目录

[1.实验内容 1](#_Toc151307041)

[1.1实验目的 1](#_Toc151307042)

[1.2实验要求 1](#_Toc151307043)

[1.3实验环境 1](#_Toc151307044)

[2.程序设计 3](#_Toc151307045)

[2.1程序层级设计 3](#_Toc151307046)

[2.2数据结构设计 4](#_Toc151307047)

[2.3代码设计分析 6](#_Toc151307048)

[3.综合分析 22](#_Toc151307049)

[3.1 风格：代码注释-命名-其他 22](#_Toc151307050)

[3.2 设计和实现：数据结构-模块划分-功能-文档 25](#_Toc151307051)

[3.3 接口：程序间接口-人机接口 28](#_Toc151307052)

[3.4 测试：测试桩-自动化脚本测试 29](#_Toc151307053)

[3.5 记法 29](#_Toc151307054)

# 1.实验内容

## 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。**

## 1.3实验环境

**操作系统：Windows 10**

**环境依赖：所有使用到的dependency都声明于pom.xml**

**开发语言：Java、HTML、CSS、JavaScript、Python**

**开发工具：IDEA Intellij, Git**

**开发框架：SpringBoot**

**前端模板：Thymeleaf**

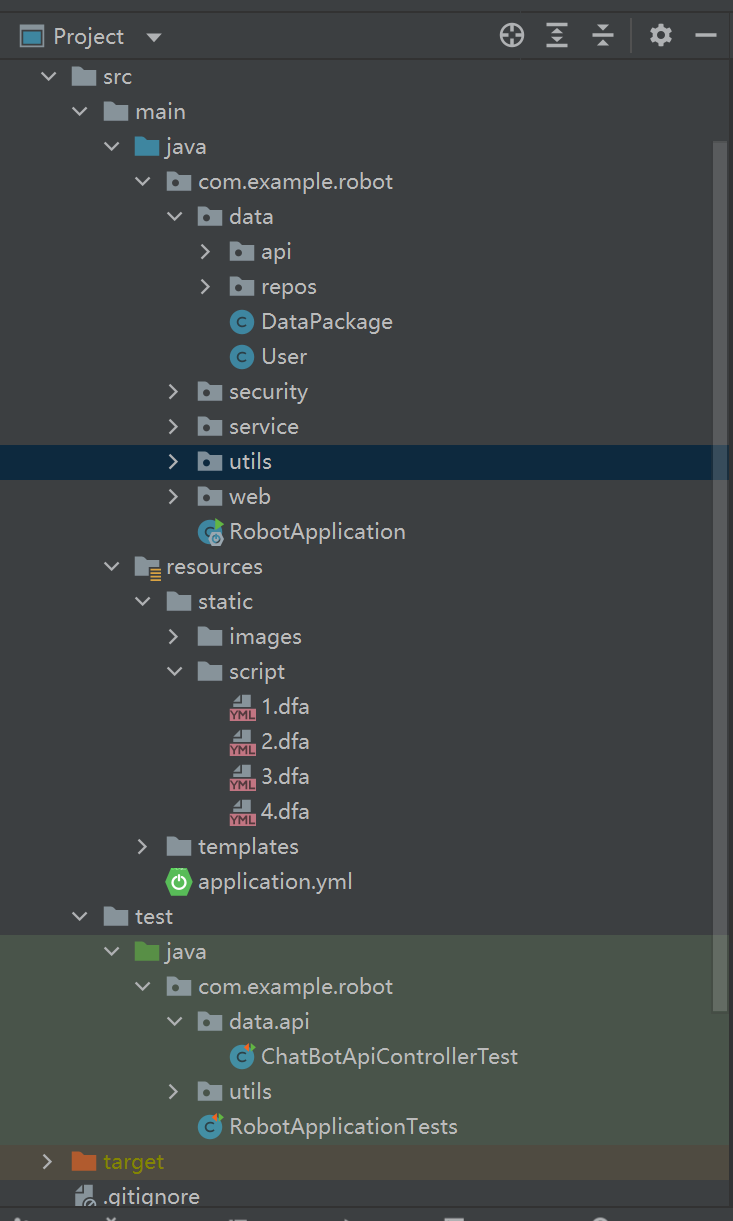
**测试环境：Chrome浏览器**

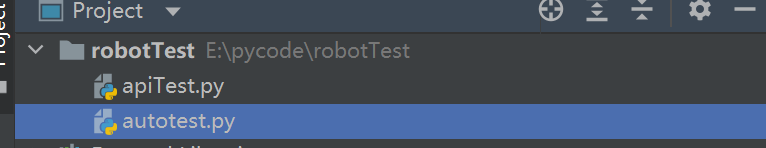
**测试工具：Mockito, JUnit5，Selenium，requests模块**

**使用方法：在IDE中直接运行，而后在浏览器访问localhost:8080**

# 2.程序设计

## 2.1程序层级设计





* **data: 用于放与数据有关的文件，并含有api、repos两个文件夹，其中api中含有提供REST API的类，repos含有User与DataPackage的仓库类，用于对这两种对象进行CRUD。**
* **security: 含有配置Spring Security的类，管理登录成功时的跳转以及页面的授权情况等；并含有用于处理退出登录时，将响应用户的机器人删除的handler。**
* **service: 含有各种提供服务的类，即实际用于业务处理的类。**
* **utils: 用于放工具类DFAParser，即解释器的具体实现部分。**
* **web: 用于放各种controller，控制对GET、POST的处理以及页面的跳转。**
* **resources: 用于放各种资源文件，编写的dfa脚本代码就放在script文件夹中，html文件均放置在templates文件夹中。**
* **test: 用于放置打桩测试的类（测试桩）**
* **robotTest: 含有用selenium进行自动化测试以及用requests对api进行测试的代码。**

## 2.2数据结构（实体）设计

package com.example.robot.data;  
  
import jakarta.persistence.\*;  
import lombok.\*;  
import org.springframework.security.core.GrantedAuthority;  
import org.springframework.security.core.authority.SimpleGrantedAuthority;  
import org.springframework.security.core.userdetails.UserDetails;  
  
import java.util.ArrayList;  
import java.util.Collection;  
import java.util.List;  
  
*/\*\*  
 \** ***@date*** *2023/11/13  
 \** ***@package*** *com.example.robot.data  
 \*/*@Entity  
@Data  
@NoArgsConstructor(access= AccessLevel.*PUBLIC*, force=true)  
@RequiredArgsConstructor  
public class User implements UserDetails {  
 @Id  
 @GeneratedValue(strategy= GenerationType.*AUTO*)  
 private Long id;  
   
 private final String username;  
 private final String password;  
 private double balance;  
   
 @ManyToMany(fetch = FetchType.*EAGER*)  
 @JoinTable(  
 name = "user\_package\_list", // 中间表的名称  
 joinColumns = @JoinColumn(name = "user\_id"), // 当前实体在中间表中的外键列名  
 inverseJoinColumns = @JoinColumn(name = "package\_id") // 关联实体在中间表中的外键列名  
 )  
 private final List<DataPackage> packageList = new ArrayList<>();  
   
 @Override  
 public Collection<? extends GrantedAuthority> getAuthorities () {  
 return List.*of*(new SimpleGrantedAuthority("ROLE\_USER"));  
 }  
   
 @Override  
 public boolean isAccountNonExpired () {  
 return true;  
 }  
   
 @Override  
 public boolean isAccountNonLocked () {  
 return true;  
 }  
   
 @Override  
 public boolean isCredentialsNonExpired () {  
 return true;  
 }  
   
 @Override  
 public boolean isEnabled () {  
 return true;  
 }  
}

package com.example.robot.data;  
  
import jakarta.persistence.\*;  
import lombok.\*;  
  
*/\*\*  
 \** ***@date*** *2023/11/18  
 \** ***@package*** *com.example.robot.data  
 \*/*@Entity  
@Data  
@RequiredArgsConstructor  
@NoArgsConstructor(access= AccessLevel.*PUBLIC*, force=true)  
public class DataPackage {  
   
 @Id  
 @GeneratedValue(strategy = GenerationType.*AUTO*)  
 private Long id;  
   
 @Column(unique = true)  
 final private String packageName;  
   
 final private int packagePrice;  
   
}

**现有实体类有User类（登录的用户信息）与DataPackage类（套餐信息），利用JPA来进行数据库的管理，其CRUD操作由repos文件夹底下的相应Repository来提供。后续继续拓展还可以加入其他类。**

## 2.3代码设计分析

1. **api：**

**ChatBotApiController.java:**

package com.example.robot.data.api;  
  
import com.example.robot.service.ChatBotService;  
import com.example.robot.utils.Message;  
import lombok.Data;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
import java.util.Map;  
  
*/\*\*  
 \** ***@date*** *2023/11/14  
 \** ***@package*** *com.example.robot.data.api  
 \*/  
  
/\*\*  
 \* 用于提供聊天机器人的REST服务接口  
 \*/*@RestController  
@RequestMapping(path="/api/chatbot")  
@CrossOrigin(origins="http://localhost:8080")  
@Data  
public class ChatBotApiController {  
   
 private ChatBotService service;  
   
 @Autowired  
 public ChatBotApiController(ChatBotService service) {  
 this.service = service;  
 }  
   
 */\*\*  
 \* 用于接收用户消息  
 \*  
 \** ***@param*** *message 用户消息,前端传过来的json用Map接收，然后取出message字段  
 \** ***@param*** *username 用户名，用于区分不同用户的聊天记录  
 \*/* @PostMapping(path="/user/{username}", produces="application/json")  
 public void sendUserMessage(@RequestBody Map<String, String> message,  
 @PathVariable String username) {  
 service.sendUserMessage(message.get("message"), username);  
 }  
   
 */\*\*  
 \* 用于获取机器人消息,并返回给调用者  
 \*  
 \** ***@param*** *username 用户名，用于区分不同用户的聊天记录  
 \** ***@return*** *机器人消息  
 \*/* @GetMapping("/robot/{username}")  
 public Message getBotMessage(@PathVariable String username) {  
 return service.getBotMessage(username);  
 }  
   
 */\*\*  
 \** ***@param*** *username 用户名，用于区分不同用户的聊天记录  
 \** ***@return*** *将用户和机器人的聊天记录返回给调用者  
 \*/* @GetMapping("/history/{username}")  
 public List<Message> getChatHistory(@PathVariable String username) {  
 return service.getChatHistory(username);  
 }  
   
}

**上面的代码用于提供REST API，供前端调用进行数据的传输。对于提供的API均将用法在注释中说明清楚了，本质是将ChatBotService进行包装，内部都是调用的ChatBotService的方法，从而将service与controller进行分离，让该controller专注于web相关的操作。**

1. **repos：**

**UserRepository.java, DataPackageRepository.java:**

package com.example.robot.data.repos;  
  
import com.example.robot.data.User;  
import jakarta.transaction.Transactional;  
import org.springframework.data.jpa.repository.Modifying;  
import org.springframework.data.jpa.repository.Query;  
import org.springframework.data.repository.CrudRepository;  
  
import java.util.List;  
  
public interface UserRepository extends CrudRepository<User, Long> {  
   
 User findByUsername(String username);  
   
