于 0）
- silence 是否合法（下一步存在）
- default 是否合法（下一步存在）

4. 逻辑检验函数

```
98 func Test_script_response(script_name string) bool {
99     // read reference
100     data, err := ioutil.ReadFile( filename: prefix_reference + strings.Split(script_name, sep: ".")[0] + ".txt")
101     if err != nil {
102         fmt.Println( a...: "Open test reference file error!", err)
103     }
104     test := string(data)
105
106     // read in lines & generate response into ans
107     ans := ""
108     input_file, err := os.OpenFile( name: prefix_input+strings.Split(script_name, sep: ".")[0]+".txt", os.O_RDWR, perm: 0666)
109     if err != nil {
110         fmt.Println( a...: "Open input file error!", err)
111     }
112     defer input_file.Close()
113     br := bufio.NewReader(input_file)
114     for {
115         // read into line
116         line, err := br.ReadString( delim: '\n')
117         line = strings.Replace(line, old: "\n", new: "", n: -1)
118         if err != nil {
119             break
120         }
121         // send this line to service layer and generate string-ans
122         msg, _ := service.Service(test_account, line)
123         ans += msg + "\n"
124     }
125     service.Clear_User(test_account)
126
127     return ans == test
128 }
```

因为已经初始化了 service 和 script，所以我们用用户 2020211376，进行应答测试。根据适当的 input 生成 ans 字符串，然后比对 reference 文件中读出的字符串，比对返回 bool 值。

2.5 templates

templates

- login.html
- service.html

该文件夹应该不用过多赘述。就是存放前端 html 模板的地方。

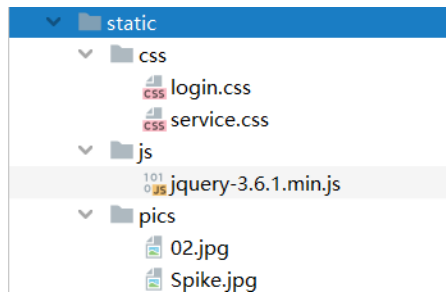
这个文件夹会在 router 创建时上传到服务器，并且资源路径由 router 负责。

分别为：

/login

/service

2.6 static



这个文件夹也很简单。在路由创建时会上传到服务器，资源路径是`/static`。

3. 综合分析

3.1 风格

满分 15 分，其中代码注释 6 分，命名 6 分，其它 3 分。

这里我们的代码注释都很清晰且不会太杂。所有的注释我都进行了检查和修改，在符合阅读习惯的基础上进行了风格和格式的统一。尤其在逻辑分支模块中，不同的逻辑分支的注释一定会保持高度的结构一致性，如下图：

```
61 // judge Start
62 if _, ok := steps[symbol_Start]; !ok {...}
66 // judge Exit
67 if _, ok := steps[symbol_Exit]; !ok {
68     fmt.Println(a...: "No Exit step")
69     return false
70 }
71 for _, value := range steps {
72     // judge listen
73     if value.S_listen < 0 {...}
77     // judge branch
78     for _, v := range value.S_branch {...}
84     // judge silence
85     if _, ok := steps[value.S_silence]; value.S_listen > 0 && !ok {...}
89     // judge default
90     if _, ok := steps[value.S_default]; value.S_listen == 0 && !ok && value.S_default != symbol_End {...}
94 }
```

命名的话，也是如出一辙：

1. 命名一定包含特定的意思，除非是可能会出现的暂存变量，如 key, value, item

```
18 // login page
19 group_login := router.Group( relativePath: "/login")
20 group_login.GET( relativePath: "/", func(c *gin.Context) {
21     c.HTML(http.StatusOK, name: "login.html", gin.H{
22         "title": "login.html",
23     })
24 })
25 group_login.POST( relativePath: "/", controller.Call_login)
26
27 // service page
28 group_service := router.Group( relativePath: "/service")
29 group_service.GET( relativePath: "/", func(c *gin.Context) {
30     c.HTML(http.StatusOK, name: "service.html", gin.H{
31         "title": "service.html",
32     })
33 })
34 group_service.POST( relativePath: "/", controller.Call_service)
```

2. 下划线命名法。而且同簇的变量会在前缀上保持一致性，如下图：

```
14 const script_path = "./scripts"
15 const prefix_input = "./scripts/test/input/input_"
16 const prefix_reference = "./scripts/test/reference/reference_"
17 const test_account = "2020211376"
18 const symbol_Start = "Start"
19 const symbol_Exit = "Exit"
20 const symbol_End = "End"
```

3.2 设计和实现

满分 30 分，其中数据结构 7 分，模块划分 7 分，功能 8 分，文档 8 分。

- 数据结构

我所有的数据结构都秉持简单，易用，不重复存储，不占用时间的原则。

比如状态存储：我们的 state 用 `map[string]string` 变量存储，所以使得十分明确，`account` 作为主键，能够对应唯一的一个用户。所有的数据流转也通过 `account` 这个数值把所有的东西联系起来。

