

作业 4——移动大厅 V3

1711430 江玥

2019 12 12

目录

1	作业要求及实现	1
1.1	基本要求	1
1.2	额外功能	1
2	准备知识	1
2.1	MyBatis	1
2.2	数据库连接池	2
2.2.1	UNPOOLED	2
2.2.2	POOLED	3
2.3	springMVC	3
2.3.1	MVC	3
2.3.2	SpringMVC	4
3	SSM 环境配置	5
3.1	环境要求	5
3.2	数据库环境	5
3.3	基本环境搭建	6
3.4	Mybatis 层编写	8
3.5	Spring 层	11
3.6	SpringMVC 层	11
3.7	Controller 和视图层编写	11
4	web 页面展示 & 特殊代码说明	12
4.1	登录注册	12
4.1.1	登录	12
4.1.2	注册	17
4.2	用户信息	19
4.2.1	查看用户信息	19
4.2.2	修改用户信息	19
4.3	使用 soso	20
4.4	充值	20
4.5	套餐变更	21
4.6	查看用户消费记录	21
4.7	办理退网	22

1 作业要求及实现

1.1 基本要求

此次作业的的基本要求：

1. 实现“嗖嗖移动”官网的部分功能
2. 用户模拟消费，包括通话、上网、发短信
3. **MySQL** 存取移动业务大厅数据
4. 系统界面 **Web** 形式
5. 采用 **SSM** 框架，实现与数据库之间的数据交互。

1.2 额外功能

在本次作业中，我自己还实现了以下几点：

1. 使用 **Maven** 完成项目的自动化配置
2. 解决 **json** 中文乱码问题
3. 使用 **mybatis** 自动化生成工具
4. 使用 **bootstrap** 对网页进行修饰
5. 使用 **ajax** 实现异步更新
6. 使用 **log4j** 记录项目运行日志
7. 使用 **lombok** 插件自动生成 **model** 的 **getter**、**setter** 以及构造方法

2 准备知识

2.1 MyBatis

MyBatis 是一个支持普通 **SQL** 查询，存储过程和高级映射的优秀持久层框架。**MyBatis** 消除了几乎所有的 **JDBC** 代码和参数的手工设置以及对结果集的检索封装。**MyBatis** 可以使用简单的 **XML** 或注解用于配置和原始映射，将接口和 **Java** 的 **POJO** (**Plain Old Java Objects**，普通的 **Java** 对象) 映射成数据库中的记录。

使用 **maven** 来构建项目，配置 **MyBatis**，需要在 **pom.xml** 文件里的 **dependencies** 里加入 **mybatis** 依赖：

```
1      <dependency>
2          <groupId>org.mybatis</groupId>
3          <artifactId>mybatis</artifactId>
4          <version>3.5.2</version>
5      </dependency>
6      <dependency>
7          <groupId>org.mybatis</groupId>
8          <artifactId>mybatis-spring</artifactId>
9          <version>2.0.2</version>
10     </dependency>
```

创建 **MyBatis** 配置文件: **mybatis-config.xml**, 配置数据库连接池、**dao** 模式、**mapper** 映射等等。

2.2 数据库连接池

JDBC 操作数据库的时候需要指定连接类型、加载驱动、建立连接、最终执行 **SQL** 语句, 这一过程十分冗杂。一个 **SQL** 的执行, 如果需要经过这么多步骤, 到下一个 **SQL** 语句的执行时, 还需要先经过这些步骤, 这样会造成很多不必要的性能损失, 并且对数据库本身也是一种压力。

为了减少这种不必要的消耗, 可以对数据的操作进行**拆分**。在 **Mybatis** 中, 数据库连接的建立和管理的部分叫做**数据库连接池**。

2.2.1 UNPOOLED

UNPOOLED: 不使用连接池的数据源, 当 **dataSource** 的 **type** 属性被配置成了 **UNPOOLED**, **MyBatis** 首先会实例化一个 **UnpooledDataSourceFactory** 工厂实例, 然后通过 **getDataSource()** 方法返回一个 **UnpooledDataSource** 实例对象引用。

UnpooledDataSource 会做以下事情:

1. 初始化驱动: 判断 **driver** 驱动是否已经加载到内存中, 如果还没有加载, 则会动态地加载 **driver** 类, 并实例化一个 **Driver** 对象, 使用 **DriverManager.registerDriver()** 方法将其注册到内存中, 以供后续使用。
2. 创建 **Connection** 对象: 使用 **DriverManager.getConnection()** 方法创建连接。
3. 配置 **Connection** 对象: 设置是否自动提交 **autoCommit** 和隔离级别 **isolationLevel**。
4. 返回 **Connection** 对象

每调用一次 **getConnection()** 方法, 都会通过 **DriverManager.getConnection()** 返回新的 **java.sql.Connection** 实例, 所以**没有连接池**。

2.2.2 POOLED

对于需要频繁地跟数据库交互的应用程序，可以在创建了 **Connection** 对象，并操作完数据库后，可以不释放掉资源，而是将它放到内存中，当下次需要操作数据库时，可以直接从内存中取出 **Connection** 对象，不需要再创建了，这样就极大地节省了创建 **Connection** 对象的资源消耗，而在内存中存放 **Connection** 对象的容器称之为**连接池**。数据库连接池便是运用了这种方式来大大节省访问数据库时间。