 @Modifying(clearAutomatically = true)  
 @Transactional  
 @Query("update User u set u.balance = u.balance + ?2 where u.username = ?1")  
 void updateBalanceByUsername(String username, double balance);  
}

package com.example.robot.data.repos;  
  
import com.example.robot.data.DataPackage;  
import org.springframework.data.repository.CrudRepository;  
  
  
public interface DataPackageRepository  
 extends CrudRepository<DataPackage, Long> {  
   
 DataPackage findByPackageName(String packageName);  
}

**这两个Repository类都继承自CrudRepository，每个方法接口的具体实现会由Spring自动生成，同时添加了一些自定义的CRUD方法，用于更新与查找。**

1. **security：**
2. **SecurityConfig.java**

package com.example.robot.security;  
  
import com.example.robot.service.ChatBotService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.http.HttpMethod;  
import org.springframework.security.config.annotation.web.builders.HttpSecurity;  
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;  
import org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;  
import org.springframework.security.core.userdetails.UserDetailsService;  
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  
import org.springframework.security.crypto.password.PasswordEncoder;  
import org.springframework.security.web.SecurityFilterChain;  
import org.springframework.security.web.authentication.logout.LogoutSuccessHandler;  
  
*/\*\*  
 \** ***@date*** *2023/11/14  
 \** ***@package*** *com.example.robot.security  
 \*/*@Configuration  
@EnableWebSecurity  
public class SecurityConfig {  
   
 private final UserDetailsService userDetailsService;  
 private final LogoutSuccessHandler logoutSuccessHandler;  
   
 @Autowired  
 public SecurityConfig(UserDetailsService userDetailsService,  
 LogoutSuccessHandler logoutSuccessHandler) {  
 this.userDetailsService = userDetailsService;  
 this.logoutSuccessHandler = logoutSuccessHandler;  
 }  
   
 @Bean  
 public PasswordEncoder encoder() {  
 return new BCryptPasswordEncoder();  
 }  
   
 @Bean  
 public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {  
 http  
 .csrf(AbstractHttpConfigurer::disable)  
 .formLogin(  
 c -> c.loginPage("/login").defaultSuccessUrl("/chatbot", true)  
 .failureUrl("/login?error")  
 )  
 .logout(  
 c -> c.logoutSuccessUrl("/").logoutSuccessHandler(logoutSuccessHandler)  
 )  
 .userDetailsService(userDetailsService)  
 .authorizeHttpRequests(  
 c -> c.requestMatchers(HttpMethod.*GET*, "/", "/login", "/register", "/images/\*").permitAll()  
 .requestMatchers(HttpMethod.*POST*, "/login", "/register").permitAll()  
 .anyRequest().authenticated()  
 );  
   
 return http.build();  
 }  
}

**SecurityConfig用于对Spring Security进行配置，利用用户编写的UserDetailsService与LogoutSuccessHandler对用户的登录与退出进行处理；设置登录界面、登录成功与失败的界面；设置对不同url的授权限制。**

1. **LogoutSuccessHandlerImpl：**

package com.example.robot.security;  
  
import com.example.robot.service.ChatBotService;  
import jakarta.servlet.ServletException;  
import jakarta.servlet.http.HttpServletRequest;  
import jakarta.servlet.http.HttpServletResponse;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.security.core.Authentication;  
import org.springframework.security.web.authentication.logout.LogoutSuccessHandler;  
import org.springframework.stereotype.Component;  
  
import java.io.IOException;  
  
*/\*\*  
 \** ***@date*** *2023/11/21  
 \** ***@package*** *com.example.robot.security  
 \*/  
  
  
/\*\*  
 \* 用于处理用户退出登录的逻辑，将退出用户的聊天机器人（解释器DFAParser）删除  
 \*/*@Component  
@Slf4j  
public class LogoutSuccessHandlerImpl implements LogoutSuccessHandler {  
   
 private final ChatBotService chatBotService;  
   
 public LogoutSuccessHandlerImpl(ChatBotService chatBotService) {  
 this.chatBotService = chatBotService;  
 }  
   
 @Override  
 public void onLogoutSuccess(HttpServletRequest request, HttpServletResponse response, Authentication authentication) throws IOException, ServletException {  
 String user = authentication.getName();  
   
 *log*.info("用户{}退出登录。", user);  
 chatBotService.deleteUserChatBot(user);  
 response.sendRedirect("/");  
 }  
}

**对用户退出登录时的动作进行定义，将该用户对应的机器人删除，再次登录不会显示之前的聊天记录，方便对对象的收集，有利于空间利用。**

1. **service：**
2. **ChatBotService.java:**

package com.example.robot.service;  
  
  
import com.example.robot.utils.DFAParser;  
import com.example.robot.utils.Message;  
import org.springframework.core.env.Environment;  
import org.springframework.core.io.ResourceLoader;  
import org.springframework.stereotype.Service;  
  
import java.util.HashMap;  
import java.util.List;  
import java.util.Map;  
  
*/\*\*  
 \** ***@date*** *2023/11/18  
 \** ***@package*** *com.example.robot.service  
 \*/  
  
/\*\*  
 \* 聊天机器人服务类，将机器人操作封装成服务，供对应的RestController调用  
 \*  
 \** ***@apiNote*** *该类方法的注释均在ChatBotApiController中写过，这里就不重复写了，  
 \* 对应的方法名称相同。  
 \*/*@Service  
public class ChatBotService {  
   
 private final RepositoryService reposService;  
   
 private final Map<String, DFAParser> userParserMap = new HashMap<>();  
   
 private final Environment env;  
   
 private final ResourceLoader loader;  
   
   
 public ChatBotService(RepositoryService reposService, ResourceLoader loader,  
 Environment env) {  
 this.reposService = reposService;  
 this.loader = loader;  
 this.env = env;  
 }  
   
 */\*\*  
 \* 删除用户聊天机器人  
 \** ***@param*** *user 用户名  
 \*/* public void deleteUserChatBot(String user) {  
 userParserMap.remove(user);  
 }  
   
 */\*\*  
 \* 根据用户发送的消息进行相应的状态转换与响应  
 \** ***@param*** *message 用户消息  
 \** ***@param*** *username 当前用户  
 \*/* public void sendUserMessage(String message, String username) {  
 DFAParser parser = userParserMap.get(username);  
 List<Message> messages = parser.getMsgList();  
 parser.transferState(message);  
   
 // 处理用户消息  
 messages.add(new Message("user", message));  
   
 // 模拟机器人的回复  
 Message robotMessage = new Message("robot", parser.getCurrentResponse());  
 messages.add(robotMessage);  
 }  
   
 public Message getBotMessage(String username) {  
 DFAParser parser = userParserMap.get(username);  
   
 if (parser != null) {  
 List<Message> messages = parser.getMsgList();  
   
 // 返回聊天记录  
 return messages.get(messages.size() - 1);  
 } else {  
 return getChatHistory(username).get(0);  
 }  
   
 }  
   
 public List<Message> getChatHistory(String username) {  
 DFAParser parser;  
   
 if (!userParserMap.containsKey(username)) {  
 parser = new DFAParser(reposService, username,  
 loader, env);  
 userParserMap.put(username, parser);  
 } else {  
 parser = userParserMap.get(username);  
 }  
 List<Message> messages = parser.getMsgList();  
   
 // 返回聊天记录  
 if (messages.isEmpty()) {  
 messages.add(new Message("robot", parser.getCurrentResponse()));  
 }  
   
 return messages;  
 }  
}

**将每个用户与其对应的机器人（DFAParser对象）用HashMap进行存储，从而可以对多个用户同时进行处理。该service类对外提供删除指定用户机器人、根据用户消息进行处理、获取机器人消息、获取聊天记录的接口，供ChatBotApiController进行调用。**

1. **RepositoryService.java:**

package com.example.robot.service;  
  
import com.example.robot.data.repos.DataPackageRepository;  
import com.example.robot.data.repos.UserRepository;  
import lombok.Data;  
import org.springframework.stereotype.Service;  
  
*/\*\*  
 \** ***@date*** *2023/11/19  
 \** ***@package*** *com.example.robot.service  
 \*/  
  
/\*\*  
 \* 将所有的Repository注入到该类中，供其他Service调用，便于后续扩展  
 \*/*@Service  
@Data  
public class RepositoryService {  
   
 private UserRepository userRepos;  
 private DataPackageRepository dataPackageRepos;  
   
 public RepositoryService(UserRepository userRepos, DataPackageRepository dataPackageRepos) {  
 this.userRepos = userRepos;  
 this.dataPackageRepos = dataPackageRepos;  
 }  
}

**将实现的Repository集中在该service类中，便于后续开发中再加入新的Repository时的代码调整。**

1. **UserDetailRepositoryService.java:**

package com.example.robot.service;  
  
import com.example.robot.data.User;  
import com.example.robot.data.repos.UserRepository;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.security.core.userdetails.UserDetails;  
import org.springframework.security.core.userdetails.UserDetailsService;  
import org.springframework.security.core.userdetails.UsernameNotFoundException;  
import org.springframework.stereotype.Service;  
  
*/\*\*  
 \** ***@date*** *2023/11/13  
 \** ***@package*** *com.example.robot.service  
 \*/  
  
/\*\*  
 \* 该类实现了Spring Security的UserDetailsService接口，重写loadUserByUsername,  
 \* 用于从数据库中获取用户信息  
 \*/*@Service  
public class UserDetailRepositoryService  
 implements UserDetailsService {  
   
 final private UserRepository userRepos;  
   
 @Autowired  
 public UserDetailRepositoryService (UserRepository userRepos) {  
 this.userRepos = userRepos;  
 }  
   
 @Override  
 public UserDetails loadUserByUsername (String username)  
 throws UsernameNotFoundException {  
 User user = userRepos.findByUsername(username);  
   
 if (user != null) {  
 return user;  
 }  
   
 throw new UsernameNotFoundException(  
 "User '" + username + "' not found"  
 );  
 }  
}

**UserDetailsService的具体实现，根据UserRepository对象，进行用户的获取，供Spring Security用来进行用户的登录验证等操作。**

1. **DFAParser.java：**

package com.example.robot.utils;  
  
  
import com.example.robot.data.DataPackage;  
import com.example.robot.data.User;  
import com.example.robot.data.repos.UserRepository;  
import com.example.robot.service.RepositoryService;  
import lombok.Data;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.core.env.Environment;  
import org.springframework.core.io.Resource;  
import org.springframework.core.io.ResourceLoader;  
import org.yaml.snakeyaml.Yaml;  
  
import java.io.FileInputStream;  
import java.io.InputStream;  
import java.lang.reflect.Method;  
import java.text.DecimalFormat;  
import java.util.ArrayList;  
import java.util.HashMap;  
import java.util.List;  
import java.util.Map;  
  
*/\*\*  
 \** ***@date*** *2023/11/15  
 \** ***@package*** *com.example.robot.utils  
 \*/  
  
/\*\*  
 \* dfa脚本的解释器，即每个用户对应的客服机器人的底层实现  
 \*  
 \*/*@Data  
@Slf4j  
public class DFAParser {  
   
 */\*\*  
 \* 以HashMap形式存储dfa脚本中各个状态对应的信息  
 \*/* private HashMap<String, Object> stateMap;  
   
