**二次开发**

**详细设计报告**

学院：计算机学院

小组：有请下一组

项目名称：Code-Wizard

基于OpenHands的工程级代码智能助手二次开发

成员：冯敏恒 张呦呦 薄冬 丁好

指导老师： 金福生

北京理工大学2024年软件体系结构与设计模式课程项目

2024年秋季学期

**目 录**

[1. 引言 3](#_Toc17906)

[1.1 目的 3](#_Toc5034)

[1.2 背景 3](#_Toc25802)

[2. 项目背景和目标 3](#_Toc26564)

[2.1 项目背景 3](#_Toc19513)

[2.2 二次开发目标 4](#_Toc9566)

[3. 需求分析 4](#_Toc23122)

[3.1 功能性需求 4](#_Toc15997)

[3.2 非功能性需求 5](#_Toc31832)

[3.3 需求优先级 5](#_Toc26976)

[4. 系统架构 7](#_Toc15731)

[4.1 现有系统的关键组件与核心机制（OpenHands） 7](#_Toc743)

[4.2 架构及变更（Code-Wizard） 8](#_Toc29678)

[5. 二次开发功能设计分析 11](#_Toc2400)

[5.1 知识库、代码库的集成 11](#_Toc18793)

[5.2 支持快捷调用VSCode外部工具 15](#_Toc6755)

[5.3 代码注释生成 18](#_Toc32170)

[5.4 代码优化建议 20](#_Toc29924)

[5.5 代码内联补全 22](#_Toc246)

[5.6 有效的Prompt模板预设与自定义 24](#_Toc18611)

[5.7 代码翻译转换 27](#_Toc10766)

[5.8 项目级代码生成 30](#_Toc9973)

[5.9 潜在问题漏洞修复 33](#_Toc11596)

[6. 部分测试用例示例 35](#_Toc17239)

[7. 二次开发进展情况 37](#_Toc25971)

[7.1 时间线 37](#_Toc29719)

[7.2 关键里程碑 38](#_Toc17453)

[8. 项目需求实现总结表 39](#_Toc3862)

**1. 引言**

**1.1 目的**

本设计报告旨在详细说明基于OpenHands平台的工程级代码智能助手的二次开发工作。 其主要目的是为项目团队提供明确的开发指导，确保开发工作的顺利进行，并为未来的维护和升级工作提供参考。目标读者包括项目开发人员、测试人员、项目经理以及可能使用本系统的技术人员。

**1.2 背景**

随着软件开发行业的快速发展，自动化编程工具的需求日益增长。尽管已有一些工具提供了基本的代码生成和辅助功能，但市场仍需要一个更智能、更全面的解决方案，以支持软件开发的全生命周期。现有系统OpenHands虽然提供了一些基础功能，但在代码生成、理解和交互方面仍有待提升。因此，我们决定进行二次开发，以增强系统的功能和性能，满足市场和用户的需求。

**2. 项目背景和目标**

**2.1 项目背景**

现有系统OpenHands是一个开源的代码生成平台，它提供了基础的代码生成和理解功能。然而，该系统在处理大规模代码库、提供深度文档分析和支持复杂代码生成方面存在局限性。此外，系统在用户体验和交互设计方面也有改进空间。为了提升系统的竞争力，我们需要对现有系统进行二次开发，增加新的功能，优化现有性能，并改善用户界面。

**2.2 二次开发目标**

二次开发的目标是在现有OpenHands平台的基础上，通过集成先进的生成式模型、工具学习和沙盒环境，构建一个全面的工程级代码智能助手。该助手将支持代码生成、代码补全、代码审查等多个功能模块，并提供高效的检索增强生成、工具调用和知识库集成能力。预期成果包括提升系统的代码生成质量、增强系统的交互能力和用户体验，以及提高系统在处理大规模代码库时的性能。