数据库连接池在 **mybatis** 中的配置：

```
1 <environments default="development">
2   <environment id="development">
3     <transactionManager type="JDBC"/>
4     <dataSource type="POOLED">
5       <property name="driver"
6         ↪ value="com.mysql.jdbc.Driver"/>
7       <property name="url"
8         ↪ value="jdbc:mysql://127.0.0.1:3306/soso?
9         useUnicode=true&characterEncoding=utf8"/>
10      <property name="username" value="root"/>
11      <property name="password" value="xxxxxx"/>
12    </dataSource>
  </environment>
</environments>
```

transactionManager（决定事务作用域和控制方式的事务管理器）的类型为 **JDBC**，说明使用了 **JDBC** 类型的数据库连接池了；**dataSource**（获取数据库连接实例的数据源）的类型为 **POOLED**，指明要使用数据库连接池对数据库进行连接。这里的 **property** 配置是数据库的具体信息，利用“键-值”对的形式，设置了数据库的驱动、数据库地址、用户名和密码。

2.3 springMVC

2.3.1 MVC

- **MVC** 是模型 (**Model**)、视图 (**View**)、控制器 (**Controller**) 的简写，是一种软件设计规范。
- 是将业务逻辑、数据、显示分离的方法来组织代码。
- **MVC** 主要作用是降低了视图与业务逻辑间的双向耦合。

Model（模型）：数据模型，提供要展示的数据，因此包含数据和行为，可以认为是领域模型或 **JavaBean** 组件（包含数据和行为），不过现在一般都分离开来：**Value Object**（数据 **Dao**）和服务层（行为 **Service**）。也就是模型提供了模

型数据查询和模型数据的状态更新等功能，包括数据和业务。

View (视图)：负责进行模型的展示，一般就是我们见到的用户界面，客户想看到的东西。

Controller (控制器)：接收用户请求，委托给模型进行处理（状态改变），处理完毕后把返回的模型数据返回给视图，由视图负责展示。也就是说控制器做了个调度员的工作。

2.3.2 SpringMVC

Spring MVC 是 **Spring Framework** 的一部分，是基于 **Java** 实现 **MVC** 的轻量级 **Web** 框架。**Spring** 的 **web** 框架围绕 **DispatcherServlet** [调度 **Servlet**] 设计。

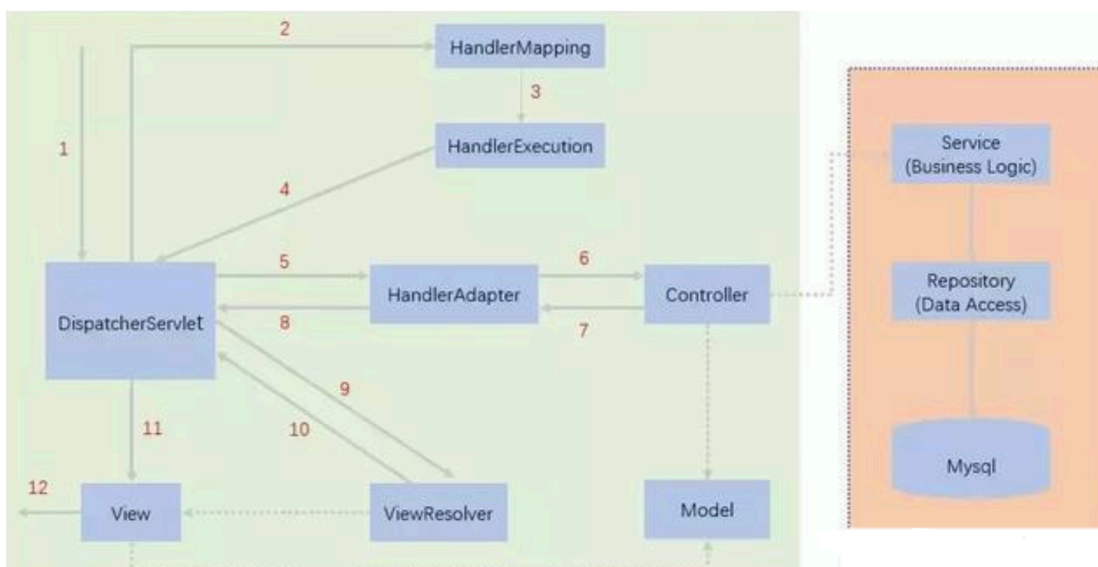


图 1：SpringMVC 执行原理

图为 **SpringMVC** 的一个较完整的流程图，实线表示 **SpringMVC** 框架提供的技术，不需要开发者实现，虚线表示需要开发者实现。

1. **DispatcherServlet** 表示前置控制器，是整个 **SpringMVC** 的控制中心。用户发出请求，**DispatcherServlet** 接收请求并拦截请求。
2. **HandlerMapping** 为处理器映射。**DispatcherServlet** 调用 **HandlerMapping**，**HandlerMapping** 根据请求 url 查找 **Handler**。
3. **HandlerExecution** 表示具体的 **Handler**，其主要作用是根据 url 查找控制器，如上 url 被查找控制器为：**hello**。
4. **HandlerExecution** 将解析后的信息传递给 **DispatcherServlet**，如解析控制器映射等。

5. **HandlerAdapter** 表示处理器适配器，其按照特定的规则去执行 **Handler**。
6. **Handler** 让具体的 **Controller** 执行。
7. **Controller** 将具体的执行信息返回给 **HandlerAdapter**，如 **ModelAndView**。
8. **HandlerAdapter** 将视图逻辑名或模型传递给 **DispatcherServlet**。
9. **DispatcherServlet** 调用视图解析器 (**ViewResolver**) 来解析 **HandlerAdapter** 传递的逻辑视图名。
10. 视图解析器将解析的逻辑视图名传给 **DispatcherServlet**。
11. **DispatcherServlet** 根据视图解析器解析的视图结果，调用具体的视图。
12. 最终视图呈现给用户。

