

# 深圳大学课程作业报告

课程名称：Java 程序设计

课程作业名称：结合大模型的 Java 知识问答机器人

学院：计算机与软件学院

专业：计算机科学与技术

指导教师：潘微科

报告人：曹秦鲁 学号：2023150203 班级：国际班

作业时间：2024 年 11 月 22 日（周五）~2024 年 12 月 25 日（周三）

课程作业报告提交时间：

教务部制

### 作业目的与要求：

**课程作业目的：**Java 程序设计综合练习，含 GUI、多线程、I/O、网络通信等。

### 课程作业要求：

(1). 编写 Java 应用程序，实现“结合大模型的 Java 知识问答机器人”。

- 人机交互界面：使用命令行窗口（不得分）或图形用户界面（10 分）
- 大模型：允许用户从腾讯混元大模型、百度文心一言大模型、阿里通义千问大模型或其他大模型中（至少支持两个大模型）选择使用一个大模型（20 分）
- 多线程&网络通信：能够并行处理多个用户的网络请求，每个用户请求都应该在一个新的线程中处理，以避免阻塞主线程（20 分）
- 测试案例：要求测试 Java 程序设计方面的选择题、判断题、程序填空题和编程题（每个题型至少测试两个例子），并就测试效果做出简单分析（20 分）
- 日志记录：在本地文本文件（.txt）中记录人机对话过程（20 分）
- 特色功能：可以列出自己实现的其他特色功能（每个特色功能 5 分，最多记两个特色功能共 10 分）

在报告中附上程序截图、完整的运行结果截图和详细的文字说明。

报告写作。要求：主要思路有明确的说明，重点代码有详细的注释，行文逻辑清晰可读性强，报告整体写作较为专业。（20 分）

### 说明：

(1) 本次课程作业满分为 100 分，占总成绩的比例（待定）。

(2) 本次课程作业截至时间 2024 年 12 月 25 日（周\*\*）21:59。

(3) 报告正文：请在**指定位置填写**，本次课程作业**需要单独提交源程序文件**（源程序单独打包在 Blackboard 中上传，不要包含外部导入的包）。

(4) 个人信息：WORD 文件名中的“姓名”、“学号”，请改为你的**姓名和学号**；课程作业报告的首页，请准确填写“学院”、“专业”、“报告人”、“学号”、“班级”、“课程作业报告提交时间”等信息。

(5) 提交方式：截至时间前，请在 Blackboard 平台中提交。

(6) 发现抄袭（包括复制&粘贴整句话、整张图），**抄袭者和被抄袭者的成绩记零分**。

(7) 延迟提交，不得分；如有特殊情况，请于截至日期之后的**48 小时内**发邮件到 panweike@szu.edu.cn，并在邮件中注明课程名称、作业名称、姓名、学号等信息，以及特殊情况的说明，我收到后会及时回复。

(8) 期末考试阶段补交无效。

(1). 编写 Java 应用程序，实现“结合大模型的 Java 知识问答机器人”。

- 人机交互界面：使用命令行窗口（不得分）或图形用户界面（10 分）
- 大模型：允许用户从腾讯混元大模型、百度文心一言大模型、阿里通义千问大模型或其他大模型中（至少支持两个大模型）选择使用一个大模型（20 分）
- 多线程&网络通信：能够并行处理多个用户的网络请求，每个用户请求都应该在一个新的线程中处理，以避免阻塞主线程（20 分）
- 测试案例：要求测试 Java 程序设计方面的选择题、判断题、程序填空题和编程题（每个题型至少测试两个例子），并就测试效果做出简单分析（20 分）
- 日志记录：在本地文本文件（.txt）中记录人机对话过程（20 分）
- 特色功能：可以列出自己实现的其他特色功能（每个特色功能 5 分，最多记两个特色功能共 10 分）

在报告中附上程序截图、完整的运行结果截图和详细的文字说明。

# 1. 项目概述

本项目旨在开发一个结合大模型的 Java 知识问答机器人，用户可以通过图形用户界面（GUI）与机器人进行交互，选择不同的大模型（如阿里通义千问、百度文心一言）来获取问题的答案。项目实现了多线程处理，确保用户请求不会阻塞主线程，并且能够记录聊天日志到本地文件。

## 需求分析

人机交互界面：使用图形用户界面（GUI）进行交互。

大模型选择：支持至少两种大模型（阿里通义千问、百度文心一言）。

多线程处理：每个用户请求在新的线程中处理，避免阻塞主线程。

日志记录：记录聊天过程到本地文本文件。

测试案例：编写测试案例以验证程序功能。

# 2. 系统架构

## 2.1 核心组件

- **JavaKnowledgeBot**: 主界面类，继承自 **JFrame**，负责 **UI** 展示和用户交互
- **ModelAPI**: 统一的 **API** 接口定义
- **AliyunAPI**: 阿里云通义千问 **API** 实现
- **BaiduAPI**: 百度文心一言 **API** 实现
- **ModelAPIFactory**: **API** 工厂类，负责创建具体的 **API** 实例
- **Config**: 配置类，存储 **API** 密钥等配置信息
- **StreamingResponse**: 流式输出处理类，实现打字机效果 **特色功能**
- **CodeHighlighter**: 代码高亮处理类，支持多种编程语言的语法高亮 **特色功能**

## 2.2 技术栈

- 界面框架: **Java Swing**
- 网络请求: **OkHttp3**（支持异步请求和超时控制）
- **JSON** 处理: **org.json**
- 并发处理: **Java ExecutorService**（线程池管理）
- 文本处理: **Java Regex**（正则表达式处理）
- **API** 集成:
  - 阿里云通义千问 **API**
  - 百度文心一言 **API**
- 配置管理: **Properties**（配置文件管理）
- 文件操作: **Java IO**（日志和历史记录管理）

## 3. 功能特性

### 3.1 用户界面

- 厂商选择：支持在阿里云和百度 AI 模型间切换
- 聊天界面：显示对话历史
- 输入区域：用户输入框和发送按钮
- 实时响应：异步处理 AI 响应，不阻塞 UI

### 3.2 对话功能

- 多模型支持：支持与不同 AI 服务商的模型对话
- 历史记录：自动保存对话历史到本地文件
- 实时切换：支持在不同模型间无缝切换
- 异步处理：使用线程池处理 AI 请求，保证 UI 响应性

### 3.3 数据持久化

- 对话日志：按厂商分别保存对话记录
- 配置管理：通过配置文件管理 API 密钥等敏感信息

## 4. 代码实现详解

### 4.1 主界面实现 (JavaKnowledgeBot.java)

主要功能：

- 初始化图形界面组件
- 处理用户输入和发送消息
- 管理对话历史记录
- 处理模型切换
- 流式输出控制            特色功能
- 代码高亮显示            特色功能

代码解释：

1) 类定义和成员变量：

```
public class JavaKnowledgeBot extends JFrame {  
  
    private JComboBox<String> vendorComboBox;    // 模型选择下拉框  
  
    private JTextPane chatTextPane;              // 聊天记录显示区域  
  
    private JTextField inputTextField;            // 用户输入框  
  
    private JButton sendButton;                   // 发送按钮  
  
    private ExecutorService threadPool;           // 线程池，用于异步处理请求  
  
    private StreamingResponse streamingResponse;   // 流式输出管理器  
  
    private JButton pauseButton;                  // 暂停/继续按钮  
  
    private JSlider speedSlider;                  // 速度调节滑块  
  
}
```

```
private static final String[] VENDORS = {"阿里-通义千问", "百度-文心一言"}; // 支持的AI 模型
}

=
```