**3. 需求分析**

**3.1 功能性需求**

代码整库分析：系统应能深入理解仓库级的代码，并支持项目文件级的代码编写与生成。

文档分析增强：系统应融合文档知识库与知识图谱，通过检索和推理增强，为文档分析提供深层次支持。

代码定制化交互式生成：系统应支持基于多轮对话的代码生成和优化，满足用户特定的代码需求。

工具和知识库集成：系统应支持与多种开发工具和知识库的集成，提供代码补全、建议、注释生成等智能化功能。

私有化部署和模型支持：系统应支持依托开源大模型的私有化部署，并能接入各类开源模型与API。

**3.2 非功能性需求**

性能需求：系统应具备高推理速度，支持32K以上的上下文处理能力，并能高效处理大型代码仓库。

安全性需求：系统应确保代码和数据的安全性，防止外部数据泄露，并具有良好的防护机制，检测和阻止异常的代码请求。

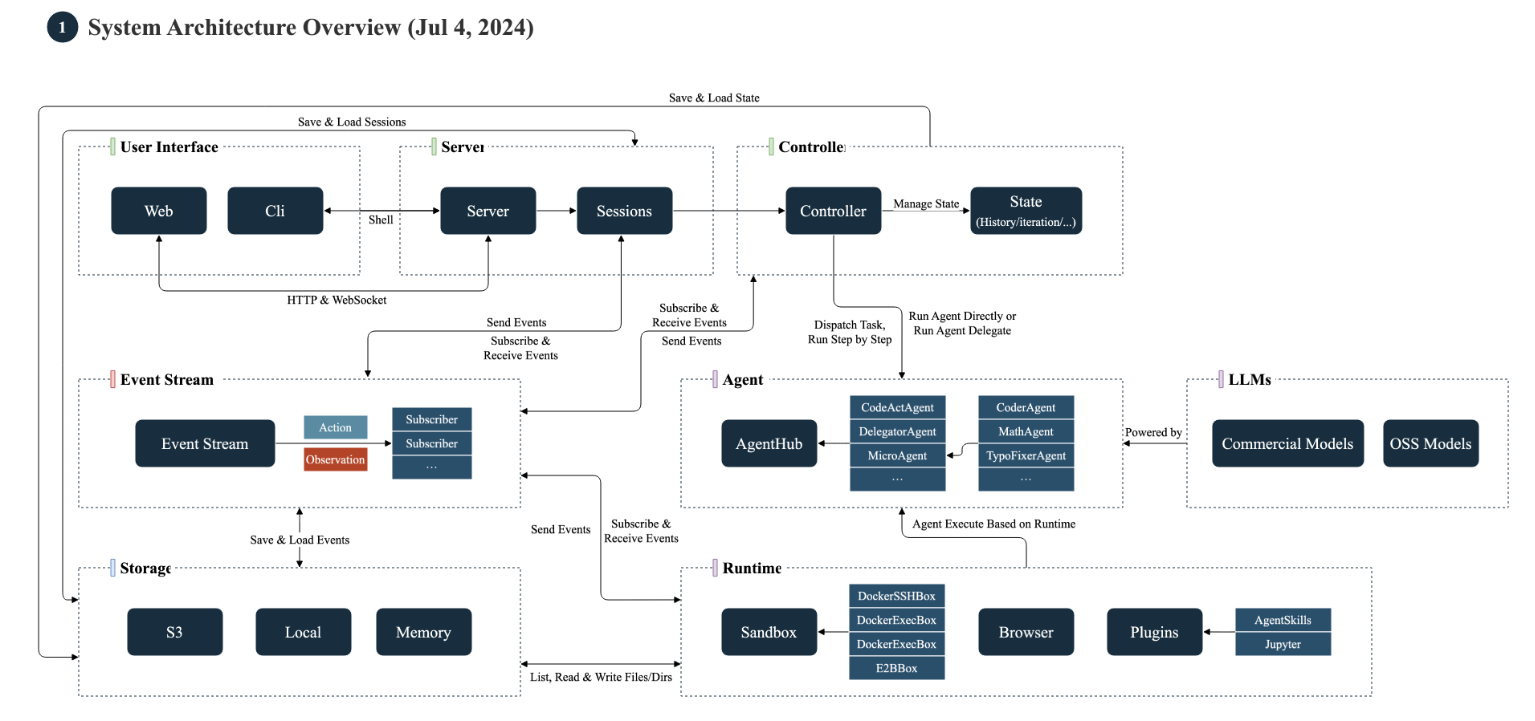
兼容性需求：系统需支持多种编程语言的代码生成与优化，并兼容主流开发平台和开发环境。