 */\*\*  
 \* 当前用户所处的状态，规定start为起始状态  
 \*/* private String state = "start";  
   
 */\*\*  
 \* 用户当前输入的内容，即机器人需要进行回应的消息  
 \*/* private String currentMsg = "";  
   
 */\*\*  
 \* 相应用户与机器人的聊天消息记录  
 \*/* private ArrayList<Message> msgList = new ArrayList<>();  
   
 private String username;  
 private RepositoryService reposService;  
   
 public DFAParser(RepositoryService reposService, String username,  
 ResourceLoader loader, Environment env) {  
 this.username = username;  
 this.reposService = reposService;  
   
 // 从配置文件中读取dfa脚本名称，并从resources文件夹中进行读取  
 String script = env.getProperty("dfa.filename");  
 String filePath = "classpath:/static/script/" + script;  
 Resource resource = loader.getResource(filePath);  
 *log*.info("script: {}", script);  
   
 try(  
 InputStream inputStream = new FileInputStream(resource.getFile())  
 ) {  
 Yaml yaml = new Yaml();  
 stateMap = yaml.load(inputStream);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
   
 */\*\*  
 \* 获取当前状态根据用户消息进行的回应  
 \** ***@return*** *响应消息  
 \*/* @SuppressWarnings("unchecked")  
 public String getCurrentResponse() {  
 StringBuilder builder = new StringBuilder();  
 Map<String, Object> current = (Map<String, Object>) stateMap.get(state);  
 ArrayList<Map<String, String>> list = (ArrayList<Map<String, String>>) current.get("response");  
   
 for (Map<String, String> map : list) {  
 // 对action进行处理，反射调用对应的操作方法  
 if (map.containsKey("action")) {  
 try {  
 Method method = this.getClass().getDeclaredMethod(map.get("action"));  
   
 if (method.getReturnType() != void.class) {  
 builder.append(method.invoke(this));  
 } else {  
 method.invoke(this);  
 }  
 } catch (Exception e) {  
 // 如果是ActionException，则直接返回异常信息  
 if (e.getCause() instanceof ActionException) {  
 builder.setLength(0);  
 builder.append(e.getCause().getMessage());  
 break;  
 }  
 e.printStackTrace();  
 }  
 } else if (map.containsKey("content")) {  
 builder.append(map.get("content"));  
 }  
 }  
   
 return builder.toString();  
 }  
   
 */\*\*  
 \* 根据用户消息进行状态转移，并进行currentMsg的更新  
 \** ***@param*** *msg 当前用户消息  
 \*/* @SuppressWarnings("unchecked")  
 public void transferState(String msg) {  
 currentMsg = msg;  
 Map<String, String> shift = (Map<String, String>) ((Map<String, Object>) stateMap.get(state)).get("shift");  
   
 if (shift.containsKey(msg)) {  
 state = shift.get(msg);  
 } else {  
 state = shift.get("default");  
   
 if (state == null) {  
 state = "start";  
 }  
 }  
 }  
   
 */\*\*  
 \* 判断用户是否已购买输入的套餐，已经购买则抛出异常，负责继续判断余额是否足够购买，不够则抛出相应异常  
 \** ***@throws*** *ActionException 用于包装异常信息并返回  
 \*/* public void hasPackageAndAfford() throws ActionException {  
 DataPackage p = reposService.getDataPackageRepos().findByPackageName(currentMsg);  
 if (p == null) {  
 throw new ActionException("您输入的套餐不存在");  
 }  
   
 User user = reposService.getUserRepos().findByUsername(username);  
 if (user.getPackageList().contains(p)) {  
 throw new ActionException("您已经购买了该套餐");  
 } else if (user.getBalance() < p.getPackagePrice()) {  
 throw new ActionException("您的余额不足");  
 }  
 }  
   
 */\*\*  
 \* 购买套餐，即将套餐添加到用户的套餐列表中，并扣除相应的余额  
 \*/* public void buyPackage() {  
 DataPackage p = reposService.getDataPackageRepos().findByPackageName(currentMsg);  
 User user = reposService.getUserRepos().findByUsername(username);  
   
 user.getPackageList().add(p);  
 user.setBalance(user.getBalance() - p.getPackagePrice());  
   
 reposService.getUserRepos().save(user);  
 }  
   
 */\*\*  
 \* 查找用户余额，并格式化为字符串  
 \** ***@return*** *格式化后的余额字符串  
 \*/* public String findBalance() {  
 // 创建DecimalFormat对象并设置格式  
 DecimalFormat format = new DecimalFormat("#,##0.00");  
   
 return format.format(reposService.getUserRepos().findByUsername(username).getBalance());  
 }  
   
 */\*\*  
 \* 判断用户输入的是否为数字  
 \** ***@throws*** *ActionException 用于包装异常信息并返回  
 \*/* public void isNumber() throws ActionException {  
 try {  
 Double.*parseDouble*(currentMsg);  
 } catch (NumberFormatException e) {  
 throw new ActionException("请输入数字");  
 }  
 }  
   
 */\*\*  
 \* 更新用户余额  
 \*/* public void updateBalance() {  
   
 reposService.getUserRepos().updateBalanceByUsername(username, Double.*parseDouble*(currentMsg));  
 }  
   
 */\*\*  
 \* 查找可以购买的套餐（还未被购买的）  
 \** ***@return*** *可购买套餐的字符串  
 \** ***@throws*** *ActionException 用于包装异常信息并返回  
 \*/* public String findAvailablePackage() throws ActionException {  
 List<DataPackage> boughtPackages = reposService.getUserRepos().findByUsername(username).getPackageList();  
 StringBuilder builder = new StringBuilder();  
   
 for (DataPackage p : reposService.getDataPackageRepos().findAll()) {  
 if (!boughtPackages.contains(p)) {  
 builder.append(p.getPackageName()).append("\n");  
 }  
 }  
   
 if (builder.length() == 0) {  
 throw new ActionException("无可购买套餐");  
 }  
   
 return builder.toString();  
 }  
   
 */\*\*  
 \* 判断用户是否已经购买了输入的套餐  
 \** ***@throws*** *ActionException 用于包装异常信息并返回  
 \*/* public void inBoughtPackages() throws ActionException {  
 List<DataPackage> boughtPackages = reposService.getUserRepos().findByUsername(username).getPackageList();  
   
 if (!boughtPackages.contains(reposService.getDataPackageRepos().findByPackageName(currentMsg))) {  
 throw new ActionException("您未购买该套餐");  
 }  
 }  
   
 */\*\*  
 \* 退订套餐，即将套餐从用户的套餐列表中移除，并退还相应的余额  
 \*/* public void unsubscribePackage() {  
 User user = reposService.getUserRepos().findByUsername(username);  
 DataPackage p = reposService.getDataPackageRepos().findByPackageName(currentMsg);  
   
 user.getPackageList().remove(p);  
 user.setBalance(user.getBalance() + p.getPackagePrice());  
   
 reposService.getUserRepos().save(user);  
 }  
   
 */\*\*  
 \* 查找已经购买的套餐  
 \** ***@return*** *已购买套餐的字符串  
 \** ***@throws*** *ActionException 用于包装异常信息并返回  
 \*/* public String findPackage() throws ActionException {  
 StringBuilder builder = new StringBuilder();  
 List<DataPackage> boughtPackages = reposService.getUserRepos()  
 .findByUsername(username)  
 .getPackageList();  
   
 if (boughtPackages.isEmpty()) {  
 throw new ActionException("您未购买任何套餐");  
 }  
   
 builder.append("您已购买的套餐有：\n");  
 for (DataPackage p : boughtPackages) {  
   
 builder.append(p.getPackageName()).append("\n");  
 }  
   
 return builder.toString();  
 }  
}  
  
class ActionException extends RuntimeException {  
 public ActionException(String message) {  
 super(message);  
 }  
}

**解释器的具体实现，对外实际提供的接口是getCurrentResponse与transferState，前者获取当前状态的响应内容，后者可根据当前的用户输入进行状态的转换。getCurrentResponse就是对dfa文件中当前状态的response部分的内容进行遍历：若为content，则将内容追加到结果后边；若为action，则根据用户输入的action名称反射调用相应的方法。通过反射调用主要是为了使代码更加简洁，免去了一堆if else 或 switch case的烦恼，只需要专注于编写某个动作对应的方法就可以。而transferState则是根据dfa文件中该状态对应的shift部分进行状态转换，获取当前输入对应的状态名。所有的动作方法均在该类中实现。**

1. **web：**
2. **WebConfig.java:**

package com.example.robot.web;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  
import org.springframework.security.crypto.password.PasswordEncoder;  
import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;  
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;  
  
*/\*\*  
 \** ***@date*** *2023/11/13  
 \** ***@package*** *com.example.robot.web  
 \*/*@Configuration  
public class WebConfig implements WebMvcConfigurer {  
   
 @Override  
 public void addViewControllers(ViewControllerRegistry registry) {  
 registry.addViewController("/login");  
 registry.addViewController("/").setViewName("login");  
 }  
}

**设置登录界面，并设置“/”对应的是login界面。避免了再单独给他们编写对应的Controller的麻烦。**

1. **RegisterForm.java:**

package com.example.robot.web;  
  
import com.example.robot.data.User;  
import lombok.Data;  
  
  
import org.springframework.security.crypto.password.PasswordEncoder;  
  
  
*/\*\*  
 \** ***@date*** *2023/11/14  
 \** ***@package*** *com.example.robot.web  
 \*/*@Data  
public class RegisterForm {  
   
 private String username;  
 private String password;  
 private String confirm;  
   
 public User toUser(PasswordEncoder encoder) {  
 return new User(username, encoder.encode(password));  
 }  
}

**注册用的数据类，用于前后端数据的传输与交互。**

1. **RegisterController.java:**

package com.example.robot.web;  
  
import com.example.robot.data.User;  
import com.example.robot.data.repos.UserRepository;  
import org.springframework.security.crypto.password.PasswordEncoder;  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PostMapping;  
import org.springframework.web.bind.annotation.RequestMapping;  
  
*/\*\*  
 \** ***@date*** *2023/11/14  
 \** ***@package*** *com.example.robot.web  
 \*/*@Controller  
@RequestMapping("/register")  
public class RegisterController {  
   
 private UserRepository userRepos;  
 private PasswordEncoder passwordEncoder;  
   
 public RegisterController(UserRepository userRepos, PasswordEncoder passwordEncoder) {  
 this.userRepos = userRepos;  
 this.passwordEncoder = passwordEncoder;  
 }  
   