3 SSM 环境配置

3.1 环境要求

- **IDEA**
- **MySQL 5.1.47**
- **Tomcat 9**
- **Maven 3.6**

3.2 数据库环境

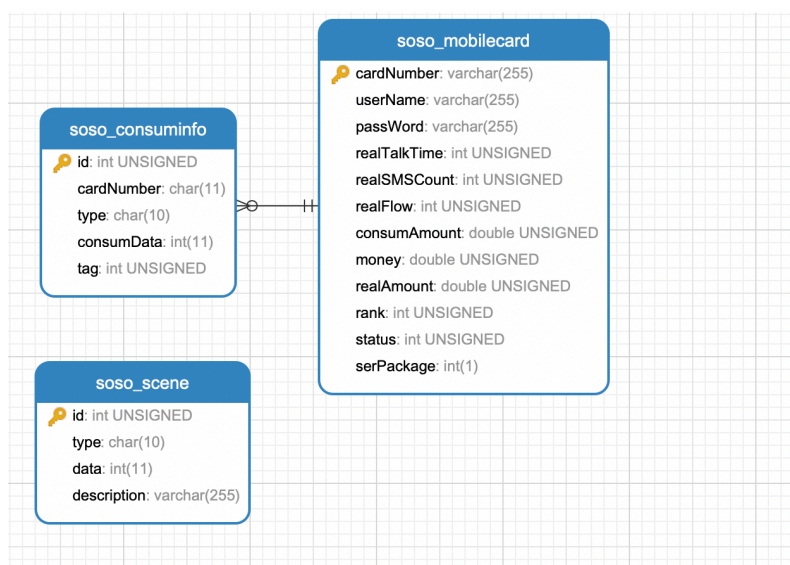


图 2: soso 数据库

三个数据库表单, `soso_consuminfo` 对应于 `ConsumInfo` 类; `soso_mobilecard` 对应于 `MobileCard` 类; `soso_scene` 对应于 `Scene` 类。

3.3 基本环境搭建

1. 新建 Maven 项目

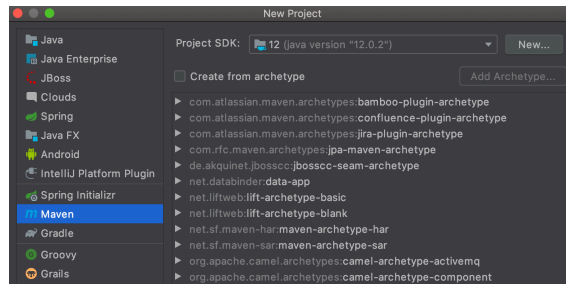


图 3：建立 Maven 项目

添加 `web` 框架的支持, 自动生成 `web` 文件夹, `web` 文件夹下的 `web.xml` 是 `web` 框架下的配置文件

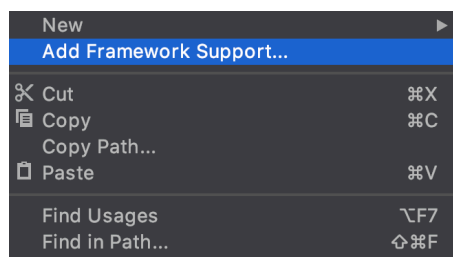


图 4：添加额外的框架依赖

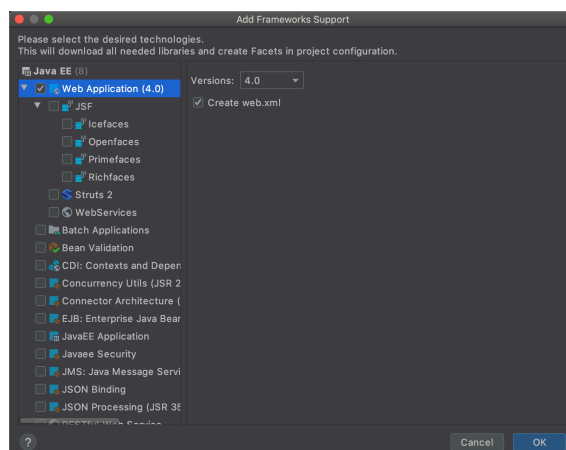


图 5：添加 web 框架依赖

通过这种方式添加的 `web` 框架, 是最新版本的 `web` 框架。

2. 导入相关的 pom 依赖

- **junit**(单元测试)
- 数据库驱动
- 数据库连接池 **c3p0**
- **Servlet - JSP**
- **Mybatis**
- **Spring**
- **lombok**(自动生成 **getter**、**setter** 方法和类构造函数)
- **fastjson**(解决 **json** 中文乱码问题)

3. Maven 资源过滤设置

```
1 <build>
2     <resources>
3         <resource>
4             <directory>src/main/java</directory>
5             <includes>
6                 <include>**/*.properties</include>
7                 <include>**/*.xml</include>
8             </includes>
9             <filtering>>false</filtering>
10        </resource>
11        <resource>
12            <directory>src/main/resources</directory>
13            <includes>
14                <include>**/*.properties</include>
15                <include>**/*.xml</include>
16            </includes>
17            <filtering>>false</filtering>
18        </resource>
19    </resources>
20 </build>
```

4. 建立基本结构和配置框架

- **com.kuang.pojo**
- **com.kuang.dao**
- **com.kuang.service**
- **com.kuang.controller**
- **mybatis-config.xml** **mybatis** 配置文件