**3.3 需求优先级**

| 序号 | 需求描述 | 优先级 | 备注 |
| --- | --- | --- | --- |
| 1 | 代码整库分析 | 高 | 实现仓库级的代码深入理解和项目文件级的代码编写与生成 |
| 2 | 文档分析增强 | 高 | 融合文档知识库与知识图谱，提供深层次文档分析支持 |
| 3 | 项目级代码生成 | 高 | 支持生成项目工程级代码 |
| 4 | 代码定制化交互式生成 | 高 | 支持多轮对话，总结用户需求，生成优化代码 |
| 5 | 工具和知识库集成 | 中 | 集成多种开发工具和知识库，提供智能化代码辅助功能 |
| 6 | 私有化部署和模型支持 | 中 | 支持开源大模型的私有化部署，兼容高效推理框架 |
| 7 | 统一API调用和模型API接入扩充能力 | 中 | 提供API接口，支持模型及API的接入和扩充 |
| 8 | 代码补全、建议、注释生成 | 中 | 提供代码编写辅助功能 |
| 9 | 代码转换及智能问答交互功能 | 中 | 支持代码的转换和提供智能问答交互 |
| 10 | 系统性能优化 | 低 | 优化系统响应速度和处理能力 |
| 11 | 用户界面和交互体验改善 | 低 | 提升系统的易用性和用户体验 |

**4. 系统架构**

**4.1 现有系统的关键组件与核心机制（OpenHands）**



OpenHands 是基于 CodeAct 1.0 的智能助手，它集成了大语言模型（LLM）代理的行为，提供了一个统一的代码行动空间。这个架构允许 OpenHands 在编程任务中扮演全方位的智能助手角色，包括自然语言交流、代码执行、文件编辑等功能。

在每次交互中，OpenHands 可以执行自然语言对话（与用户进行自然语言交流，获取或澄清信息）、代码行动（CodeAct）：执行代码、运行 Linux 命令、编辑文件等编程任务。

其在 SWE-Agent 的基础上进行了优化，增加部分实用工具，包括：在指定路径创建文件、浏览和编辑文件内容、在有限的交互次数内完成任务、简化命令书写和操作解析过程等。

其集成了多种工具和框架，使用的关键技术包括但不限于：FastAPI，用于快速构建和部署 API 的工具；uvicorn Lite，用于异步处理；LLM，大语言模型，用于自然语言处理和代码生成；Docker，容器化技术，用于隔离和部署应用；Ruff，用于处理 HTTP 请求的 Python 库；lama，用于构建和训练机器学习模型的库；React，前端 JavaScript 库，用于构建用户界面等等。