## 2) 构造函数实现:

```
public JavaKnowledgeBot() {
    setTitle("知识问答机器人");
    setSize(800, 600);
    setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
    setLocationRelativeTo(null);
    setMinimumSize(new Dimension(600, 400));
    addWindowListener(new java.awt.event.WindowAdapter() {
        @Override
        public void windowClosing(java.awt.event.WindowEvent windowEvent) {
            closeApplication();
        }
    });
    // 初始化线程池
    threadPool = Executors.newCachedThreadPool();
    // 创建布局
    setLayout(new BorderLayout(5, 5));
    // 创建所有面板和组件
    createPanels();

    // 初始化流式输出管理器
    streamingResponse = new StreamingResponse(chatTextPane);

    // 添加控制面板
    addControlPanel();
    // 初始加载默认厂商的聊天记录
    loadChatRecord();
}
```

## 3) 消息发送处理:

```java

```
private void onSendMessage(ActionEvent e) {
    String userInput = inputTextField.getText().trim();
    if (!userInput.isEmpty()) {
        // 显示用户消息
        chatTextArea.append("User: " + userInput + "\n");
        saveChatRecord("User: " + userInput);
        inputTextField.setText("");
        // 异步处理AI 响应
        threadPool.submit(() -> {
            try {
```

```

        ModelAPI api = createAPIInstance((String) vendorComboBox.getSelectedItem());
        String modelResponse = api.call(userInput);

        SwingUtilities.invokeLater(() -> {
            chatTextArea.append("Bot: " + modelResponse + "\n");
            saveChatRecord("Bot: " + modelResponse);
        });
    } catch (Exception ex) {
        ex.printStackTrace();
        SwingUtilities.invokeLater(() -> {
            chatTextArea.append("Error: " + ex.getMessage() + "\n");
        });
    }
}
}
}

```

#### 4.2 API 接口定义 (ModelAPI.java)

```

public interface ModelAPI {
    // 统一的 API 调用接口，接收用户输入，返回 AI 响应
    String call(String userInput) throws Exception;
}

```

这个接口定义了所有 AI 模型实现必须提供的方法，确保了不同模型实现的一致性。

#### 4.3 阿里云 API 实现 (AliyunAPI.java)

阿里云 API 使用官方提供的 SDK 进行调用，不需要手动管理 URL 和 HTTP 请求。

```

public class AliyunAPI implements ModelAPI {
    private final String apiKey;
    private final String model;

    public AliyunAPI() {
        // 从配置类获取 API 密钥和模型名称
        this.apiKey = Config.ALIYUN_API_KEY;
        this.model = Config.ALIYUN_MODEL;
    }

    @Override
    public String call(String userInput) throws ApiException {
        // 创建通义千问 API 调用实例
        Generation gen = new Generation();

        // 设置系统角色消息
        Message systemMsg = Message.builder()
            .role(Role.SYSTEM.getValue())
            .content("You are a helpful assistant.")
            .build();
    }
}

```

```

// 设置用户输入消息
Message userMsg = Message.builder()
    .role(Role.USER.getValue())
    .content(userInput)
    .build();

// 构建API 调用参数
GenerationParam param = GenerationParam.builder()
    .apiKey(apiKey) // 设置API 密钥
    .model(model) // 设置模型名称
    .messages(Arrays.asList(systemMsg, userMsg)) // 设置消息列表
    .resultFormat(GenerationParam.ResultFormat.MESSAGE) // 设置返回格式
    .build();

// 调用API 并获取响应
GenerationResult result = gen.call(param);

// 从结果中提取回复内容
return result.getOutput().getChoices().get(0).getMessage().getContent();
}
}

```

主要特点：

#### 1. SDK 集成

- 使用官方 SDK 管理 API 调用
- 自动处理认证和请求
- 提供类型安全的 API

#### 2. 消息构建

- 支持系统角色设置
- 支持多轮对话
- 结构化的消息格式

#### 3. 参数配置

- 灵活的参数设置
- 支持多种模型选择
- 可配置返回格式

#### 4. 错误处理

- SDK 级别的异常处理
- 类型安全的响应解析
- 完整的错误信息

### 4.4 百度 API 实现 (BaiduAPI.java)

代码解释：

```

public class BaiduAPI implements ModelAPI {
    private final String apiKey;
    private final String secretKey;
    private final String apiUrl;
    private final OkHttpClient httpClient;
}

```



```

public BaiduAPI() {
    this.apiKey = Config.BAIDU_API_KEY;
    this.secretKey = Config.BAIDU_SECRET_KEY;
    this.apiUrl = Config.BAIDU_API_URL;

    // 配置 OkHttpClient, 添加超时设置
    this.httpClient = new OkHttpClient.Builder()
        .connectTimeout(30, TimeUnit.SECONDS)
        .writeTimeout(30, TimeUnit.SECONDS)
        .readTimeout(60, TimeUnit.SECONDS)
        .build();
}

@Override
public String call(String userInput) throws Exception {
    try {
        String accessToken = getAccessToken();
        String url = apiUrl + accessToken;
        // 构建请求体
        JSONObject requestBody = new JSONObject();
        JSONArray messages = new JSONArray();
        JSONObject userMessage = new JSONObject();
        userMessage.put("role", "user");
        userMessage.put("content", userInput);
        messages.put(userMessage);
        requestBody.put("messages", messages);
        // 发送请求
        Request request = new Request.Builder()
            .url(url)
            .post(RequestBody.create(
                MediaType.parse("application/json"),
                requestBody.toString()
            ))
            .addHeader("Content-Type", "application/json")
            .build();

        try (Response response = httpClient.newCall(request).execute()) {
            if (!response.isSuccessful()) {
                throw new IOException("API 调用失败: " + response.code() + " " + response.message());
            }

            JSONObject jsonResponse = new JSONObject(response.body().string());

            if (jsonResponse.has("error_code")) {
                throw new IOException("API 错误: " +
                    jsonResponse.getInt("error_code") + " - " +

```

```

        jsonResponse.getString("error_msg"));
    }
    return jsonResponse.getString("result");
}
} catch (Exception e) {
    System.err.println("API 调用出错: " + e.getMessage());
    throw e;
}
}
/**
 * 获取访问令牌
 */
private String getAccessToken() throws IOException {
    String tokenUrl = "https://aip.baidubce.com/oauth/2.0/token";
    RequestBody body = new FormBody.Builder()
        .add("grant_type", "client_credentials")
        .add("client_id", apiKey)
        .add("client_secret", secretKey)
        .build();
    Request request = new Request.Builder()
        .url(tokenUrl)
        .post(body)
        .addHeader("Content-Type", "application/x-www-form-urlencoded")
        .build();
    try (Response response = httpClient.newCall(request).execute()) {
        if (!response.isSuccessful()) {
            throw new IOException("认证失败: " + response.code() + " " + response.message());
        }
        JSONObject jsonResponse = new JSONObject(response.body().string());
        return jsonResponse.getString("access_token");
    }
}
}
}

```

#### 4.5 工厂类实现 (ModelAPIFactory.java)

代码解释:

```

public class ModelAPIFactory {
    public static ModelAPI createAPI(String vendorModel) {
        // 根据选择的厂商创建对应的API 实例
        if (vendorModel.contains("阿里")) {
            return new AliyunAPI();
        } else if (vendorModel.contains("百度")) {
            return new BaiduAPI();
        }
    }
}

```

```
        throw new IllegalArgumentException("不支持的 AI 模型: " + vendorModel);
    }
}
```

## 工厂模式 (Factory Pattern)

工厂模式是一种创建型设计模式，它提供了一种创建对象的最佳方式。在本项目中，工厂模式主要用于创建不同 AI 模型的 API 实例。

### 工厂模式的 2 个核心作用

#### a. 封装对象创建过程

- 客户端代码不需要知道具体类的创建细节
- 示例：

// 不使用工厂模式

```
if (vendor.equals("阿里")) {
    api = new AliyunAPI();
    api.setConfig(...);
    api.initialize(...);
} else if (vendor.equals("百度")) {
    api = new BaiduAPI();
    api.setCredentials(...);
    api.setup(...);
}
```

// 使用工厂模式

```
ModelAPI api = ModelAPIFactory.createAPI(vendor); // 一行代码搞定
...
```

#### b. 便于扩展新功能

- 添加新的 AI 模型只需要三步：

1. 创建新的 API 实现类
2. 实现 ModelAPI 接口
3. 在工厂类中添加创建逻辑

- 示例：添加讯飞星火模型

// 1. 创建新的 API 实现类

```
public class XunfeiAPI implements ModelAPI {
    @Override
    public String call(String userInput) throws Exception {
        // 讯飞星火 API 的具体实现
    }
}
```

// 2. 在工厂类中添加创建逻辑

```
public class ModelAPIFactory {
    public static ModelAPI createAPI(String vendorModel) {
```

```

        if (vendorModel.contains("阿里")) {
            return new AliyunAPI();
        } else if (vendorModel.contains("百度")) {
            return new BaiduAPI();
        } else if (vendorModel.contains("讯飞")) {
            return new XunfeiAPI(); // 新增的模型
        }
        throw new IllegalArgumentException("不支持的 AI 模型");
    }
}

```

## 4.6 配置类实现 (Config.java)

配置类的主要作用是存储和管理 API 密钥等配置信息。通过静态常量的方式提供配置访问。

```

public class Config {
    // 阿里云配置
    public static final String ALIYUN_API_KEY = "sk-2c51c448693f437ab806cea4b110d2a9";
    public static final String ALIYUN_MODEL = "qwen-plus";
    // 百度配置
    public static final String BAIDU_API_KEY = "RxSzjmUWJQANqe3GLVN2N6rp";
    public static final String BAIDU_SECRET_KEY = "dQ1XFW9pG9VpGLJQXg94VawLBPFKCjU2";
    public static final String BAIDU_API_URL = "https://aip.baidubce.com/rpc/2.0/ai_custom/v1/wenxinwor
kshop/chat/completions?access_token=";
    // 禁止实例化
    private Config() {}
}

```

主要特点：

1. 配置集中管理
  - 所有 API 密钥和 URL 集中存储
  - 便于统一修改和维护
  - 避免配置散落在代码中
2. 静态访问
  - 使用静态常量提供配置
  - 无需创建实例即可访问
  - 私有构造函数防止实例化
3. 分组管理
  - 按照不同 AI 服务商分组
  - 配置项命名清晰
  - 注释说明配置用途

### 4. 使用示例

```

// 在 API 类中使用配置
public class BaiduAPI implements ModelAPI {

```

```

private final String apiKey;

private final String secretKey;


public BaiduAPI() {

    this.apiKey = Config.BAIDU_API_KEY;

    this.secretKey = Config.BAIDU_SECRET_KEY;

}

}

```

以下为两个特色功能的实现：**流式输出**和**输出代码高亮**

#### 4.7 流式输出实现 (StreamingResponse.java)

代码解释：

```

/**
 * 流式输出处理类
 * 实现打字机效果的文本输出，支持暂停、继续、跳过和速度调节功能
 */

public class StreamingResponse {

    // UI 组件：用于显示文本的面板

    private final JTextPane textPane;


    // 控制标志

    private volatile boolean isPaused = false;    // 暂停标志，volatile 确保多线程可见性

    private volatile boolean skipToEnd = false;    // 跳过标志，直接显示全部内容


    // 打字速度控制（毫秒/字符）

    private int typingSpeed = 50;    // 默认 50ms 输出一个字符


    // 调度器：用于定时输出字符

    private ScheduledExecutorService scheduler;

    /**
     * 构造函数
     * @param textPane 目标文本显示面板
     */

    public StreamingResponse(JTextPane textPane) {

        this.textPane = textPane;

        // 创建单线程调度器，用于顺序输出字符

        this.scheduler = Executors.newSingleThreadScheduledExecutor();

    }

    /**
     * 流式输出文本
     * 将文本按照打字机效果逐字符显示
     * @param response 要显示的完整文本
     */
}

```

```

public void streamResponse(String response) {
    if (response == null || response.isEmpty()) return;

    // 当前文本缓冲区
    StringBuilder currentText = new StringBuilder(textPane.getText());
    // 当前输出位置
    final int[] currentIndex = {0};

    // 创建定时任务
    ScheduledFuture<?>[] future = {null};
    future[0] = scheduler.scheduleAtFixedRate(() -> {
        // 处理跳过请求
        if (skipToEnd) {
            // 直接显示所有剩余文本
            currentText.append(response.substring(currentIndex[0]));
            updateTextPane(currentText.toString());
            future[0].cancel(false);
            skipToEnd = false;
            currentIndex[0] = response.length();
            return;
        }

        // 正常流式输出处理
        if (!isPaused && currentIndex[0] < response.length()) {
            // 追加下一个字符
            currentText.append(response.charAt(currentIndex[0]));
            updateTextPane(currentText.toString());
            currentIndex[0]++;

            // 检查是否完成输出
            if (currentIndex[0] >= response.length()) {
                future[0].cancel(false);
            }
        }
    }, 0, typingSpeed, TimeUnit.MILLISECONDS);
}

/**
 * 暂停输出
 */
public void pause() {
    isPaused = true;
}

/**
 * 继续输出

```

```

    */

    public void resume() {
        isPaused = false;
    }

    /**
     * 跳过动画直接显示全部内容
     */

    public void skipToEnd() {
        skipToEnd = true;
    }

    /**
     * 设置打字速度
     * @param speed 每个字符的显示间隔（毫秒）
     */

    public void setTypingSpeed(int speed) {
        // 限制速度在10-200ms 之间
        this.typingSpeed = Math.max(10, Math.min(200, speed));
    }

    /**
     * 更新文本显示
     * 在 EDT 线程中安全地更新 UI
     */

    private void updateTextPane(String text) {
        SwingUtilities.invokeLater(() -> {
            textPane.setText("");
            // 使用CodeHighlighter 处理代码高亮
            CodeHighlighter.insertText(textPane, text, false);
            // 滚动到最新位置
            textPane.setCaretPosition(textPane.getDocument().getLength());
        });
    }