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE configuration
3     PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-config.dtd">
5 <configuration>
6
7 </configuration>

```

- **applicationContext.xml springMVC 配置文件**

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3
4     ↪ xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5
6     ↪ xsi:schemaLocation="http://www.springframework.org/schema/beans
7     ↪ http://www.springframework.org/schema/beans/spring-beans.xsd">
8
9 </beans>

```

3.4 Mybatis 层编写

1. 数据库配置文件 **database.properties**

```

1 jdbc.driver=com.mysql.jdbc.Driver
2 # 如果使用 mysql8.0+, 需要增加时区的配置
3 jdbc.url=jdbc:mysql://localhost:3306/soso?
4     ↪ useSSL=true&useUnicode=true&characterEncoding=utf8
5 jdbc.username=root
6 jdbc.password=123456

```

2. IDEA 关联数据库

3. 编写 **MyBatis** 的核心配置文件 **mybatis-config.xml**

```

1 <configuration>
2     <!--配置数据源，交给 Spring 去做-->
3     <typeAliases>
4         <package name="com.kuang.pojo"/>
5     </typeAliases>

```

```
6      <!--配置类型处理器-->
7      <typeHandlers>
8          <typeHandler
10             ↪ handler="com.kuang.utils.typehandler.EnumRankTypeHandler"
11             ↪ javaType="com.kuang.pojo.enums.Rank"/>
12          ...
13      <!--配置 mapper 映射-->
14      <mappers>
15          <mapper class="com.kuang.dao.MobileCardMapper"/>
16          ...
17      </mappers>
18  </configuration>
```

4. 编写数据库对应的实体类 `com.kuang.pojo`，使用 `lombok` 插件。

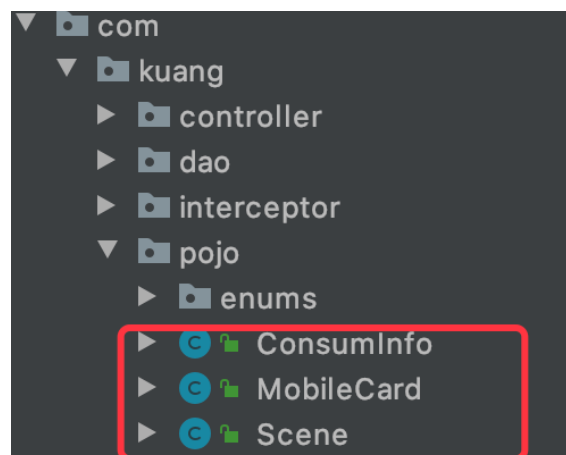


图 6：数据库对应实体类

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class ConsumInfo {
```

图 7：lombok 应用

使用了 `lombok` 插件后，无需在实体类中显式地声明 `getter`、`setter` 方法以及默认构造函数（全部参数以及无参构造函数）。

5. 编写 Dao 层的 `Mapper` 接口

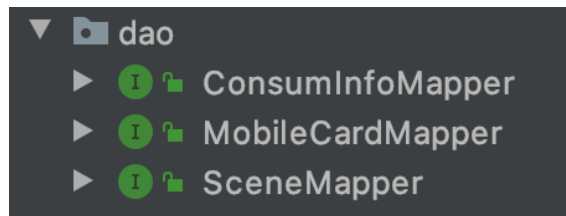


图 8: Mapper 接口

6. 编写 Service 层的接口和实现类

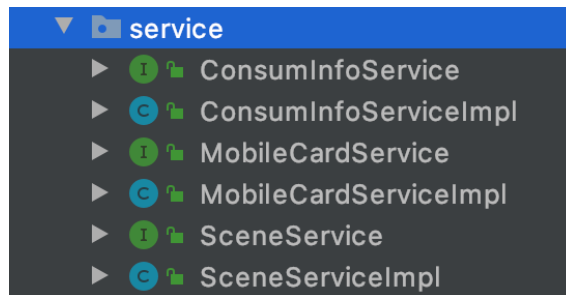


图 9: Service 接口和实现类

- 在 Service 接口中，声明与 Mapper 接口中相对应的方法如：ConsumInfoMapper 中的 delete 方法：

```
public interface ConsumInfoMapper {  
    @Update({  
        "update soso_consuminfo",  
        "set tag = 0",  
        "where cardNumber = #{cardNumber,jdbcType=CHAR}"  
    })  
    int delete(String cardNumber);  
}
```

图 10: 操作数据库表单的 delete 方法

对应于 Service 层中 ConsumInfoService 中的方法：

```
public interface ConsumInfoService {  
  
    // 删除一个卡号的所有消费记录  
    int delete(String cardNumber);  
}
```

图 11: service 接口对应方法

- 在接口的实现类中，将接口类的方法具体实现。如：

```
public class ConsumInfoServiceImpl implements ConsumInfoService{  
    private ConsumInfoMapper consumInfoMapper;  
  
    public void setConsumInfoMapper(ConsumInfoMapper consumInfoMapper) { this.consumInfoMapper = consumInfoMapper; }  
  