 @GetMapping  
 public String showRegisterPage() {  
 return "register";  
 }  
   
 @PostMapping  
 public String processRegistration(RegisterForm form) {  
 if (form.getUsername().equals("") || form.getPassword().equals("")) {  
 return "redirect:/register?blank";  
 }  
   
 if (form.getPassword().equals(form.getConfirm())) {  
 User user = form.toUser(passwordEncoder);  
   
 if (userRepos.findByUsername(user.getUsername()) != null) {  
 return "redirect:/register?repeat";  
 }  
   
 userRepos.save(user);  
 return "redirect:/login";  
 }  
   
 return "redirect:/register?error";  
 }  
}

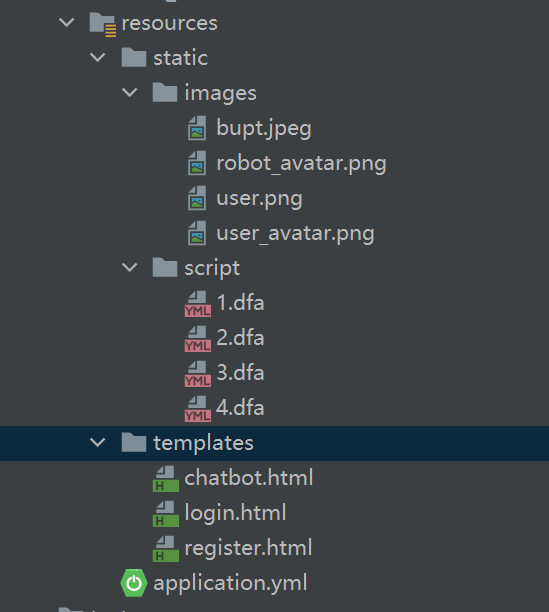
**用于控制用户注册，对注册成功、注册失败等进行不同的页面重定向的处理。**

1. **ChatBotController.java:**

package com.example.robot.web;  
  
import com.example.robot.data.User;  
import org.springframework.security.core.annotation.AuthenticationPrincipal;  
import org.springframework.stereotype.Controller;  
import org.springframework.ui.Model;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RequestMapping;  
  
*/\*\*  
 \** ***@date*** *2023/11/14  
 \** ***@package*** *com.example.robot.web  
 \*/*@Controller  
@RequestMapping("/chatbot")  
public class ChatBotController {  
  
 @GetMapping  
 public String showChatBotPage (Model model, @AuthenticationPrincipal User user) {  
 model.addAttribute("user", user.getUsername());  
 return "chatbot" ;  
 }  
   
}

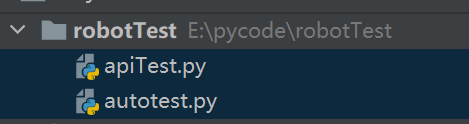
**将chatbot对应的页面进行返回，并在Model中添加user属性，便于前后端数据交互时，对不同用户的识别与处理。**

1. **resources：**



**images底下是用到的图片；script中是编写的脚本代码；templates是各种页面的html文件；application.yml是用来对程序进行配置的，如数据库配置、使用的脚本文件等。**

1. **test：**



**在程序设计中， 本地逻辑的测试以及自动化脚本测试很重要。 因此测试这部分分为了两部分，一部分是专门测试函数功能正确性的测试桩，另一部分是测试整个程序的自动化脚本测试。**

1. **测试桩**

package com.example.robot.data.api;  
  
import com.example.robot.service.ChatBotService;  
import com.example.robot.utils.Message;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import org.junit.jupiter.api.extension.ExtendWith;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;  
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;  
import org.springframework.boot.test.mock.mockito.MockBean;  
import org.springframework.http.HttpStatus;  
import org.springframework.http.MediaType;  
import org.springframework.mock.web.MockHttpServletResponse;  
import org.springframework.security.test.context.support.WithMockUser;  
import org.springframework.test.context.junit.jupiter.SpringExtension;  
import org.springframework.test.web.servlet.MockMvc;  
import org.springframework.test.web.servlet.setup.MockMvcBuilders;  
import org.springframework.web.context.WebApplicationContext;  
  
  
import java.util.Collections;  
import java.util.List;  
  
import static org.assertj.core.api.Assertions.*assertThat*;  
import static org.mockito.Mockito.*verify*;  
import static org.mockito.Mockito.*when*;  
import static org.springframework.security.test.web.reactive.server.SecurityMockServerConfigurers.*springSecurity*;  
import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.*user*;  
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*get*;  
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*post*;  
  
@ExtendWith(SpringExtension.class)  
@WebMvcTest(ChatBotApiController.class)  
@AutoConfigureMockMvc(addFilters = false)  
class ChatBotApiControllerTest {  
   
 @Autowired  
 private MockMvc mockMvc;  
   
 @MockBean  
 private ChatBotService mockService;  
   
   
 @Test  
 void testSendUserMessage() throws Exception {  
 // Setup  
 // Run the test  
 final MockHttpServletResponse response = mockMvc.perform(*post*("/api/chatbot/user/{username}", "user")  
 .param("message", "余额"))  
 .andReturn().getResponse();  
   
 // Verify the results  
 *assertThat*(response.getStatus()).isEqualTo(HttpStatus.*OK*.value());  
 *assertThat*(response.getContentAsString()).isEqualTo("expectedResponse");  
 *verify*(mockService).sendUserMessage("message", "user");  
 }  
   
 @Test  
 void testGetBotMessage() throws Exception {  
 // Setup  
 *when*(mockService.getBotMessage("username")).thenReturn(new Message("sender", "content"));  
   
 // Run the test  
 final MockHttpServletResponse response = mockMvc.perform(*get*("/api/chatbot/robot/{username}", "username")  
 .accept(MediaType.*APPLICATION\_JSON*))  
 .andReturn().getResponse();  
   
 // Verify the results  
 *assertThat*(response.getStatus()).isEqualTo(HttpStatus.*OK*.value());  
 *assertThat*(response.getContentAsString()).isEqualTo("expectedResponse");  
 }  
   
 @Test  
 void testGetChatHistory() throws Exception {  
 // Setup  
 // Configure ChatBotService.getChatHistory(...).  
 final List<Message> messages = List.*of*(new Message("sender", "content"));  
 *when*(mockService.getChatHistory("username")).thenReturn(messages);  
   
 // Run the test  
 final MockHttpServletResponse response = mockMvc.perform(*get*("/api/chatbot/history/{username}", "username")  
 .accept(MediaType.*APPLICATION\_JSON*))  
 .andReturn().getResponse();  
   
 // Verify the results  
 *assertThat*(response.getStatus()).isEqualTo(HttpStatus.*OK*.value());  
 *assertThat*(response.getContentAsString()).isEqualTo("expectedResponse");  
 }  
   
 @Test  
 void testGetChatHistory\_ChatBotServiceReturnsNoItems() throws Exception {  
 // Setup  
 *when*(mockService.getChatHistory("username")).thenReturn(Collections.*emptyList*());  
   
 // Run the test  
 final MockHttpServletResponse response = mockMvc.perform(*get*("/api/chatbot/history/{username}", "username")  
 .accept(MediaType.*APPLICATION\_JSON*))  
 .andReturn().getResponse();  
   
 // Verify the results  
 *assertThat*(response.getStatus()).isEqualTo(HttpStatus.*OK*.value());  
 *assertThat*(response.getContentAsString()).isEqualTo("[]");  
 }  
}

package com.example.robot.utils;  
  
import com.example.robot.data.DataPackage;  
import com.example.robot.data.User;  
import com.example.robot.data.repos.DataPackageRepository;  
import com.example.robot.data.repos.UserRepository;  
import com.example.robot.service.RepositoryService;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import org.junit.jupiter.api.extension.ExtendWith;  
import org.mockito.Mock;  
import org.mockito.junit.jupiter.MockitoExtension;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.core.env.Environment;  
import org.springframework.core.io.ResourceLoader;  
import org.springframework.mock.env.MockEnvironment;  
  
import static org.assertj.core.api.Assertions.*assertThat*;  
import static org.assertj.core.api.Assertions.*assertThatThrownBy*;  
import static org.mockito.Mockito.*verify*;  
import static org.mockito.Mockito.*when*;  
  
@SpringBootTest  
@ExtendWith(MockitoExtension.class)  
class DFAParserTest {  
   
 @Mock  
 private RepositoryService mockReposService;  
   
 @Mock  
 private DataPackageRepository mockDataPackageRepos;  
   
 @Mock  
 private UserRepository mockUserRepos;  
   
 @Autowired  
 private ResourceLoader loader;  
  
 @Autowired  
 private Environment env;  
   
 private DFAParser dfaParserUnderTest;  
   
 private User user;  
   
 @BeforeEach  
 void setUp() {  
 dfaParserUnderTest = new DFAParser(mockReposService, "username", loader, env);  
 }  
   
 @Test  
 void testGetCurrentResponse() {  
 // Setup  
 // Run the test  
 final String result = dfaParserUnderTest.getCurrentResponse();  
   
 // Verify the results  
 *assertThat*(result).isEqualTo("你好，请问有什么想要问的吗？（撸猫位置，撸狗位置）");  
 }  
   
 @Test  
 void testTransferState() {  
 // Setup  
 // Run the test  
 dfaParserUnderTest.transferState("撸猫位置");  
   
 // Verify the results  
 *assertThat*(dfaParserUnderTest.getCurrentResponse()).isEqualTo("撸猫位置在宿舍楼前");  
   
 dfaParserUnderTest.transferState("撸狗位置");  
   
 // Verify the results  
 *assertThat*(dfaParserUnderTest.getCurrentResponse()).isEqualTo("撸狗位置在宿舍楼后");  
   
 dfaParserUnderTest.transferState("fsijojfoisjdof");  
   
 // Verify the results  
 *assertThat*(dfaParserUnderTest.getCurrentResponse()).isEqualTo("你好，请问有什么想要问的吗？（撸猫位置，撸狗位置）");  
 }  
   
 @Test  
 void testHasPackageAndAfford() {  
 // Setup  
 user = new User("username", "pswd");  
 *when*(mockReposService.getDataPackageRepos()).thenReturn(mockDataPackageRepos);  
 *when*(mockReposService.getUserRepos()).thenReturn(mockUserRepos);  
 *when*(mockDataPackageRepos.findByPackageName("套餐1")).thenReturn(new DataPackage("套餐1", 100));  
 *when*(mockDataPackageRepos.findByPackageName("套餐x")).thenThrow(new ActionException("套餐不存在"));  
 *when*(mockUserRepos.findByUsername("username")).thenReturn(user);  
   
 user.setBalance(200);  
 dfaParserUnderTest.transferState("套餐1");  
 // Run the test  
 dfaParserUnderTest.hasPackageAndAfford();  
   
 dfaParserUnderTest.transferState("套餐x");  
 // Run the test  
 *assertThatThrownBy*(() -> dfaParserUnderTest.hasPackageAndAfford()).isInstanceOf(ActionException.class);  
   
 // Verify the results  
 }  
   
 @Test  
 void testBuyPackage() {  
 // Setup  
 user = new User("username", "pswd");  
 *when*(mockReposService.getDataPackageRepos()).thenReturn(mockDataPackageRepos);  
 *when*(mockReposService.getUserRepos()).thenReturn(mockUserRepos);  
 *when*(mockDataPackageRepos.findByPackageName("套餐1")).thenReturn(new DataPackage("套餐1", 100));  
 *when*(mockUserRepos.findByUsername("username")).thenReturn(user);  
 *when*(mockUserRepos.save(user)).thenReturn(user);  
   
 dfaParserUnderTest.transferState("套餐1");  
 // Run the test  
 dfaParserUnderTest.buyPackage();  
   
 // Verify the results  
 *verify*(mockUserRepos).save(user);  
 }  
   
 @Test  
 void testIsNumber() {  
 // Setup  
 // Run the test  
 dfaParserUnderTest.transferState("100");  
 dfaParserUnderTest.isNumber();  
   
 dfaParserUnderTest.transferState("33a");  
 *assertThatThrownBy*(() -> dfaParserUnderTest.isNumber()).isInstanceOf(ActionException.class);  
 }  
   