此外除了结构体并没有太多用高深的数据结构。当然 go 语言特性可返回多值帮了大忙。

```
// variable for store users' states, and change
var states map[string]string // when get users' post, generate response(msg) depend on this state
```

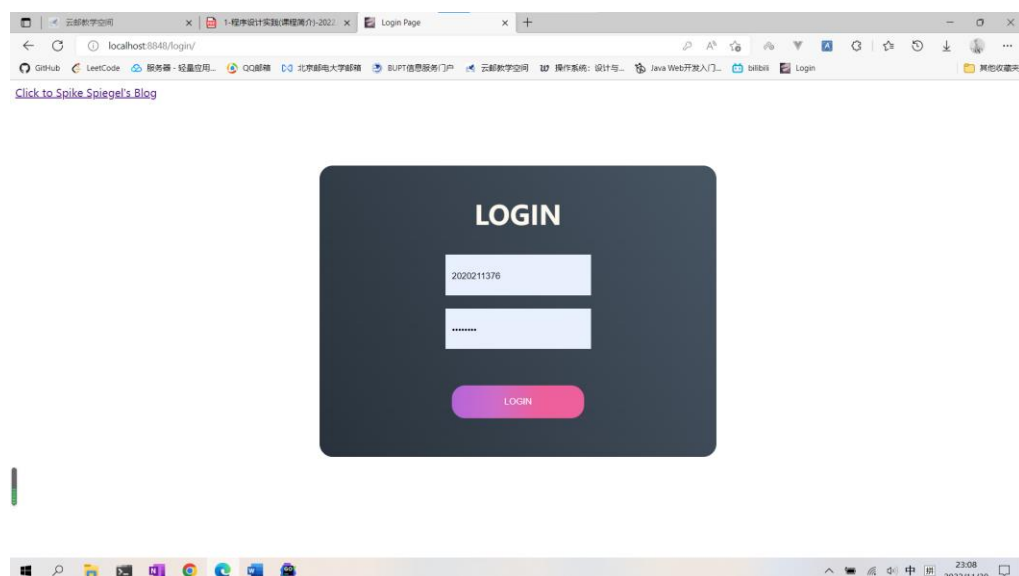
- 模块划分

我的模块划分参考了标准的 goBS 项目结构，分层设计，十分清晰，解耦程度高，除了 step 数据类型需要由 dao 层提供给 service 层进行数据整合，其余所有不会跨 package 使用。详细的模块划分见 2.2。