    /**
     * 清理资源
     * 关闭调度器，防止内存泄漏
     */

    public void cleanup() {
        if (scheduler != null && !scheduler.isShutdown()) {
            scheduler.shutdownNow();
        }
    }
}

```

#### 4.8 代码高亮实现 (CodeHighlighter.java)

代码解释：

```

/*
 * 代码高亮处理类
 * 支持多种编程语言的语法高亮显示，实现 GitHub 风格的代码块
 */

public class CodeHighlighter {

    // 代码块样式颜色定义

    private static final Color BACKGROUND_COLOR = new Color(246, 248, 250); // GitHub 风格背景色
    private static final Color HEADER_COLOR = new Color(240, 242, 244); // 代码块头部背景色

    // 语法高亮颜色定义

    private static final Color KEYWORD_COLOR = new Color(207, 34, 46); // 关键字 - 红色
    private static final Color STRING_COLOR = new Color(17, 99, 41); // 字符串 - 绿色
    private static final Color COMMENT_COLOR = new Color(110, 119, 129); // 注释 - 灰色
    private static final Color NORMAL_CODE_COLOR = new Color(36, 41, 47); // 普通代码 - 深灰

    // 语法规则定义（正则表达式）

    private static final Pattern KEYWORDS = Pattern.compile(
        "\\b(def|class|import|from|return|if|else|while|for|try|except|in|is|not|and|or|lambda|None|True|False)\\b"
    );

    private static final Pattern STRINGS = Pattern.compile(
        "\"[^\"]\\\\\\\\*(\\\\\\\\.[^\"]\\\\\\\\)*\\\\\\\\\"|'[^']*\\\\\\\\*(\\\\\\\\.[^']*\\\\\\\\)*\\\\\\\\'"
    );

    private static final Pattern COMMENTS = Pattern.compile("#.*$", Pattern.MULTILINE);

    // 代码块匹配模式

    private static final Pattern CODE_BLOCK_PATTERN = Pattern.compile(
        "```(?:.|\n)*```"
    );

    /**
     * 插入带格式的文本
     * @param textPane 目标文本面板
     * @param text 要插入的文本
     * @param isUserMessage 是否是用户消息
     */

    public static void insertText(JTextPane textPane, String text, boolean isUserMessage) {

        try {

            StyledDocument doc = textPane.getStyledDocument();

            // 创建代码块基本样式

            Style codeStyle = textPane.addStyle("codeStyle", null);

            StyleConstants.setFontFamily(codeStyle, "Consolas");

```



```

        StyleConstants.setFontSize(codeStyle, 14);
        StyleConstants.setBackground(codeStyle, BACKGROUND_COLOR);

        // 查找并处理代码块

        Matcher matcher = CODE_BLOCK_PATTERN.matcher(text);
        while (matcher.find()) {

            // 提取代码块信息

            String language = matcher.group(1).trim(); // 编程语言
            String code = matcher.group(2);           // 代码内容

            // 添加代码块头部

            doc.insertString(doc.getLength(), "\n", null);

            // 创建并应用头部样式

            Style headerStyle = textPane.addStyle("headerStyle", null);
            StyleConstants.setBackground(headerStyle, HEADER_COLOR);
            StyleConstants.setFontFamily(headerStyle, "微软雅黑");
            StyleConstants.setFontSize(headerStyle, 12);
            doc.insertString(doc.getLength(), language + "\n", headerStyle);

            // 添加复制按钮

            JButton copyButton = new JButton("Copy code");
            copyButton.setFont(new Font("微软雅黑", Font.PLAIN, 12));
            copyButton.setFocusPainted(false);
            copyButton.addActionListener(e -> copyToClipboard(code));

            // 将按钮添加到文本面板

            StyleConstants.setComponent(codeStyle, copyButton);
            doc.insertString(doc.getLength(), " ", codeStyle);

            // 处理代码高亮

            highlightCode(doc, code, language);
        }
    } catch (BadLocationException e) {
        e.printStackTrace();
    }
}

/**
 * 代码高亮处理
 * 对代码进行语法分析并应用相应的颜色样式
 * @param doc 文档对象
 * @param code 代码内容
 * @param language 编程语言
 */

```

```

private static void highlightCode(StyledDocument doc, String code, String language)
    throws BadLocationException {
    String[] lines = code.split("\n");
    for (String line : lines) {
        // 处理注释
        Matcher commentMatcher = COMMENTS.matcher(line);
        if (commentMatcher.find()) {
            Style commentStyle = doc.addStyle(null, null);
            StyleConstants.setForeground(commentStyle, COMMENT_COLOR);
            doc.insertString(doc.getLength(), line + "\n", commentStyle);
            continue;
        }

        // 处理字符串
        Matcher stringMatcher = STRINGS.matcher(line);
        int lastEnd = 0;
        while (stringMatcher.find()) {
            // 添加字符串前的普通文本
            doc.insertString(doc.getLength(),
                line.substring(lastEnd, stringMatcher.start()),
                createBaseStyle(doc));

            // 添加字符串
            Style stringStyle = createBaseStyle(doc);
            StyleConstants.setForeground(stringStyle, STRING_COLOR);
            doc.insertString(doc.getLength(),
                stringMatcher.group(),
                stringStyle);

            lastEnd = stringMatcher.end();
        }

        // 处理关键字
        String remaining = line.substring(lastEnd);
        Matcher keywordMatcher = KEYWORDS.matcher(remaining);
        lastEnd = 0;
        while (keywordMatcher.find()) {
            // 添加普通文本
            doc.insertString(doc.getLength(),
                remaining.substring(lastEnd, keywordMatcher.start()),
                createBaseStyle(doc));

            // 添加关键字
            Style keywordStyle = doc.addStyle(null, null);

```

```

        StyleConstants.setForeground(keywordStyle, KEYWORD_COLOR);
        doc.insertString(doc.getLength(),
            keywordMatcher.group(),
            keywordStyle);

        lastEnd = keywordMatcher.end();
    }

    // 添加剩余文本
    if (lastEnd < remaining.length()) {
        doc.insertString(doc.getLength(),
            remaining.substring(lastEnd) + "\n",
            createBaseStyle(doc));
    }
}

/**
 * 创建基本代码样式
 * 设置字体、大小和颜色等基本属性
 */
private static Style createBaseStyle(StyledDocument doc) {
    Style style = doc.addStyle(null, null);
    StyleConstants.setFontFamily(style, "Consolas");
    StyleConstants.setFontSize(style, 14);
    StyleConstants.setForeground(style, NORMAL_CODE_COLOR);
    StyleConstants.setBackground(style, BACKGROUND_COLOR);
    return style;
}