 @Test  
 void testIsNumber\_ThrowsActionException() {  
 // Setup  
 // Run the test  
 *assertThatThrownBy*(() -> dfaParserUnderTest.isNumber()).isInstanceOf(ActionException.class);  
 }  
   
 @Test  
 void testFindPackage() {  
 // Setup  
 *when*(mockReposService.getUserRepos()).thenReturn(mockUserRepos);  
 *when*(mockUserRepos.findByUsername("username")).thenReturn(new User("username", "dfs"));  
   
 *assertThatThrownBy*(() -> dfaParserUnderTest.findPackage()).isInstanceOf(ActionException.class);  
 }  
}

**主要对ChatBotApiController与DFAParser的方法编写了测试桩，用于进行单元测试，从而在未完成测试桩真正部分的时候，就能对局部进行测试，便于bug的发现与后期的调试。**

1. **自动化测试脚本**

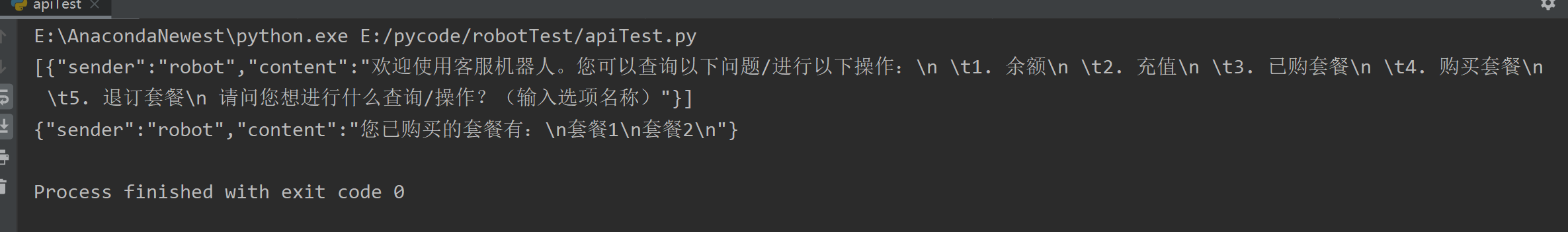
import time  
  
from selenium import webdriver  
from selenium.webdriver.common.by import By  
  
  
def sendMessage(msg):  
 global inputBox  
 global sendBtn  
 inputBox.send\_keys(msg)  
 sendBtn.click()  
 time.sleep(1)  
  
  
options = webdriver.ChromeOptions()  
options.add\_experimental\_option("detach", True)  
driver = webdriver.Chrome(options=options)  
driver.get("http://localhost:8080")  
username = driver.find\_element(By.ID, "username")  
username.send\_keys("jxy")  
password = driver.find\_element(By.ID, "password")  
password.send\_keys("jxy")  
driver.find\_element(By.XPATH, "//\*[@id='loginForm']/div/input").click()  
time.sleep(2)  
  
inputBox = driver.find\_element(By.ID, "messageInput")  
sendBtn = driver.find\_element(By.ID, "sendButton")  
  
sendMessage("余额")  
sendMessage("充值")  
sendMessage("500")  
sendMessage("已购套餐")  
sendMessage("购买套餐")  
sendMessage("套餐7")  
sendMessage("退订套餐")  
sendMessage("套餐7")  
  
time.sleep(8)  
driver.find\_element(By.ID, "logout").click()  
  
username = driver.find\_element(By.ID, "username")  
username.send\_keys("jxy")  
password = driver.find\_element(By.ID, "password")  
password.send\_keys("jxy")  
driver.find\_element(By.XPATH, "//\*[@id='loginForm']/div/input").click()  
time.sleep(2)

**这里我选择使用Selenium库进行自动化测试，模拟用户在网页上进行登录、退出、发送消息等操作的过程，借此判断整个项目是否能够按照预期的流程运行。**

**而后还用requests模块编写了用于进行模拟登录，专门对REST API进行测试的代码：**

import requests  
  
*# 登录URL和登录所需的用户名和密码*login\_url = "http://localhost:8080/login"  
apiBaseUrl = "http://localhost:8080/api/chatbot/"  
username = "jxy"  
password = "jxy"  
  
*# 创建一个会话对象*session = requests.Session()  
  
*# 发送登录请求*login\_data = {  
 "username": username,  
 "password": password  
}  
session.post(login\_url, data=login\_data)  
*# r = session.get(apiBaseUrl + "robot/jxy")  
# print(r.json())*data = {  
 "message": "余额"  
}  
session.post(apiBaseUrl + "user/jxy", json=data)  
r = session.get(apiBaseUrl + "history/jxy")  
print(r.text)  
data["message"] = "已购套餐"  
session.post(apiBaseUrl + "user/jxy", json=data)  
r = session.get(apiBaseUrl + "robot/jxy")  
print(r.text)