基于二次开发项目需求，小组在充分调研OpenHands系统后发现其已经实现以下功能：

支持依托于开源的 LLM 与 Embedding 模型，实现基于开源模型的离线私有部署，可兼容vllm等高效推理框架。

支持统一的API调用，并具备对各类开源大模型及模型 API 的接入扩充能力，具备不少于32K上下文支持。

支持登录GitHub账户并连接所持有的代码仓库进行初步整库分析与操作。

支持代码智能交互问答。

**4.2 架构及变更（Code-Wizard）**

基于二次开发项目需求，本组在OpenHands的基础上修复多个潜在漏洞，新增多项功能，包括但不限于：

·本地部署阶段在make build与make run环节的问题修复

·历史对话内容持久化存储失败问题修复

·代码编辑区组件文件保存的系统不兼容问题修复

……

·支持集成工具库、代码库及知识库，连接AnythingLLM Api进行管理

·支持在代码编辑区调用快捷键实现代码补全、代码建议、注释生成、代码翻译转换等功能

·支持对话模式选择，选择使用代理进行问答交互或连接自定义知识库以实现更深入的代码库级别理解与增强分析

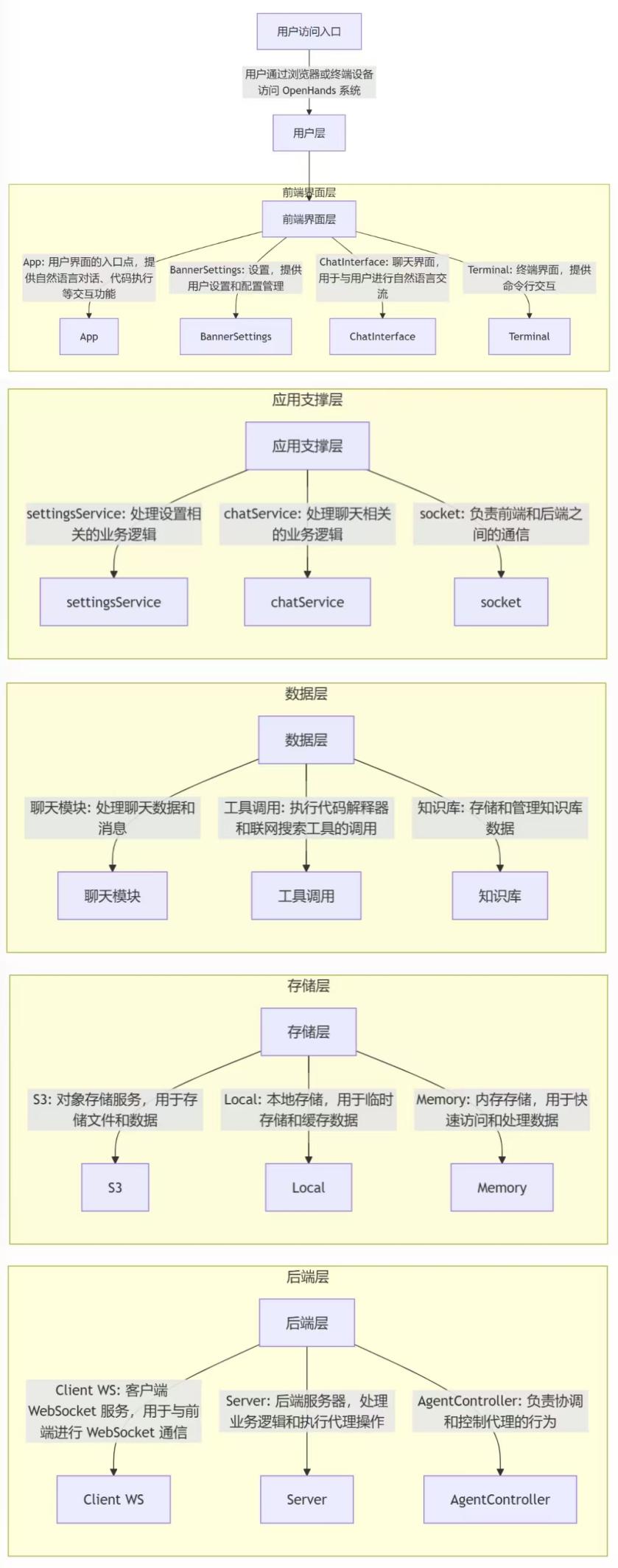
·支持工具调用能力（快捷启动VSCode协同开发、基于调用大模型的代码解释、翻译、补全等的实现等）

·支持项目级代码生成与优化

·支持有效的prompt模板预设与自定义

……

以下为Code-Wizard的系统架构分层图：



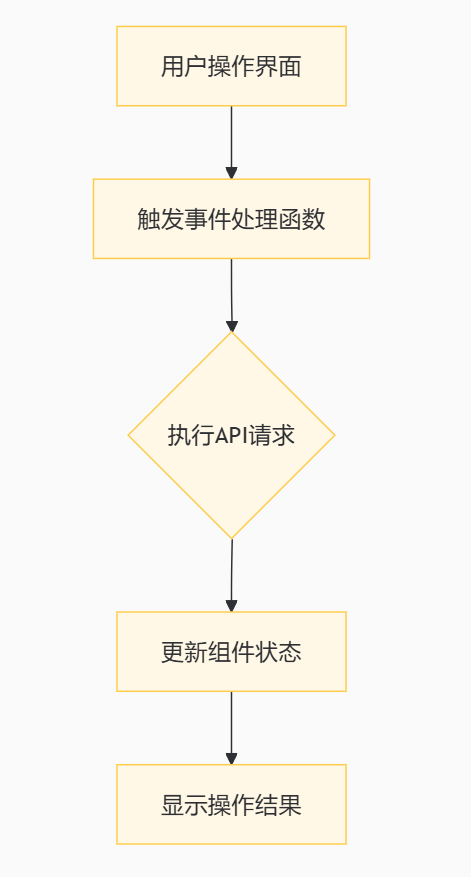
1. **二次开发功能设计分析**
   1. **知识库、代码库的集成**

**在侧边栏新增对话模式切换与知识库管理的前端界面，调用AnythingLLM部分接口以实现此功能，整体设计与实现思路如下：**

首先，使用本功能的前置条件是须在本机安装AnythingLLM的桌面端应用程序，注册登录并获得用户Token，同时须在AnythingLLM中配置语言模型、向量模型等（可选本地Ollama或在线大模型，但必须保证成功连通）。