    public int delete(String cardNumber) { return consumInfoMapper.delete(cardNumber); }
```

图 12: delete 方法的具体实现

3.5 Spring 层

1. 配置 Spring 整合 MyBatis，这里数据源使用 c3p0 连接池
2. 编写 Spring 整合 Mybatis 的相关的配置文件——spring-dao.xml
spring-dao.xml 完成以下配置：
 - (a) 关联数据库文件，导入数据库文件：通过 spring 来读
 - (b) 配置数据库连接池，使用 c3p0，自动的加载配置文件，并且设置到对象里面
 - (c) 配置 SqlSessionFactory 对象
 - (d) 配置扫描 Dao 接口包，动态实现 Dao 接口注入到 spring 容器中 (com.kuang.dao)
3. Spring 整合 service 层: spring-service.xml

3.6 SpringMVC 层

1. web.xml
 - (a) 配置 DispatcherServlet
 - (b) 设置 encodingFilter，进行乱码过滤
 - (c) 设置 Session 过期时间
2. spring-mvc.xml
 - (a) 开启 SpringMVC 注解驱动，解决 json 乱码
 - (b) 静态资源默认 servlet 配置
 - (c) 配置 jsp，显示 ViewResolver 视图解析器，设置前缀 (/WEB-INF/jsp/) 和后缀 (.jsp)
 - (d) 扫描 web 相关的 bean (com.kuang.controller)
3. Spring 配置整合文件，applicationContext.xml

3.7 Controller 和视图层编写

具体操作在后续页面展示时进行详细说明。

4 web 页面展示 & 特殊代码说明

4.1 登录注册

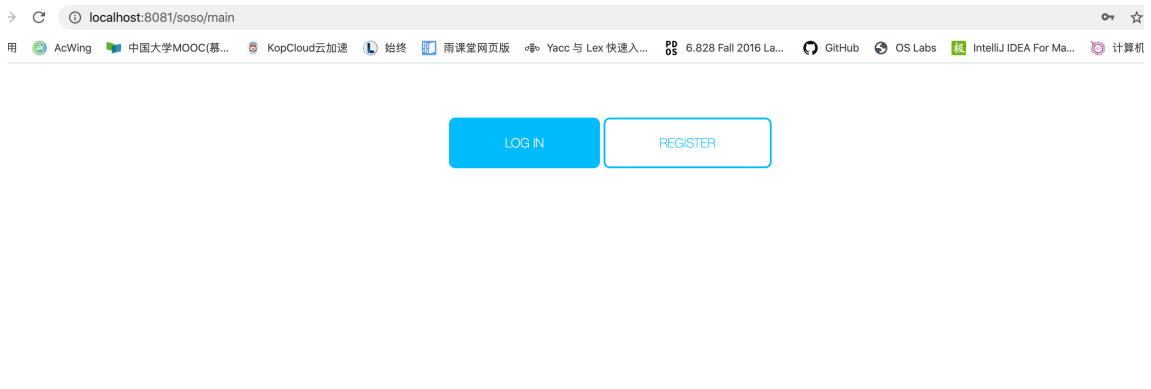


图 13： 登录注册页面

4.1.1 登录

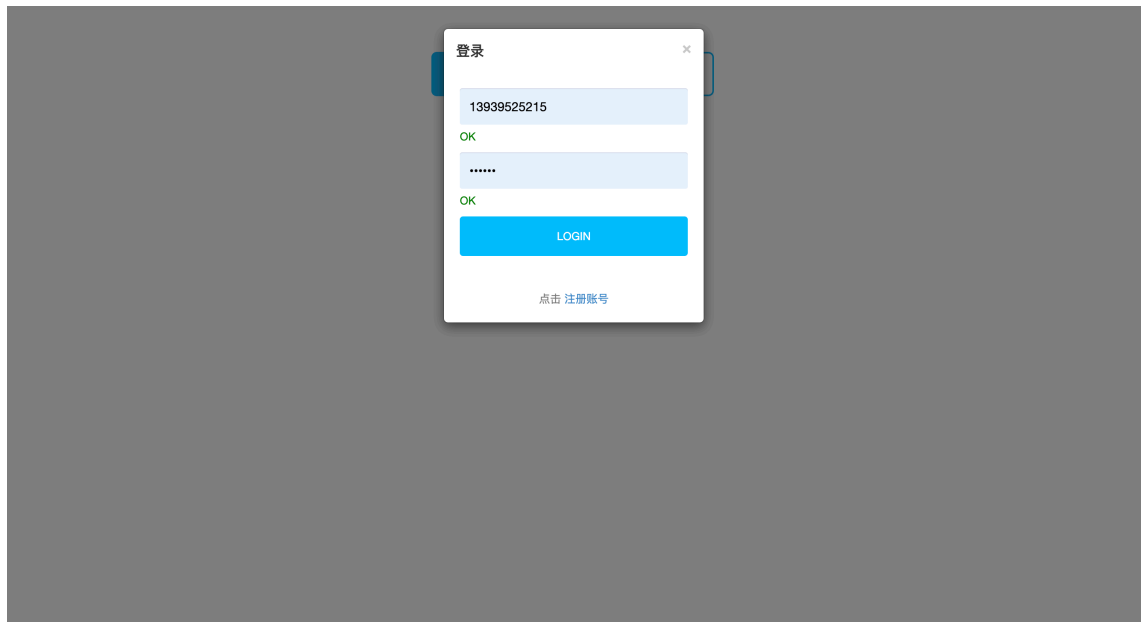
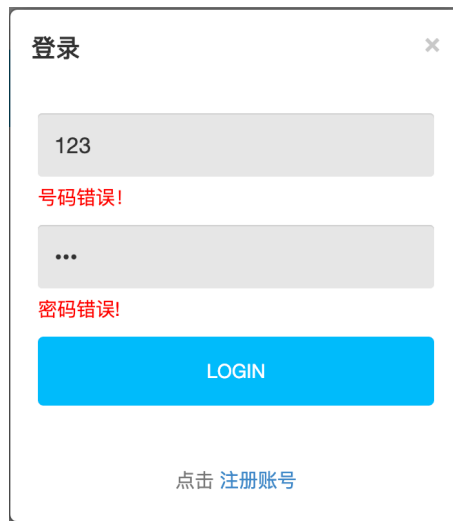


图 14： 登录页面

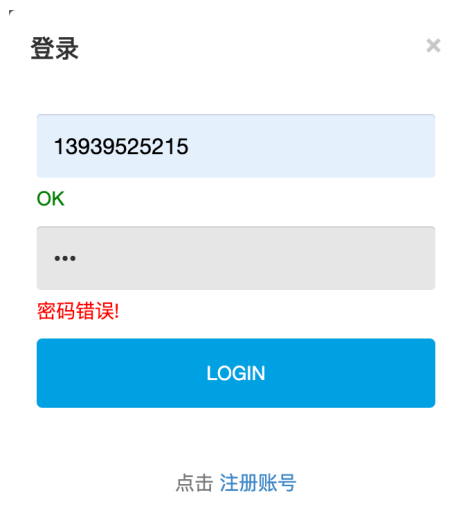
当输入的手机号为数据库中不存在的手机号的话，会报错，显示“号码错误！”