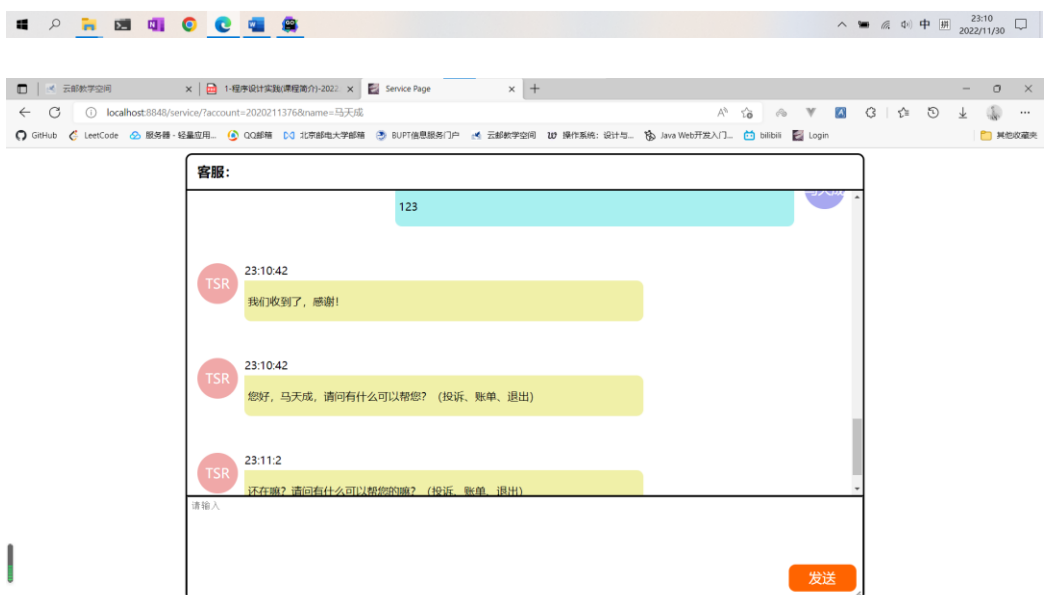
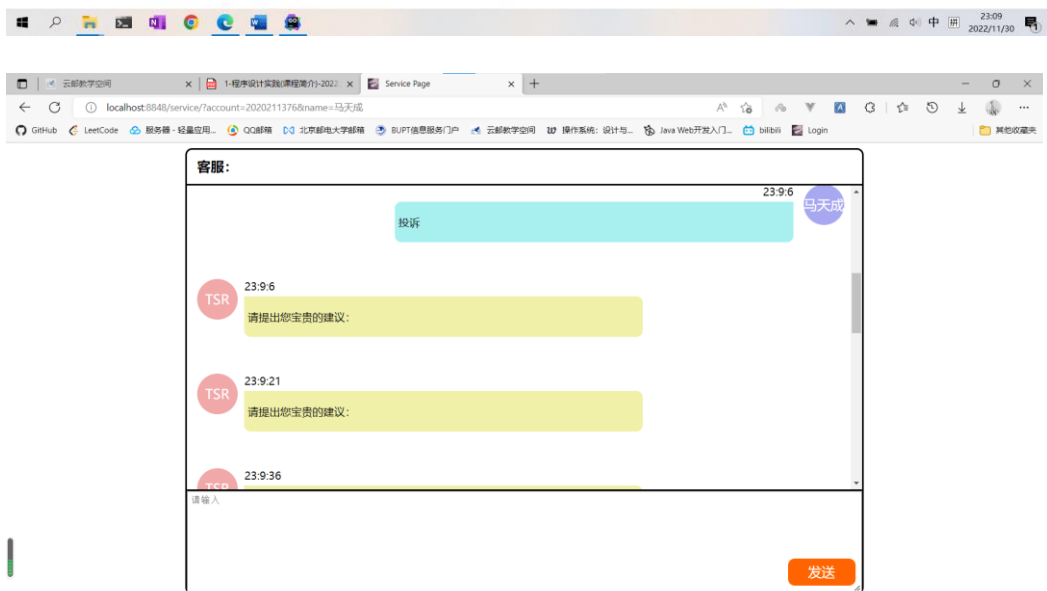
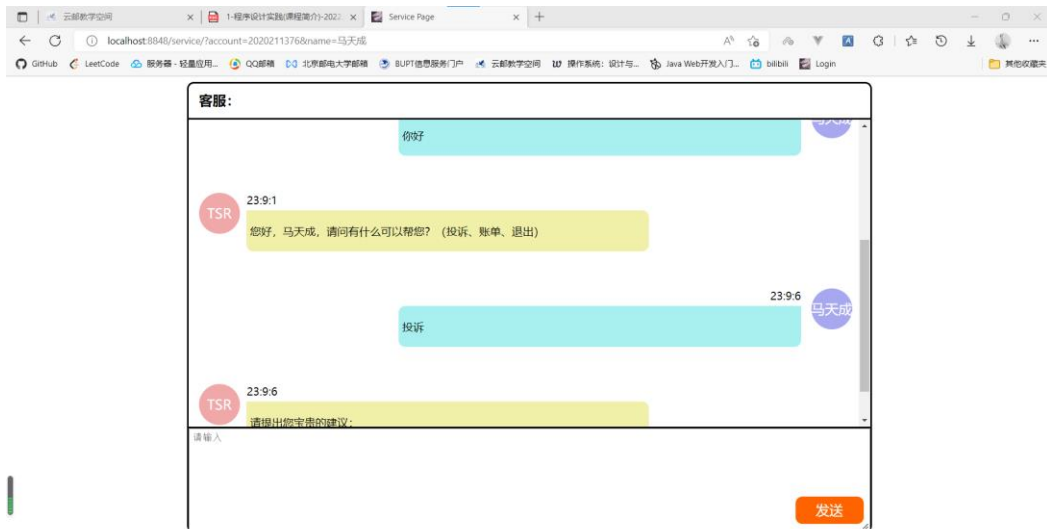
- 功能