/**
 * 复制文本到剪贴板
 * 提供代码块的快速复制功能
 */
private static void copyToClipboard(String text) {
    StringSelection selection = new StringSelection(text);
    Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
    clipboard.setContents(selection, null);
}
}

```

这两个类的主要功能和特点：

### 1. StreamingResponse 类：

- 实现打字机效果的流式输出
- 支持暂停/继续/跳过功能

- 可调节输出速度
- 使用 `ScheduledExecutorService` 实现定时输出
- 线程安全的实现

## 2. CodeHighlighter 类:

- 支持多种编程语言的语法高亮
- 实现代码块的格式化显示
- 提供代码复制功能
- 支持各种代码元素的颜色定制
- 使用正则表达式识别代码元素

这些实现极大地提升了用户体验:

1. 流式输出让 AI 回复更自然
2. 代码高亮让代码更易读
3. 复制功能方便用户使用代码
4. 整体界面更专业美观

## 5. 多线程与网络通信实现

### 5.1 多线程架构

```
public class JavaKnowledgeBot extends JFrame {  
    // 创建线程池管理多个用户请求  
    private final ExecutorService threadPool;  
  
    public JavaKnowledgeBot() {  
        // 初始化可缓存的线程池，根据需要创建新线程  
        threadPool = Executors.newCachedThreadPool();  
    }  
  
    // 处理用户发送消息的方法  
    private void onSendMessage(ActionEvent e) {  
        String userInput = inputTextField.getText().trim();  
        if (!userInput.isEmpty()) {  
            // 在新线程中处理用户请求  
            threadPool.submit(() -> {  
                try {  
                    // 创建 API 实例  
                    ModelAPI api = ModelAPIFactory.createAPI(  
                        vendorComboBox.getSelectedItem().toString()  
                    );  
  
                    // 异步调用 API  
                    String response = api.call(userInput);  
  
                    // 在 EDT 线程中更新 UI  
                    SwingUtilities.invokeLater(() -> {  
                        streamingResponse.streamResponse(response);  
                    });  
                } catch (Exception ex) {  
                    handleError(ex);  
                }  
            });  
        }  
    }  
}
```

### 5.2 网络请求优化

```
public class BaiduAPI implements ModelAPI {  
    private final OkHttpClient httpClient;
```

```

public BaiduAPI() {
    // 配置OkHttpClient, 设置超时和连接池
    this.httpClient = new OkHttpClient.Builder()
        .connectTimeout(30, TimeUnit.SECONDS)
        .writeTimeout(30, TimeUnit.SECONDS)
        .readTimeout(60, TimeUnit.SECONDS)
        // 配置连接池, 最大空闲连接5个, 保持活跃5分钟
        .connectionPool(new ConnectionPool(5, 5, TimeUnit.MINUTES))
        .build();
}

@Override
public String call(String userInput) throws Exception {
    // 构建请求
    Request request = new Request.Builder()
        .url(getApiUrl())
        .post(createRequestBody(userInput))
        .build();

    // 执行网络请求
    try (Response response = httpClient.newCall(request).execute()) {
        if (!response.isSuccessful()) {
            throw new IOException("API 调用失败: " + response.code());
        }
        return response.body().string();
    }
}
}

```

### 5.3 性能监控实现

```

```java
public class PerformanceMonitor {
    // 使用线程安全的Map 存储请求时间
    private final Map<String, Long> requestTimes = new ConcurrentHashMap<>();

    // 开始监控请求
    public void startRequest(String requestId) {
        requestTimes.put(requestId, System.currentTimeMillis());
    }

    // 结束监控请求
    public void endRequest(String requestId) {
        Long startTime = requestTimes.remove(requestId);
    }
}

```

```
        if (startTime != null) {  
            long duration = System.currentTimeMillis() - startTime;  
            logger.info("请求 {} 耗时: {}ms", requestId, duration);  
        }  
    }  
}
```

## 5.4 实现特点

### 1. 线程池管理

- 使用 `ExecutorService` 管理线程池
- 自动创建和复用线程
- 避免频繁创建和销毁线程
- 控制并发线程数量

### 2. 异步处理

- 用户请求在独立线程中处理
- 不阻塞 UI 主线程
- 使用 `SwingUtilities.invokeLater` 安全更新 UI
- 支持并行处理多个请求

### 3. 网络优化

- 使用 `OkHttpClient` 的连接池
- 设置合理的超时时间
- 复用 HTTP 连接
- 自动管理连接资源

### 4. 性能监控

- 使用 `ConcurrentHashMap` 保证线程安全
- 记录每个请求的处理时间
- 提供性能分析数据
- 支持请求追踪

## 6. 日志记录实现

### 6.1 日志记录功能概述

实现了完整的对话日志记录功能，主要特点：

- 按不同 AI 模型分别存储对话记录
- 使用本地文本文件(.txt)持久化存储
- 支持 UTF-8 编码，确保中文正确显示
- 自动创建和管理日志文件
- 实现了读写分离，提高性能

### 6.2 核心代码实现

#### 1) 保存对话记录

```
/**
 * 保存单条对话记录到文件
 * 使用 synchronized 确保线程安全
 * @param message 要保存的消息内容
 */
private synchronized void saveChatRecord(String message) {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(getLogFilePath(), true))) {
        writer.write(message);
        writer.newLine();
    } catch (IOException e) {
        e.printStackTrace();
        CodeHighlighter.insertText(chatTextPane, "Error: 保存聊天记录失败: " + e.getMessage() + "\n", false);
    }
}
```

#### 2) 加载历史记录

```
/**
 * 加载历史对话记录
 * 如果日志文件不存在会自动创建
 */
private void loadChatRecord() {
    String logFilePath = getLogFilePath();
    File logFile = new File(logFilePath);

    // 如果文件不存在，创建新文件
    if (!logFile.exists()) {
        try {
            logFile.createNewFile();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return; // 新文件创建后直接返回，因为内容为空
    }
}
```



```

    } catch (IOException e) {
        handleError(new Exception("创建聊天记录文件失败: " + e.getMessage(), e));
        return;
    }
}

// 读取现有文件内容
try (BufferedReader reader = new BufferedReader(new FileReader(logFile))) {
    String line;
    while ((line = reader.readLine()) != null) {
        chatTextPane.setText(chatTextPane.getText() + line + "\n");
    }
} catch (IOException e) {
    handleError(new Exception("加载聊天记录失败: " + e.getMessage(), e));
}
}

```

### 3) 日志文件路径管理

```

/**
 * 获取当前 AI 模型对应的日志文件路径
 * 根据选择的厂商自动生成对应的文件名
 * @return 日志文件路径
 */
private String getLogFilePath() {
    // 获取选中的厂商名称
    String selectedVendor = (String) vendorComboBox.getSelectedItem();

    // 使用 UTF-8 编码处理文件名
    try {
        String fileName = "chat_log_" + selectedVendor.replace(" ", "_") + ".txt";
        return new String(fileName.getBytes("UTF-8"), "UTF-8");
    } catch (Exception e) {
        e.printStackTrace();
        return "chat_log_default.txt";
    }
}

```

## 6.3 使用场景

### 1. 记录用户输入

```

// 发送消息时记录用户输入
chatTextArea.append("User: " + userInput + "\n");
saveChatRecord("User: " + userInput);

```

## 2. 记录 AI 响应

```
// 接收到 AI 响应时记录
chatTextArea.append("Bot: " + modelResponse + "\n");
saveChatRecord("Bot: " + modelResponse);
````
```

## 3. 切换模型时的记录处理