**测试结果如下：**

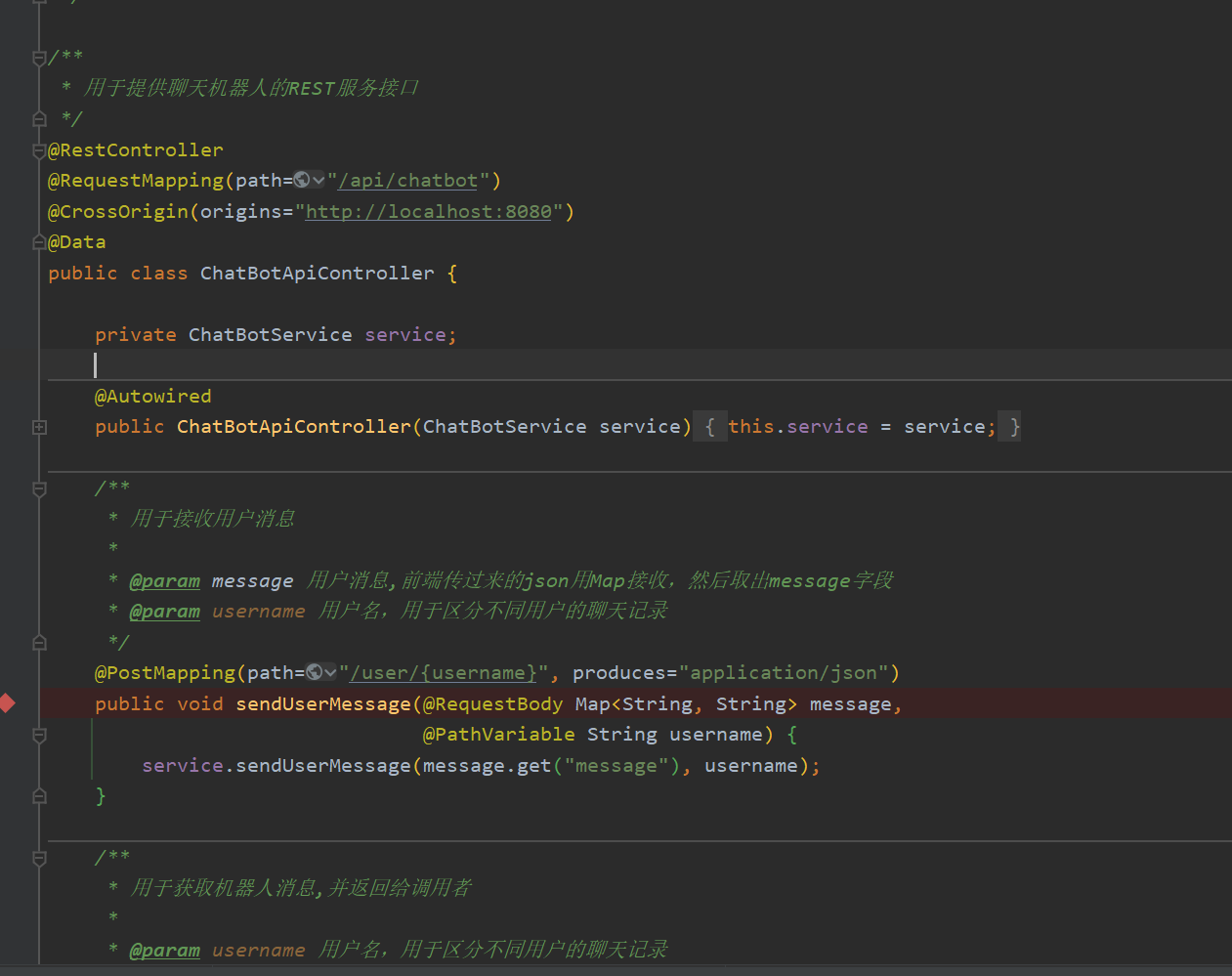


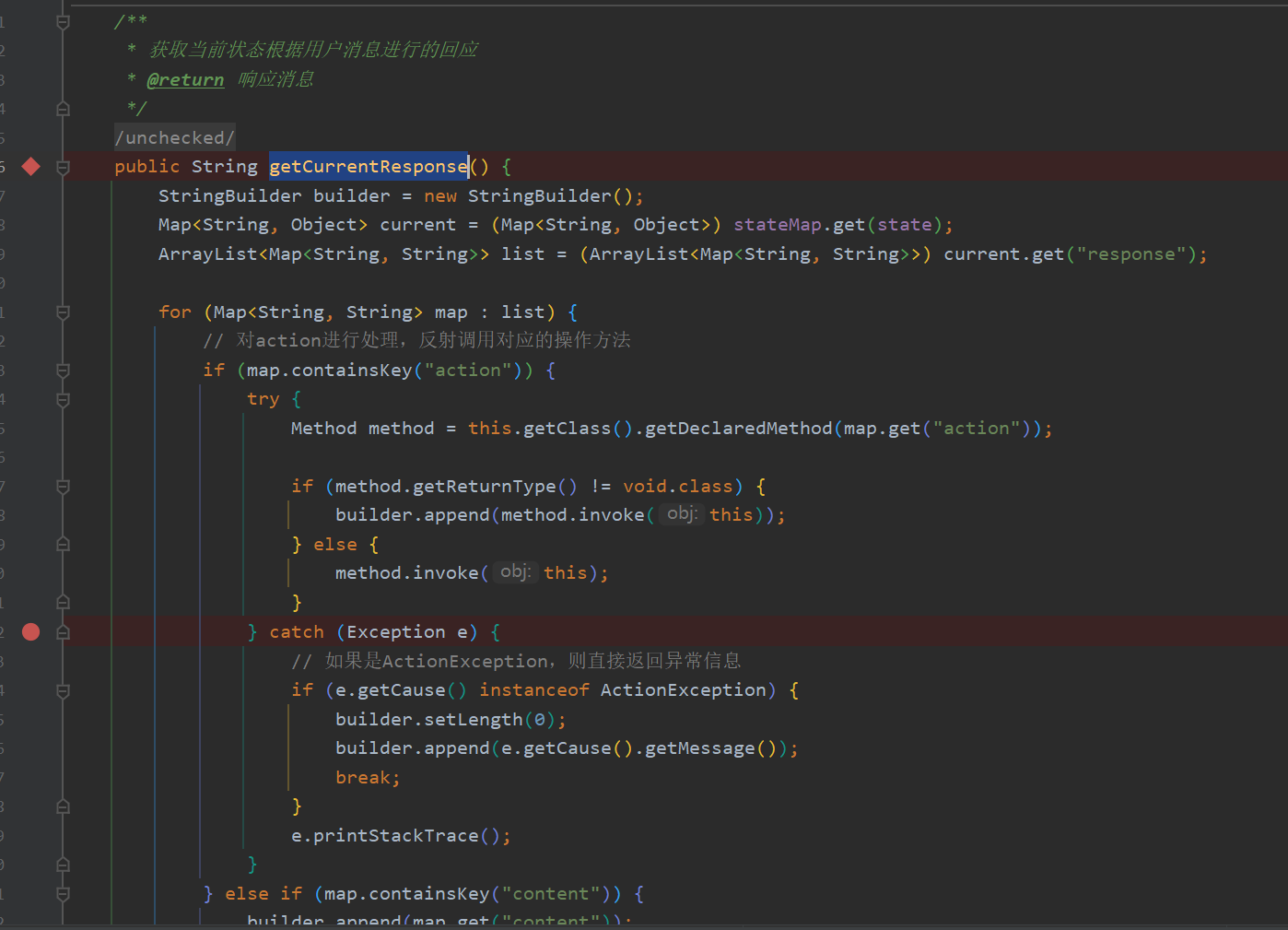
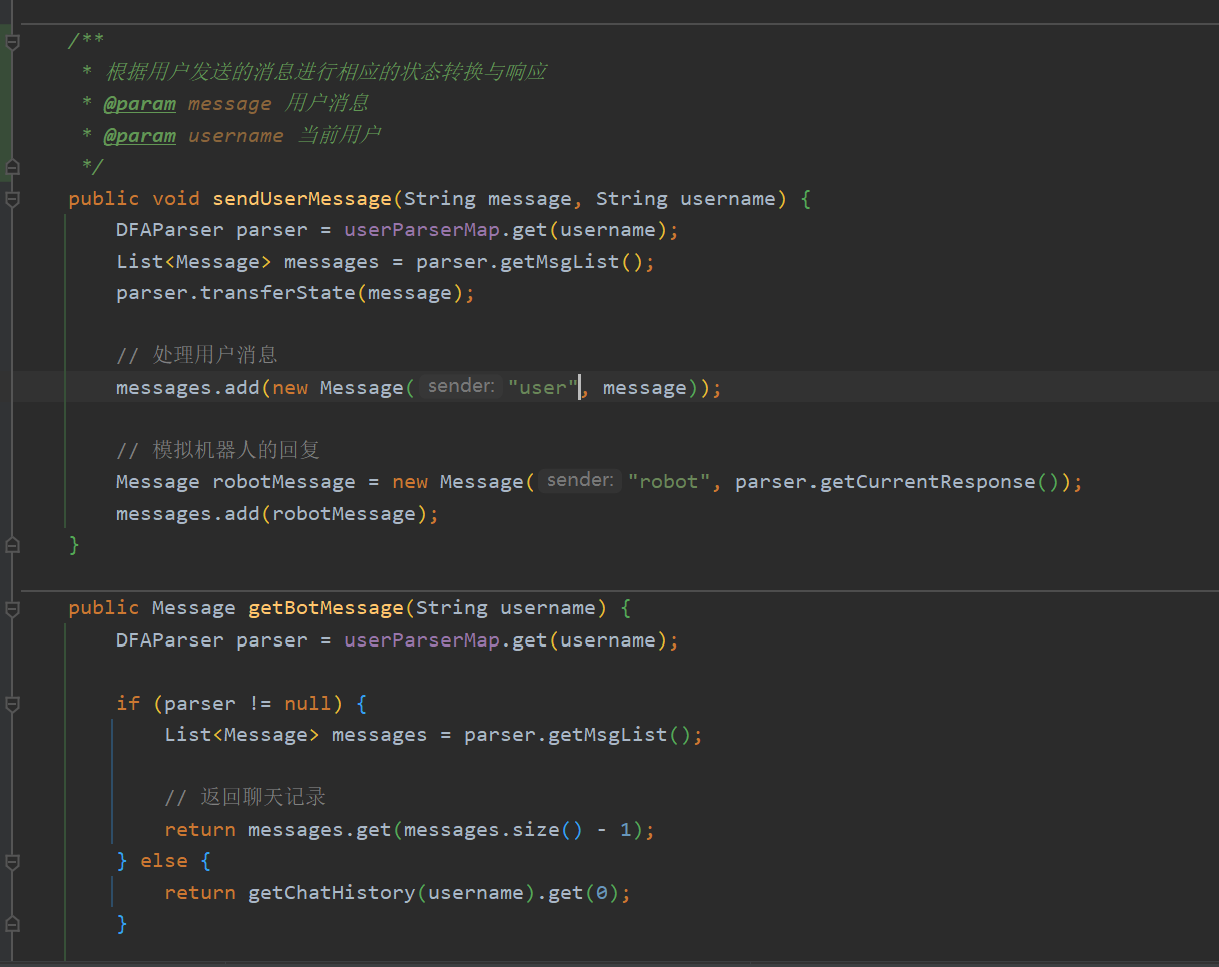
这种方法相比selenium响应更快，呈现结果更加迅速，但看起来要费力一些，不像selenium能将所有结果反应在网页上。

# 3.综合分析

## 3.1 风格：代码注释-命名-其他

1. **代码注释：采用阿里巴巴开发手册的规范，对于需要具体说明的类、属性和方法都采用块注释，对类与方法的用处进行详细说明，对于方法还会将参数与返回值进行说明，在方法内部对于语句的注释，在对应语句的上一行采用行注释。**
2. **命名：类的命名采用大驼峰命名，其余采用小驼峰命名；命名尽可能将用处表现出来，同时也为了避免过长进行适当缩写。**





## 3.2 设计和实现：数据结构-模块划分-功能-文档

1. **数据结构：2.程序设计中的2.2数据结构部分已经介绍了，这里不做过多赘述**
2. **模块划分：与文件夹的组织结构相同，在2.1节已经对结构进行说明。根据不同代码（Bean）的用处进行划分，如service主要用于具体业务逻辑的实现，web中的controller进行页面跳转的控制等。界面分为登录、注册、聊天界面三部分，对应的html文件均在resources底下的templates中。**
3. **功能：**