A login form titled "登录" (Login) with a close button (X) in the top right corner. It contains two input fields: the first contains "123" and has a red error message "号码错误!" (Number error!) below it; the second contains "..." and has a red error message "密码错误!" (Password error!) below it. At the bottom, there is a blue "LOGIN" button and a link "点击 注册账号" (Click to register account).

图 15： 号码错误提示

当输入了正确的手机号，却输入错误的密码时同样会报错，显示“密码错误！”



A login form titled "登录" (Login) with a close button (X) in the top right corner. It contains two input fields: the first contains "13939525215" and has a green "OK" message below it; the second contains "..." and has a red error message "密码错误!" (Password error!) below it. At the bottom, there is a blue "LOGIN" button and a link "点击 注册账号" (Click to register account).

图 16： 密码错误提示

不论是号码错误还是密码错误，此时点击 **LOGIN** 按钮，窗口会发生抖动，把密码清空，表示无法登陆！

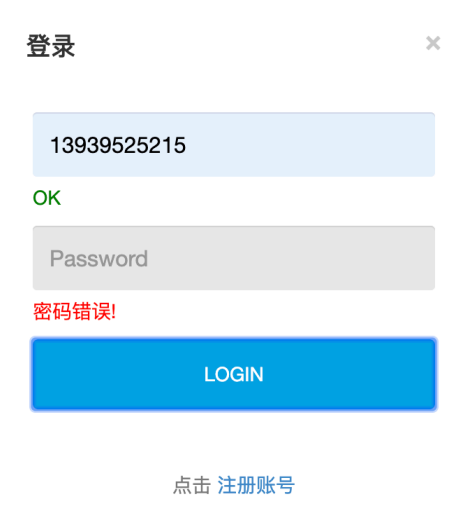


图 17： 登陆错误

若输入了正确的手机号和密码，则可以成功登陆进入 **soso** 主页面

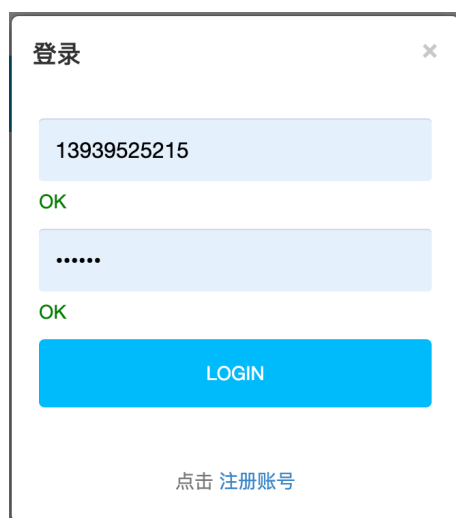


图 18： 登陆成功

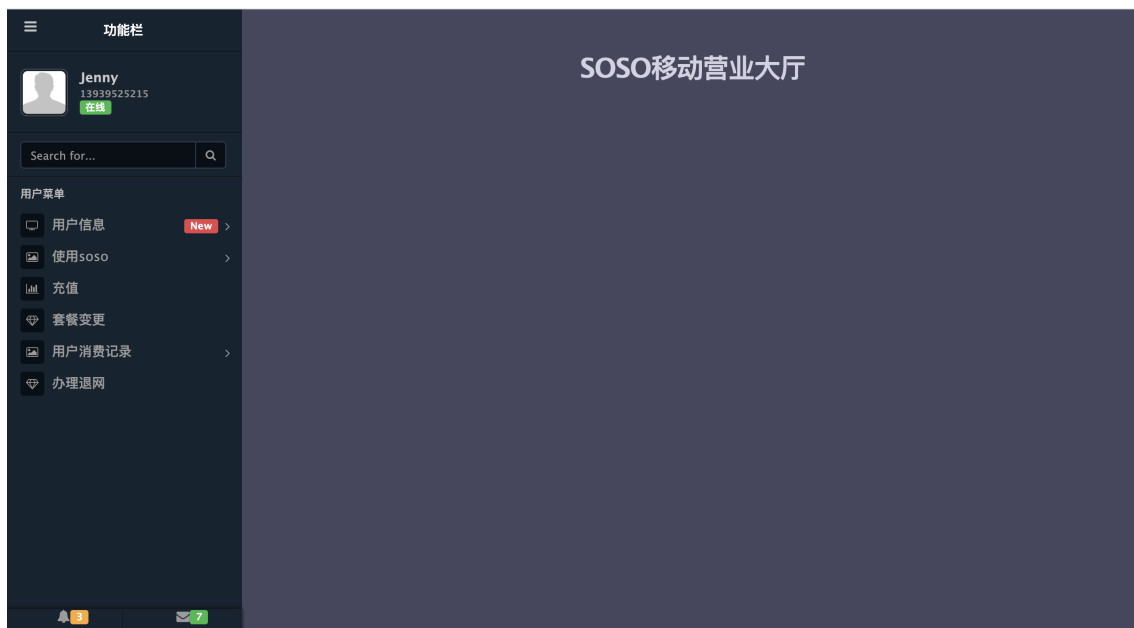


图 19： 进入主页面

异步 **ajax** 实现：