```
// 切换 AI 模型时重新加载对应的聊天记录
private void onVendorChange(ActionEvent e) {
    chatTextPane.setText(""); // 清空当前聊天记录
    loadChatRecord(); // 加载新选厂商的聊天记录
}
```

## 6.4 实现特点

### 1. 文件管理

- 按 AI 模型分别存储记录
- 自动创建日志文件
- UTF-8 编码支持
- 文件名安全处理

### 2. 读写操作

- 追加模式写入
- 按行读取记录
- 使用缓冲流提高性能
- 自动关闭资源

### 3. 异常处理

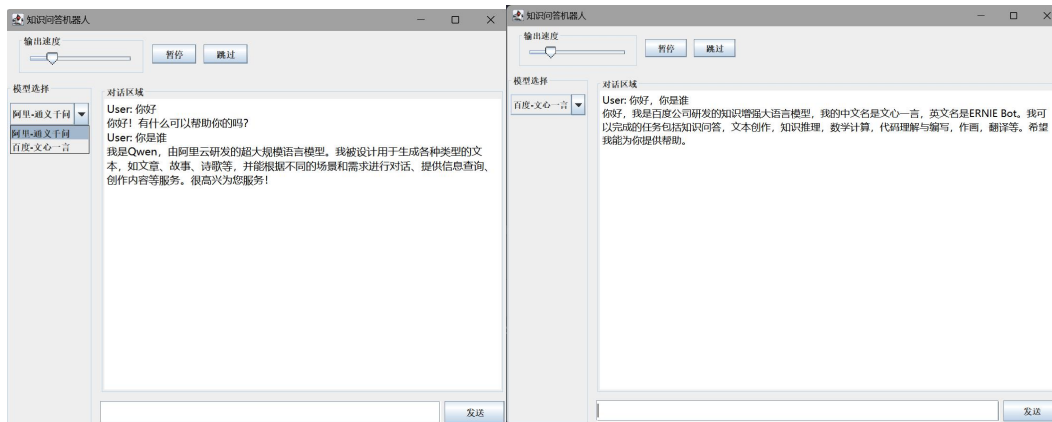
- IO 异常优雅处理
- 编码异常处理
- 提供错误反馈
- 默认值保护

### 4. 性能考虑

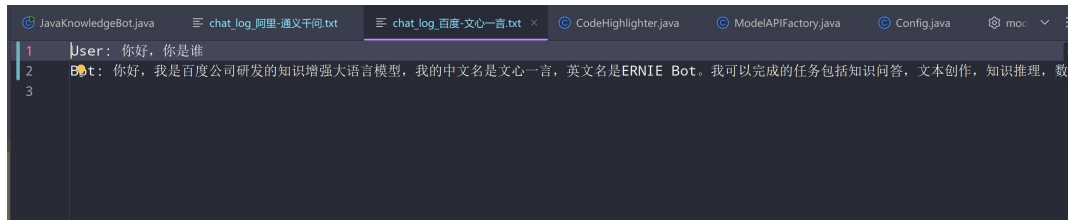
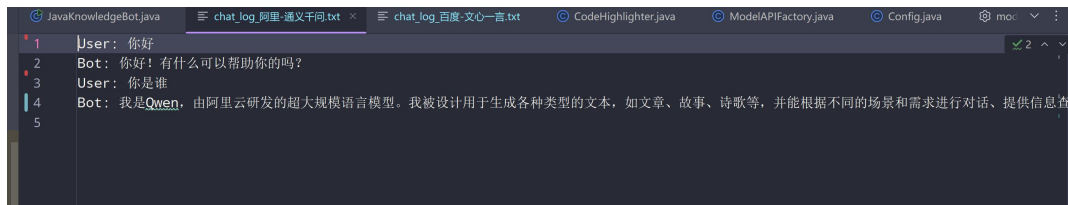
- 使用 `BufferedWriter` 提升写入性能
- 使用 `BufferedReader` 提升读取性能
- 资源自动关闭
- 最小化文件操作

测试效果分析：

## 1. 主页面图形界面：



## 2. 日志记录



3.测试案例：要求测试 Java 程序设计方面的选择题、判断题、程序填空题和编程题（每个题型至少测试两个例子），并就测试效果做出简单分析

### 选择题1

Q: Java 中，下列哪个不是基本数据类型？

- A. `int`
- B. `String`
- C. `double`
- D. `boolean`

输出速度

暂停

跳过

模型选择

百度-文心一言

对话区域

User: 你好, 你是谁  
Bot: 你好, 我是百度公司研发的知识增强大语言模型, 我的中文名是文心一言, 英文名是ERNIE Bot。我可以完成的任务包括知识问答, 文本创作, 知识推理, 数学计算, 代码理解与编写, 作画, 翻译等。希望我能为你提供帮助。  
User: Q: Java中, 下列哪个不是基本数据类型? A. int B. String C. double D. boolean  
在Java中, 下列哪个不是基本数据类型的问题, 答案是:  
  
B. String  
  
解释:  
- `int` 是基本数据类型, 表示一个整数。  
- `double` 是基本数据类型, 表示一个双精度浮点数。  
- `boolean` 是基本数据类型, 表示一个布尔值 (true 或 false) 。  
- `String` 不是基本数据类型, 它是一个类 (class) , 用于表示字符串。  
  
Java中的基本数据类型包括:  
- 整数类型: `byte`, `short`, `int`, `long`  
- 浮点类型: `float`, `double`  
- 字符类型: `char`  
- 布尔类型: `boolean`

发送

答案正确

## 选择题 2

Q: 关于 Java 接口, 下列说法正确的是?

- A. 接口可以包含普通方法
- B. 接口中的变量默认是 `private` 的
- C. 一个类可以实现多个接口
- D. 接口可以继承多个类

输出速度

暂停

跳过

模型选择

阿里-通义千问

对话区域

- 这是不正确的。在Java中, 接口不能继承类, 它们只能继承其他接口。这是通过`extends`关键字实现的, 允许接口继承另一个或多个接口。  
  
综上所述, 最准确且无误的选项是C: 一个类可以实现多个接口。这是Java接口设计的一个核心特性, 允许类实现多个接口以获取多种行为。  
  
因此, 正确答案是C。  
User: // 选择题2 Q: 关于Java接口, 下列说法正确的是? A. 接口可以包含普通方法 B. 接口中的变量默认是private的 C. 一个类可以实现多个接口 D. 接口可以继承多个类  
关于Java接口, 正确的选项是: C. 一个类可以实现多个接口。  
  
解析:  
- A. 接口中不能包含普通方法 (即非抽象方法), 直到Java 8引入了默认方法 (default methods) 和静态方法 (static methods) 。在此之前, 接口中只能有抽象方法。  
- B. 接口中的变量默认是`public static final`, 即它们是公共的、静态的和不可改变的。因此说接口中的变量默认是`private`的说法是错误的。  
- C. 这个选项是正确的。在Java中, 一个类可以通过使用`implements`关键字来实现多个接口。  
- D. 接口不能继承类, 但可以继承其他接口。如果需要从类继承, 应该使用类之间的继承关系, 即通过`extends`关键字。  
  
因此, 正确答案是C。

发送

答案正确

## 判断题 1

Q: Java 是一种解释型语言。 ( )

输出速度

暂停

跳过

模型选择

阿里-通义千问

对话区域

默认是private的 C. 一个类可以实现多个接口 D. 接口可以继承多个类

Bot: 关于Java接口, 正确的选项是: C. 一个类可以实现多个接口。

解析:

- A. 接口中不能包含普通方法(即非抽象方法), 直到Java 8引入了默认方法 (default methods) 和静态方法 (static methods)。在此之前, 接口中只能有抽象方法。
- B. 接口中的变量默认是 'public static final', 即它们是公共的、静态的和不可改变的。因此说接口中的变量默认是 'private' 的说法是错误的。
- C. 这个选项是正确的。在Java中, 一个类可以通过使用 'implements' 关键字来实现多个接口。
- D. 接口不能继承类, 但可以继承其他接口。如果需要从类继承, 应该使用类之间的继承关系, 即通过 'extends' 关键字。

因此, 正确答案是C。

User: // 判断题1 Q: Java是一种解释型语言。 ( )

Bot: 判断题1 答案: 错误。

Java 语言本身可以被编译成字节码 (.class 文件), 这种字节码可以在任何安装了Java虚拟机 (JVM) 的平台上运行。在这个意义上, Java 是一种“编译型”和“解释型”结合的语言。字节码由JVM解释执行或通过即时编译器 (Just-In-Time Compiler, JIT) 编译成机器码后执行。因此, 说Java是完全的解释型语言是不准确的。正确的表述应该是Java是一种先编译后解释 (或部分编译) 执行的语言。

发送

答案正确

## 判断题 2

Q: Java 中, 一个类可以同时继承多个父类。 ( )

输出速度

暂停

跳过

模型选择

阿里-通义千问

对话区域

解析:

- A. 接口中不能包含普通方法(即非抽象方法), 直到Java 8引入了默认方法 (default methods) 和静态方法 (static methods)。在此之前, 接口中只能有抽象方法。
- B. 接口中的变量默认是 'public static final', 即它们是公共的、静态的和不可改变的。因此说接口中的变量默认是 'private' 的说法是错误的。
- C. 这个选项是正确的。在Java中, 一个类可以通过使用 'implements' 关键字来实现多个接口。
- D. 接口不能继承类, 但可以继承其他接口。如果需要从类继承, 应该使用类之间的继承关系, 即通过 'extends' 关键字。

因此, 正确答案是C。

User: // 判断题1 Q: Java是一种解释型语言。 ( )

Bot: 判断题1 答案: 错误。

Java 语言本身可以被编译成字节码 (.class 文件), 这种字节码可以在任何安装了Java虚拟机 (JVM) 的平台上运行。在这个意义上, Java 是一种“编译型”和“解释型”结合的语言。字节码由JVM解释执行或通过即时编译器 (Just-In-Time Compiler, JIT) 编译成机器码后执行。因此, 说Java是完全的解释型语言是不准确的。正确的表述应该是Java是一种先编译后解释 (或部分编译) 执行的语言。

User: // 判断题2 Q: Java中, 一个类可以同时继承多个父类。 ( )

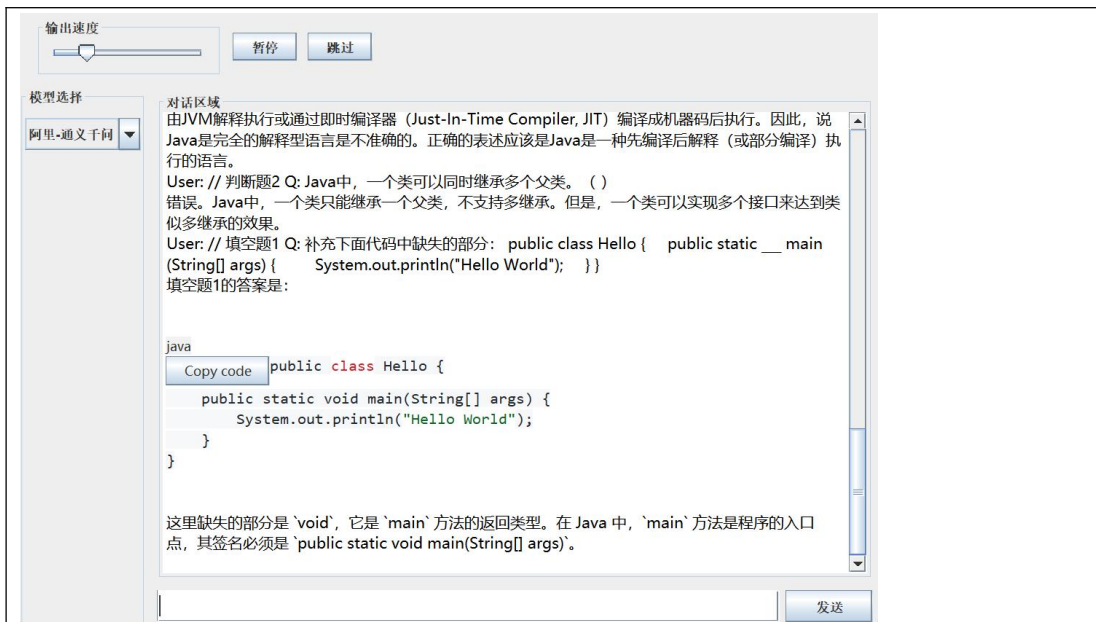
错误。Java中, 一个类只能继承一个父类, 不支持多继承。但是, 一个类可以实现多个接口来达到类似多继承的效果。

发送

## 填空题 1

Q: 补充下面代码中缺失的部分:

```
public class Hello {  
    public static ____ main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```

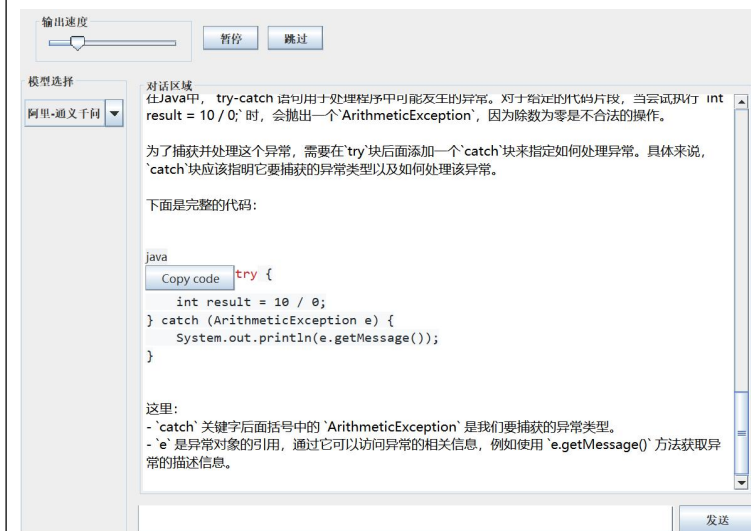


答案正确

填空题2

Q: 补充 try-catch 语句中缺失的部分：

```
try {
    int result = 10 / 0;
} ____ (____ e) {
    System.out.println(e.getMessage());
}
```