**登录界面：**

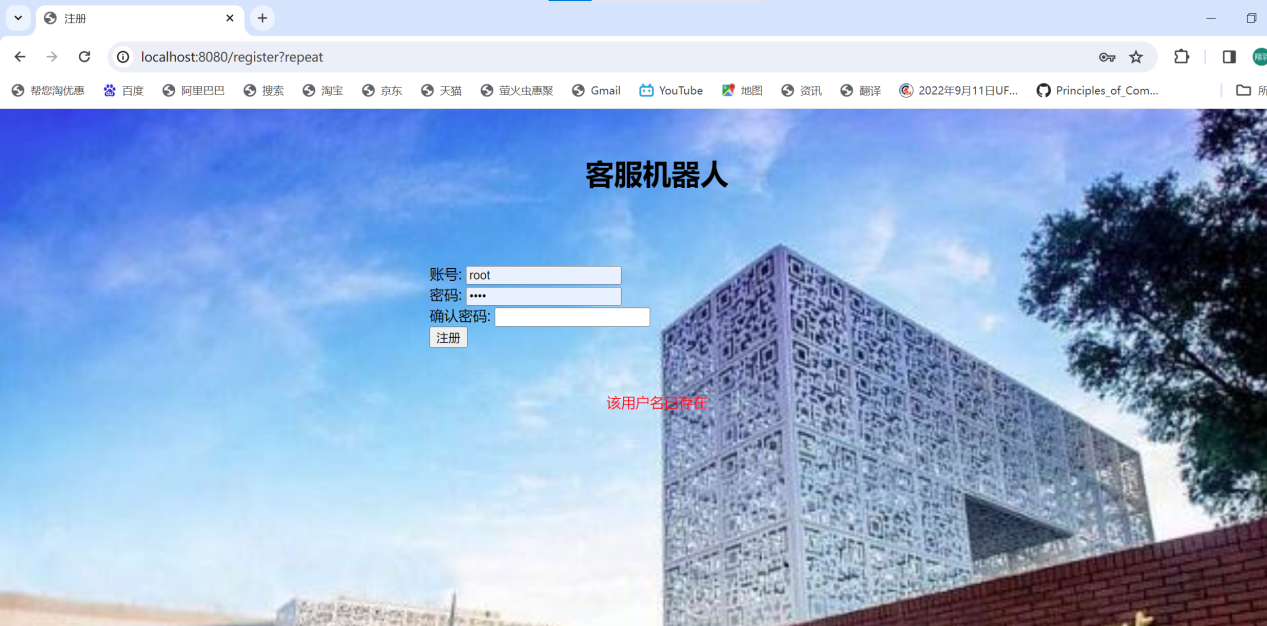


当账号密码不对时，会进行报错：

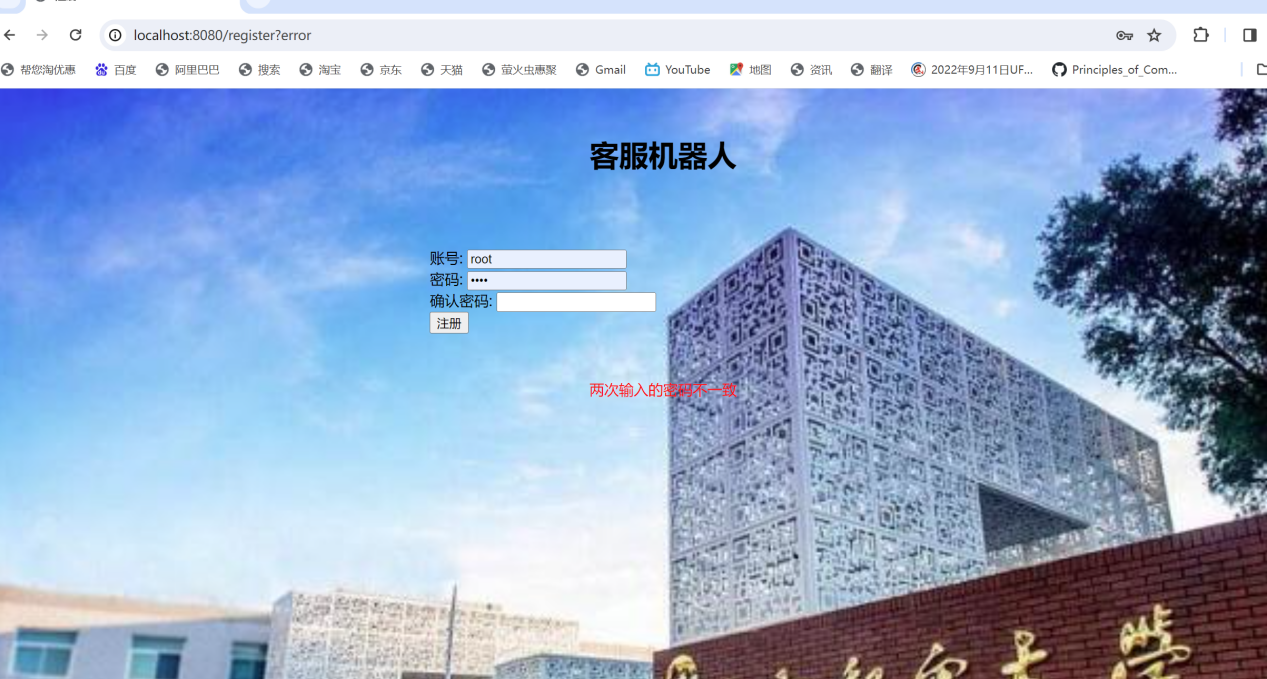


**注册界面：**

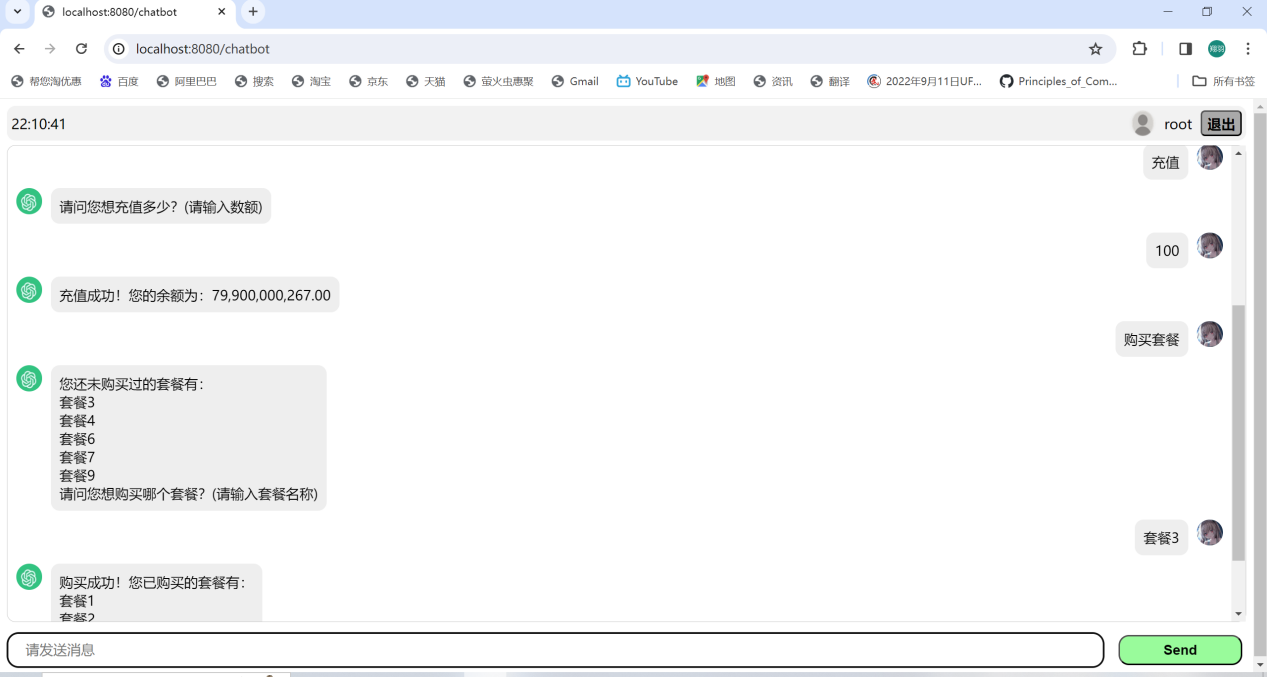


当用户名已经有人使用时：

当两次输入的密码不一样时：



**聊天界面：**



**功能：实现了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。针对不同脚本范例解释器执行之后会有不同的行为表现。在此基础上自己额外添加了显示当前用户、显示当前时间功能。**

## 3.3 接口：程序间接口-人机接口

**程序间接口:**

**不同Bean完成不同的任务，因此程序之间会互相提供调用的接口。接下来将对核心类提供的程序间接口进行详细说明与列举。**

**ChatBotService：提供的所有方法都是供ChatBotApiController调用的。**

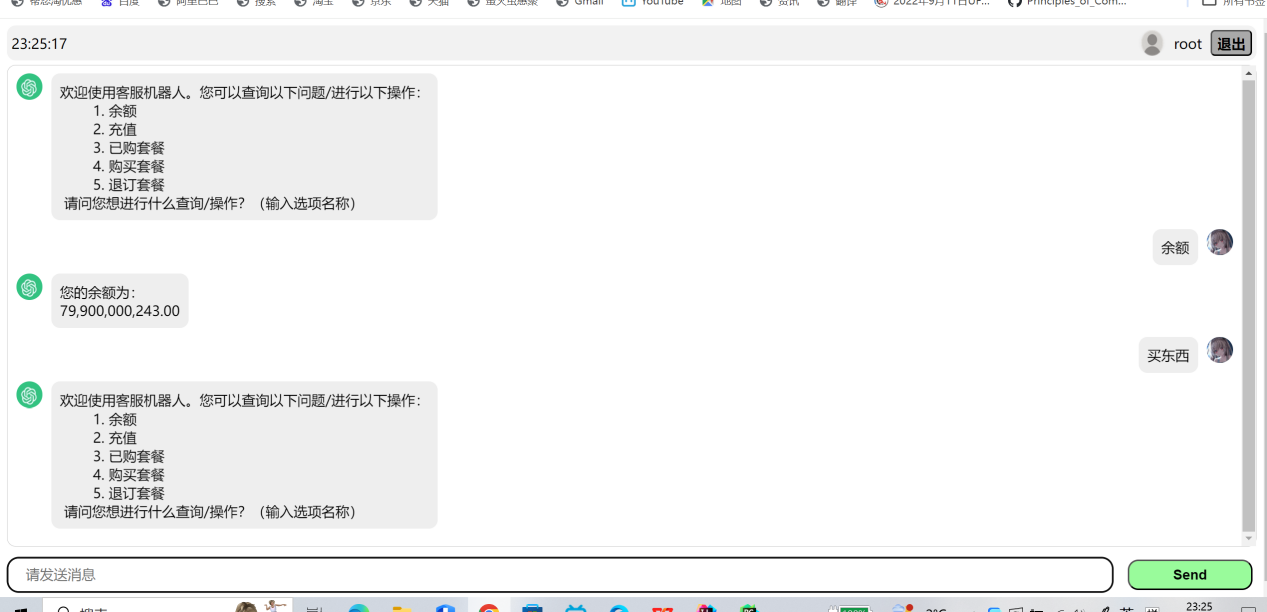
**ChatBotApiController：为前端提供请求的REST API方法。**

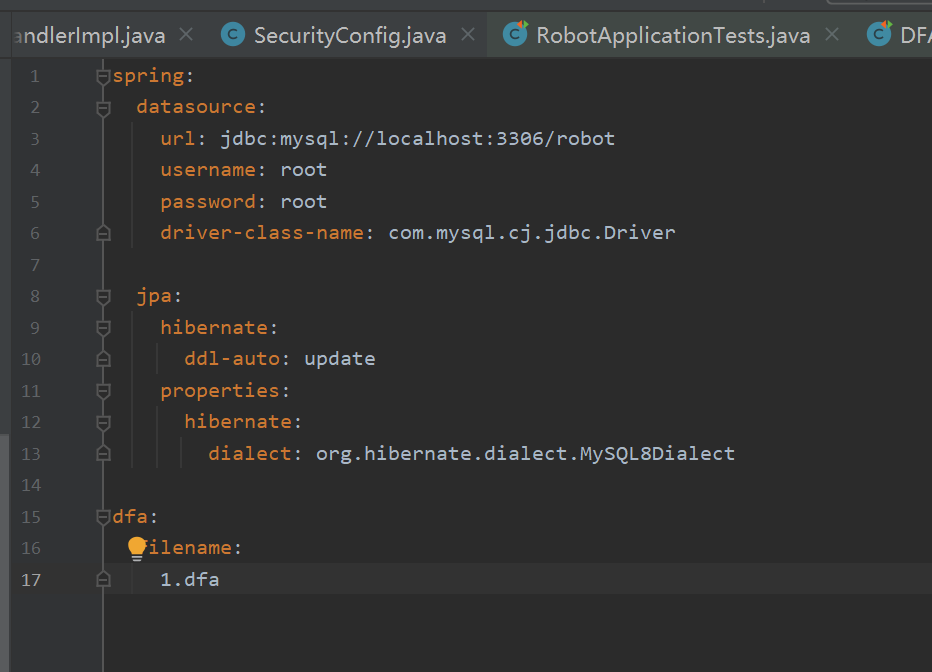
**DFAParser：getCurrentResponse与transferState供ChatBotService使用，进行当前响应内容的获取与状态转移。**

**RepositoryService：为其他类提供获取各个Repository的接口。**

**人机接口：人机接口的设计思想是按照课上提及的操作命令、GUI、配置文件等内容设计的。**

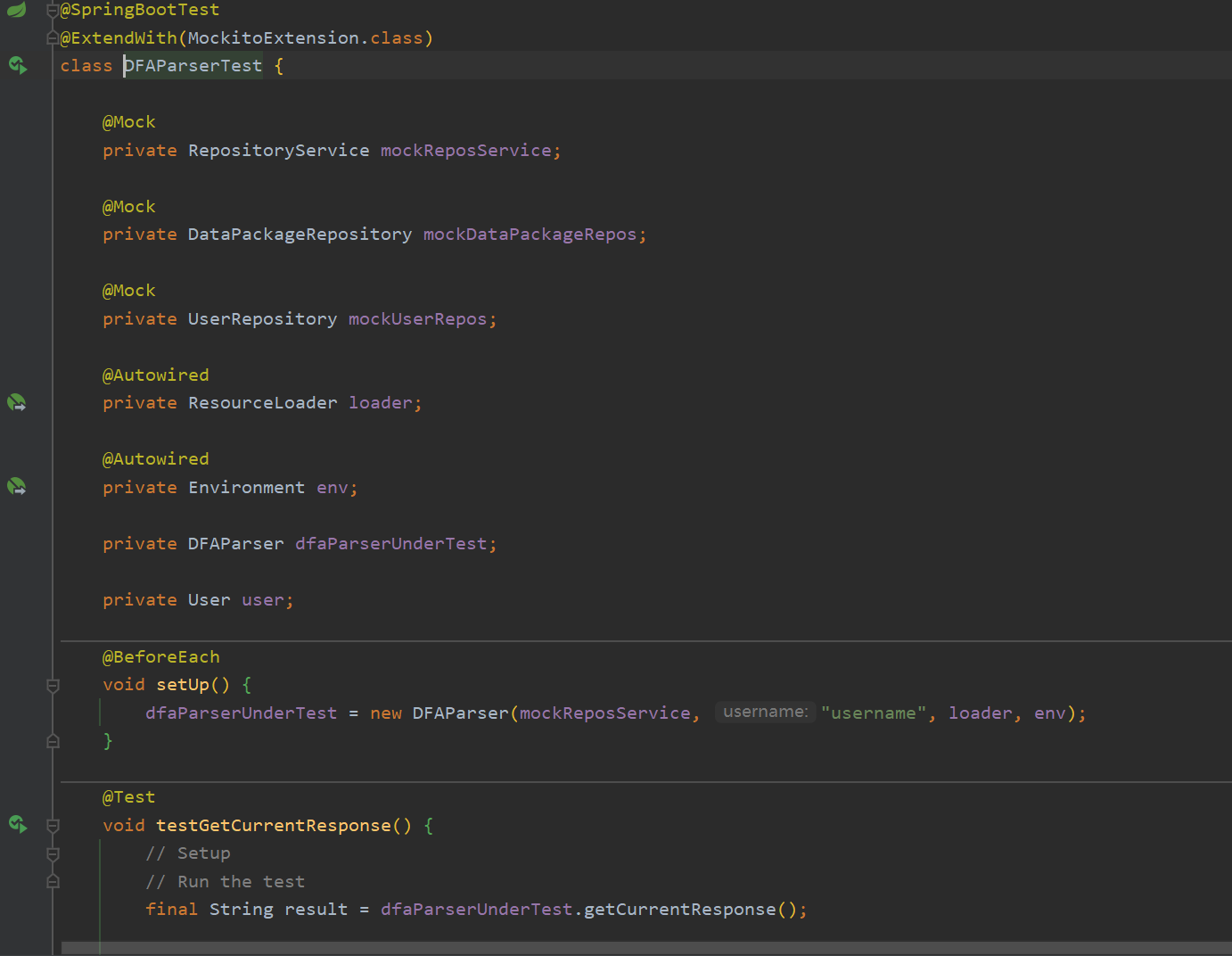
**GUI：一登陆便会发送用户可以询问的内容（可以在脚本中自行定义），若输入的内容不对，会转到设置的default状态进行相应的响应。**