```
1 <p>
2   <input id="cardNumber" class="form-control" type="text"
      ↪ placeholder="cardNumber" name="cardNumber" onblur="a1()"
      ↪ required>
3   <span id="userInfo"></span>
4 </p>
```

onblur 标签表示若有变动，则触发 **a1** 函数，**a1** 函数：

```
1 function a1() {
2   $.post({
3     url: "${pageContext.request.contextPath}/login/a3",
4     data: {'cardNumber': $("#cardNumber").val()},
5     success: function (data) {
6       console.log(data.toString());
7       if (data.toString() === 'OK') {
8         $("#userInfo").css("color", "green");
9       } else {
10        $("#userInfo").css("color", "red");
11      }
12      $("#userInfo").html(data);
13    }
14  });
15 }
```

使用 **post** 方法去寻找对应于 **url**="login/a3"的 **controller** 方法 **LoginController** 中的 **a3** 方法:

```
1      @RequestMapping("/a3")
2      public String a3(String cardNumber, String password, HttpSession
   ↪ session) {
3          MobileCard card = null;
4          String msg = "";
5          if (cardNumber != null) {
6              card = mobileCardService.getMobileCard(cardNumber);
7          }
8
9          if (card == null) {
10             msg = " 号码错误! ";
11         } else {
12             msg = "OK";
13         }
14
15         if (password != null) {
16             if (card == null ||
   ↪ !card.getPassWord().equals(password)) {
17                 msg = " 密码错误!";
18             } else {