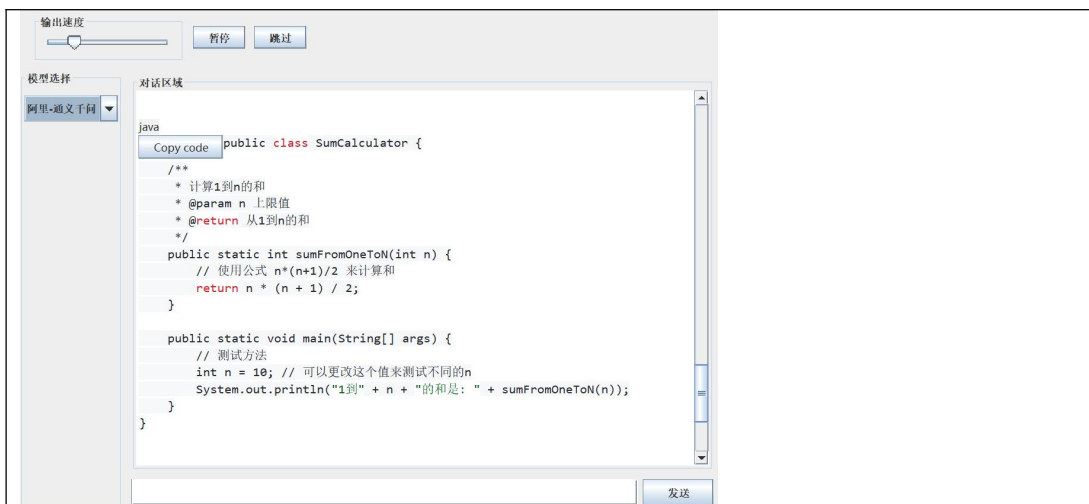


答案正确

// 编程题1

Q: 编写一个方法，计算 1 到 n 的和。

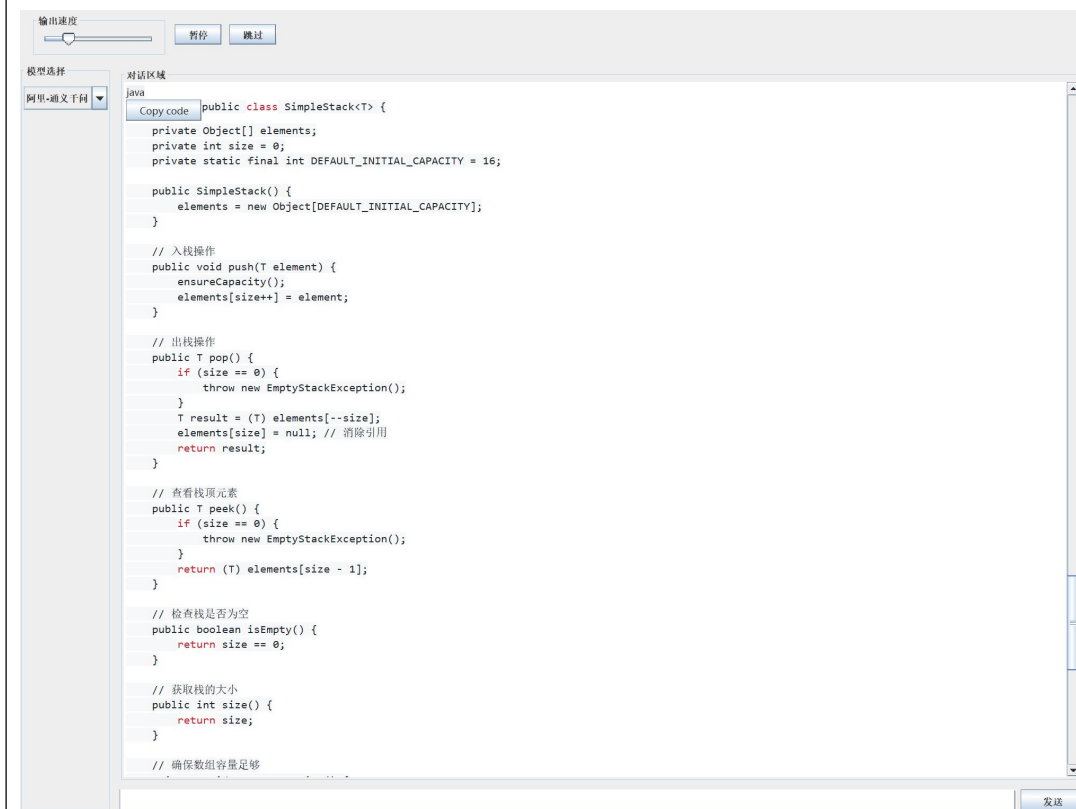
答案：



答案正确

## 编程题 2

Q: 实现一个简单的栈数据结构。



答案正确

## 编程题分析

计算 1 到 n 的和:

- 使用了数学公式  $n*(n+1)/2$

- 优点：
- 时间复杂度  $O(1)$
- 代码简洁高效
- 避免了循环累加的开销

栈的实现：

- 使用泛型数组实现

主要功能：

- **push**: 入栈
- **pop**: 出栈
- **peek**: 查看栈顶
- **isEmpty**: 检查空栈
- **size**: 获取大小
- 特点：
- 支持动态扩容
- 类型安全
- 异常处理完善
- 内存管理合理（清除无用引用）

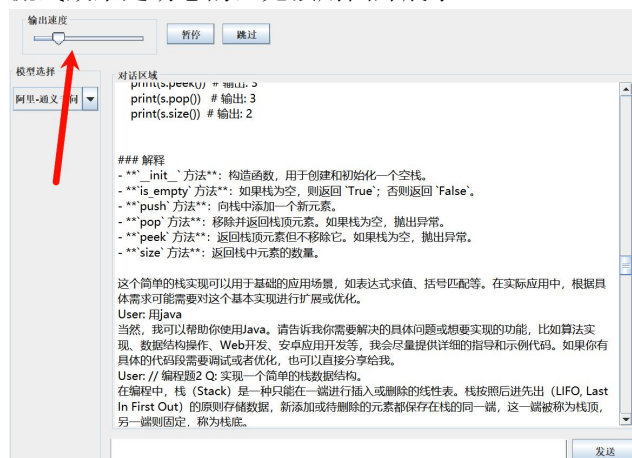
这些题目涵盖了 Java 的多个重要概念：

- 面向对象特性（继承、接口）
- 语言本质特性（编译/解释）
- 基本语法（**main** 方法）
- 异常处理机制
- 算法实现
- 数据结构设计

### 3. 特色功能 具体实现已在前面说明

流式输出：可以控制输出速度，暂停输出以及直接跳过展现全部答案。

流式效果是动态的，无法用图片展示





代码高亮:



代码背景是暗色块提示视觉效果

Copy code 按钮一键复制代码

这些实现极大地提升了用户体验:

1. 流式输出让 AI 回复更自然
2. 代码高亮让代码更易读
3. 复制功能方便用户使用代码
4. 整体界面更专业美观

## 未来优化方向

1. 添加更多 AI 模型的支持
2. 优化用户界面，提供更好的用户体验
3. 添加对话上下文管理
4. 增加错误重试机制
5. 添加消息队列支持
6. 优化网络请求性能
7. 添加更多的配置选项

+++++

其他（例如感想、建议等等）。

指导教师批阅意见：

成绩评定：

指导教师签字：

2024 年    月    日

备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生课程作业报告时间应在学生提交课程作业报告时间后 10 日内。