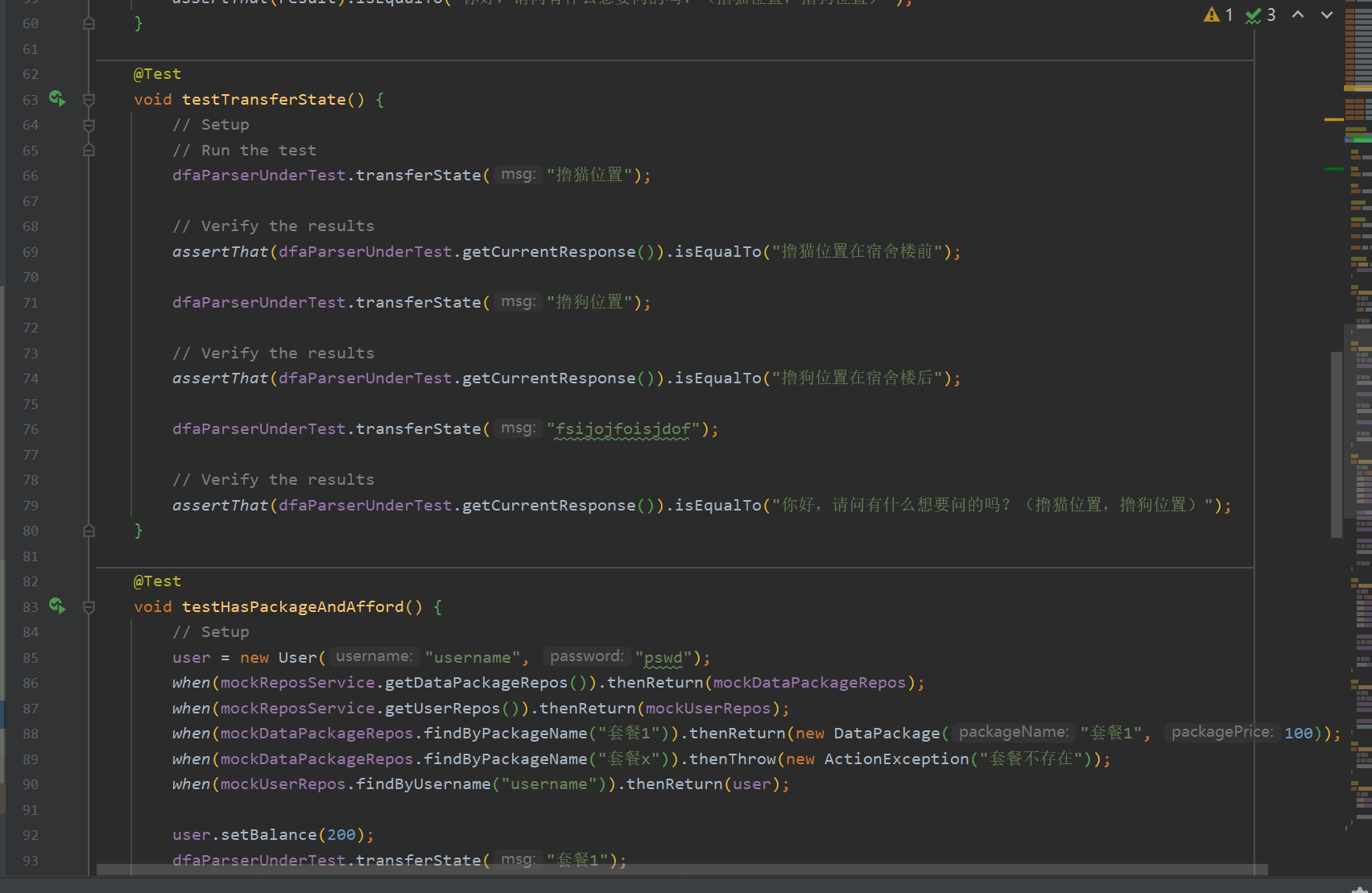


**配置文件：在application.yml中设置好采用哪个脚本，数据库等其他配置也是在该文件中定义**

## 3.4 测试：测试桩-自动化脚本测试

**测试部分在前面的第二部分的代码分析中已经提及，这里不做过多赘述，详细内容见2.3测试部分。**



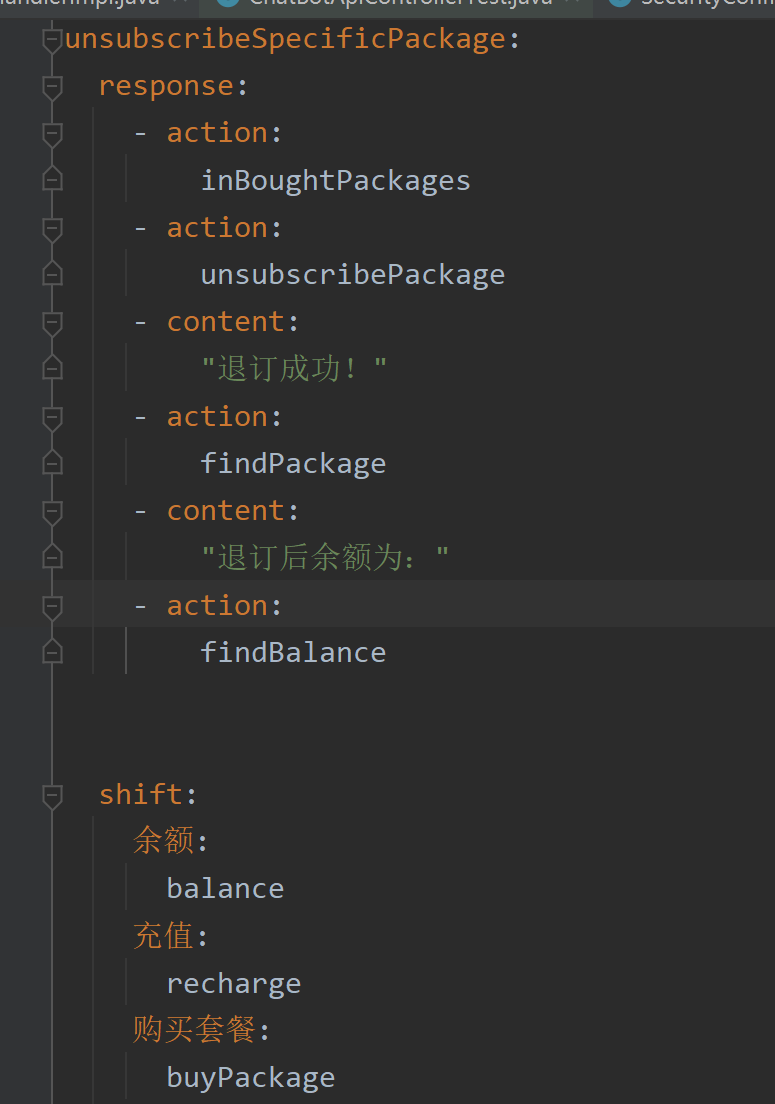


## 3.5 记法：

**脚本语言的语法描述如下：**

**1. 整体结构：**

* **脚本采用YML文件格式来编写。**
* **通过state来声明一个状态，而后在其内部对该状态的响应内容、动作以及转换进行设置，以state start来声明开始状态。**
* **每个state内部用response来声明在该状态下的回复内容，response中，可以通过content与action来组织回复内容：content后直接跟字符串，action后边跟解释器提供的动作接口；而之后的状态转移则在shift中声明，将根据下一次的输入进行相应转移，若输入的内容并没有在shift中规定，则根据编写的default分支进行跳转，若为写default分支，默认跳转到start状态。下边是一个例子：**



**2. 详细结构说明：**

**state s:**

**response:**

**- content:**

**“some content”**

**- action:**

**doAction**

**shift:**

**input1:**

**s1**

**input2:**

**s2**

**default:**

**start**

**3. 解释器提供的动作接口：**

* **hasPackageAndAfford：判断用户是否已购买输入的套餐**
* **buyPackage：购买套餐，即将套餐添加到用户的套餐列表中，并扣除相应的余额**
* **findBalance：查找当前用户余额**
* **isNumber：判断用户输入的是否为数字**
* **updateBalance：更新用户余额**
* **findAvailablePackage：查找可以购买的套餐（还未被购买的）**
* **inBoughtPackages：判断用户是否已经购买了输入的套餐**
* **unsubscribePackage：退订套餐，即将套餐从用户的套餐列表中移除，并退还相应的余额**
* **findPackage：查找已经购买的套餐**

# 4.实验总结

1. **遇到的问题：**

一开始以为Spring能自动对每一个登录的用户进行不同的处理，用户与用户之间是分离的，发送的消息不会同步，结果发现登录了用户a后，发了几个消息，用户b登录后就是在a的聊天记录的基础上进行的，使用的DFAParser都是相同的。因此采用在Model中添加user属性的方法，来区分不同用户，相互之间不可见。

1. **项目改进方案：**

将解释器提供的动作接口进一步拓展；引入更加实体与业务（如充话费、买手机、了解市场行情等），可以与网络相结合，获取网络上的实时信息进行回复；还可以结合AI技术，如引入ChatGPT等接口。

1. **总结：**

在整个项目完成的过程中，自己首先是确实学习到了如何规范编写代码，完成代码结构清晰，可读性强这一系列目标。其次是合理设计程序，学会了对系统进行了合理的设计，考虑到了各个方面的需求，合理划分层次结构。并且在做的过程中，也对Spring相关技术更加熟悉，学习到了挺多相关知识。最后是解决工程问题，成功解决了领域特定语言的设计和实现问题，满足了客服机器人自动应答逻辑的要求。