19                 msg = "OK";
20                 session.setAttribute("cardUser", card);
21             }
22         }
23         return msg;
24     }
```

由此可见，在 **controller** 方法中执行了对数据库的查询操作，若存在该号码，那么返回的 **msg** 为“OK”，对应于 **a1** 方法中 **userInfo** 被设置为绿色打印出的“OK”；否则，返回的 **msg** 为“号码错误”，对应于 **a1** 方法中 **userInfo** 被设置为红色打印出的“号码错误”。

异步 **ajax** 实现只举此例，其余例子不再详细说明。

4.1.2 注册

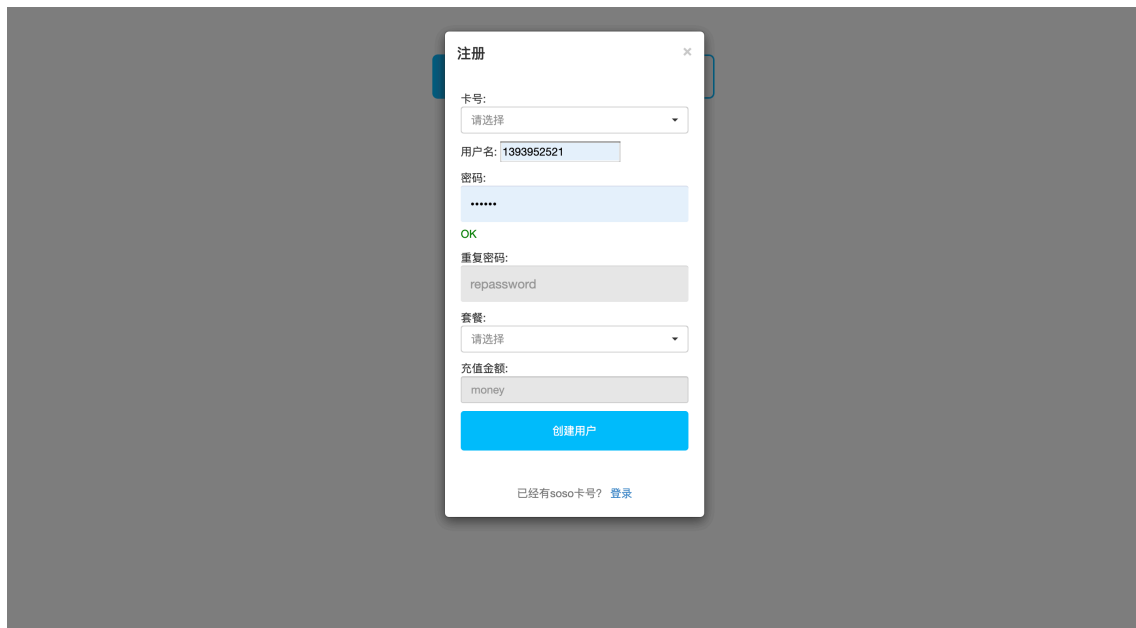


图 20：注册页面

1. 卡号：以“139”为开头，随机生成 9 个在数据库中不存在的号码，以下拉框的方式选择。

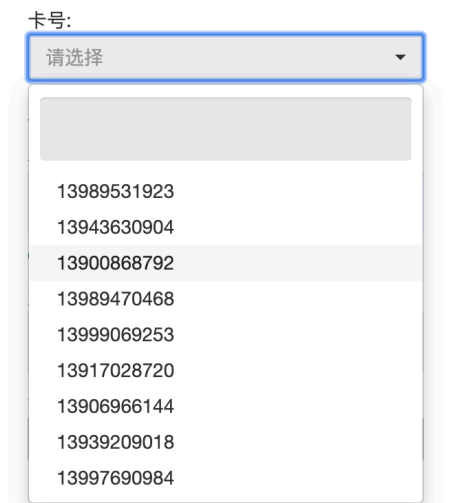
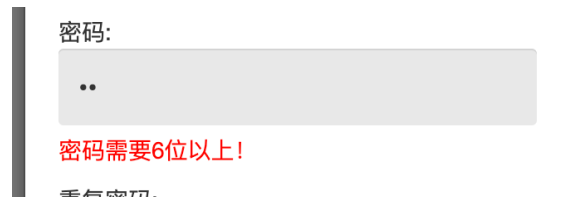


图 21：选择卡号

2. 用户名：设置用户名最大长度为 6，且不能为空
3. 密码：设置密码的最小长度为 6，最大长度为 20



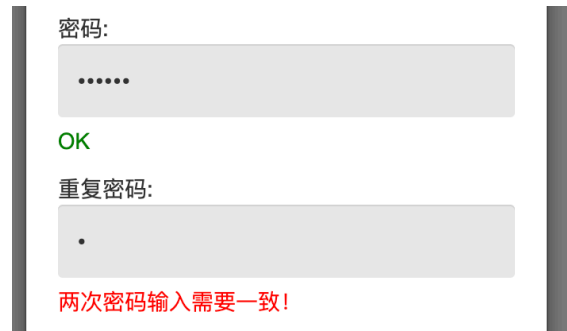
密码:

..

密码需要6位以上!

图 22：密码长度报错

4. 重复密码：重复密码需要和原密码保持一致，否则会报错提示



密码:

.....

OK

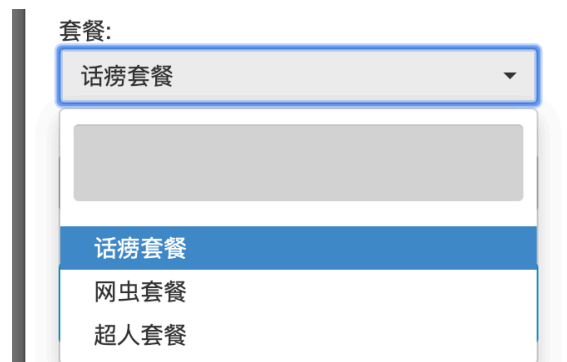
重复密码:

.

两次密码输入需要一致!

图 23：两次密码输入需要一致

5. 套餐：以下拉框形式挑选套餐



套餐:

话痨套餐

话痨套餐

网虫套餐

超人套餐

图 24：选择套餐

6. 充值金额：输入的充值金额必须大于套餐金额



充值金额:

10

充值金额小于套餐余额68.0元!

图 25：报错提示

4.2 用户信息

4.2.1 查看用户信息



图 26： 查看用户信息

4.2.2 修改用户信息



图 27： 修改用户信息

可以修改用户的名字和密码

4.3 使用 soso



图 28： 使用 soso

因为用户的状态为“`out_of_service`”，表示停机，所以无法使用 `soso` 的任何功能（通话、发短信、上网）

4.4 充值



图 29： 充值

最少的充值金额为 50 元，少于 50 元会报错提示

4.5 套餐变更



图 30：套餐变更

4.6 查看用户消费记录



图 31：本月账单查询



图 32：套餐余量查询

4.7 办理退网

直接退出，转到登录注册页面，再进行登录操作，已经无法进行登录

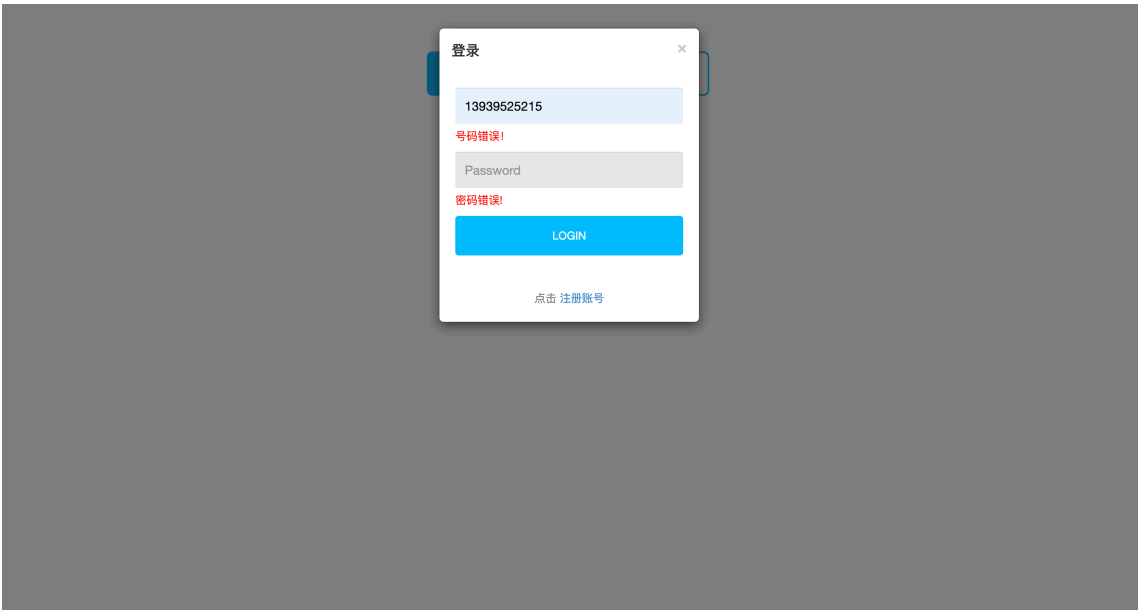


图 33：办理退网成功