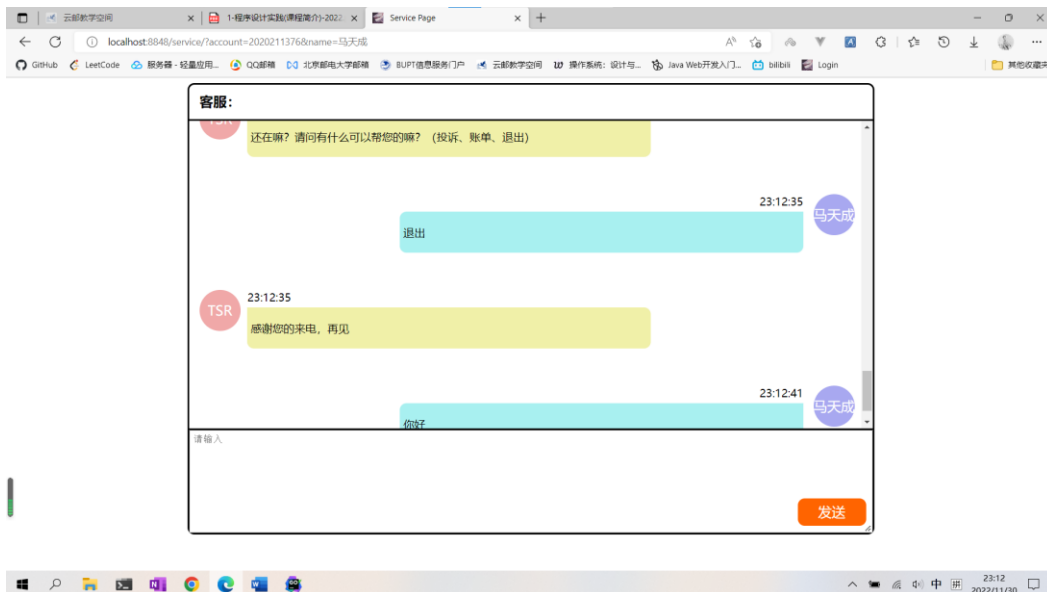
1. 发行版功能展示

登陆界面



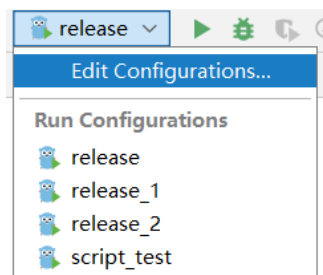
服务界面





- 功能上：完备。基本符合应答逻辑，也会延时触发事件。头像会显示用户名称，语句框上部分也会显示发送时间。
- 感官上：色调搭配合适，使用简答，输入框可以输入多行，然后一起发送。

后端运行配置



针对不同的功能，也会有不同的运行配置。这里主要体现在输入参数上。

- release 代表发布版本，不同配置运行不同脚本。
- script_test 代表本地脚本测试。

● 文档

文档很齐全。

README.md 中有项目概括和环境，使用方法介绍。

docs 中也有记法文档和实验报告。

3.3 接口

满分 15 分，其中程序间接口 8 分，人机接口 7 分。

● 程序间接口

程序间接口与我的层级分析紧密相连。其实我在最早放出的图中也写过很详细的数据传输策略。这里再放一遍：

3.4 测试

满分 30 分，测试桩 15 分，自动测试脚本 15 分。
测试模块已经在模块分析中详细介绍了。这里就不过多赘述。这里主要看一下本地测试的运行效果（前提，按照多脚本的要求）



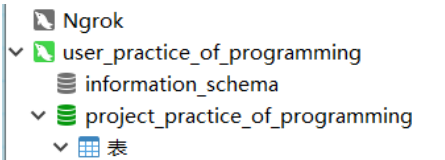
```
Run: script_test
GOROOT=D:\_Environment\Go #gosetup
GOPATH=D:\_Codes\GoLand\GOPATH #gosetup
D:\_Environment\Go\bin\go.exe build -o D:\_Codes\GoLand\demo\bin\script_test.exe . #gosetup
D:\_Codes\GoLand\demo\bin\script_test.exe script_test
####SCRIPT_TEST MODE
-----Dao- Database Link successfully.
Test: script.json
Logic correct!
Response correct!
Test: script_1.json
Logic correct!
Response correct!
Test: script_2.json
Logic correct!
Response correct!
Process finished with the exit code 0
```

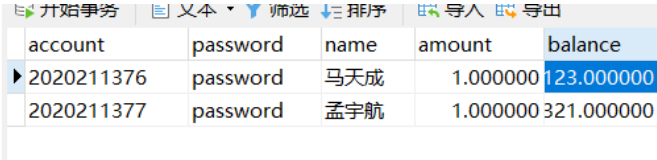
3.5 记法

满分 10 分，文档中对此脚本语言的语法的准确描述。
记法文档在 docs 文档中有详细介绍。

4. 设计方案

- 1. 采用分布式计算的思想，将计时器放到前端页面。后端只负责对应用户的状态逻辑修改和应答，降低后端服务器负荷，避免线程问题。
- 2. 程序维护一个脚本文件，但对于不同用户维护一张帐号状态对应表。
- 3. 符合 BS 架构程序设计规范，有良好的分层设计，规范化所有数据接口，不允许变量（只允许定义的结构类型）跨包使用。
- 4. 设计库数据设计





account	password	name	amount	balance
2020211376	password	马天成	1.000000	123.000000
2020211377	password	孟宇航	1.000000	321.000000

5. 改进方案

- 1. 每个脚本对应一个客服名字传到前端，然后个性化显示名称。
- 2. 客服应答可以随机化，随机选择同一性质的语句进行回复。