由于OpenHands原来的前端运行在3001端口，与AnythingLLM端口发生冲突，故最终将OpenHands前端运行的端口改为3002，避免在api调用时发生冲突。

**以下流程图描述了用户在OpenHands系统实现对AnythingLLM后端的连接，进而实现知识库与代码知识库管理、问答等各项功能的本质，即通过的API接口，以编程方式访问和管理嵌入知识库和代码库的模型实例。**

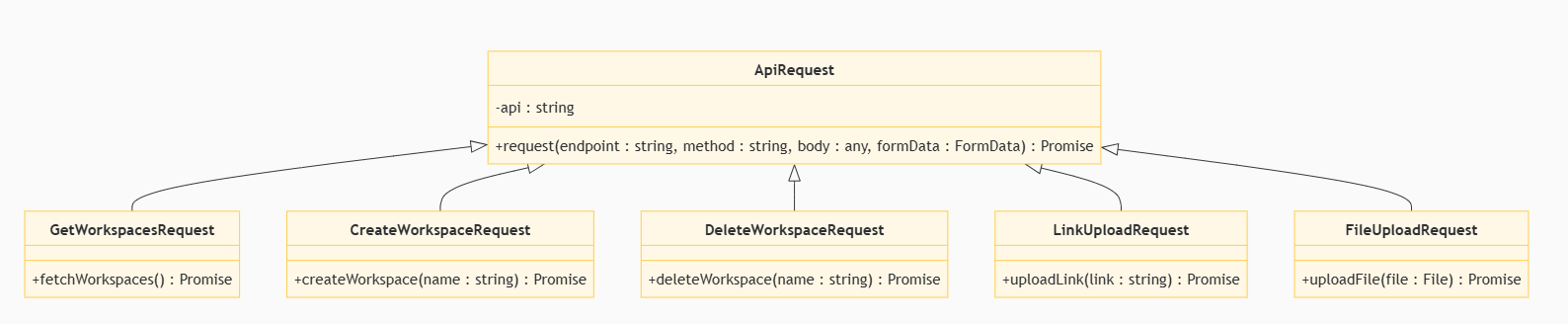


**以下为集成知识库多项功能时所用到的后端api接口：**



**本功能设计体现的设计模式包括：**

1. 工厂模式



·工厂类（ApiRequest）：

ApiRequest类是一个抽象的工厂类，它定义了执行API请求的模板方法request。这个方法封装了发起HTTP请求的通用逻辑，包括设置请求头、请求体和处理响应。

·具体产品类（子类）：

GetWorkspacesRequest、CreateWorkspaceRequest、DeleteWorkspaceRequest、LinkUploadRequest和FileUploadRequest是具体的产品类，它们继承自ApiRequest工厂类。这些子类实现了特定的API请求方法，如fetchWorkspaces、createWorkspace、deleteWorkspace、uploadLink和uploadFile，它们是不同的API操作，包括创建工作区、删除工作区、上传文档、向知识库提问等。

·客户端代码：

在DialogModeForm组件中，客户端代码通过创建ApiRequest子类的实例来执行特定的API请求。比如当需要获取工作区列表时，客户端代码实例化GetWorkspacesRequest对象，并调用其fetchWorkspaces方法。这种设计使得客户端代码不需要知道具体的API请求细节，只需要知道需要执行的操作类型。

因此，在此处运用工厂模式的优势在于能够将对象的创建逻辑封装在工厂类中，使得客户端代码不需要知道具体的创建细节，进而提高扩展性，在新增API请求类型时，只需添加新的子类，无需修改现有代码，提高了系统的扩展性。同时，客户端代码只需要与工厂类交互，不需要直接依赖具体的产品类，降低了客户端代码和产品类之间的耦合度。

1. MVC模式

在集成知识库的过程中，我们通过将API请求逻辑（Model）、用户界面（View）和用户交互（Controller）分离，实现了MVC模式。

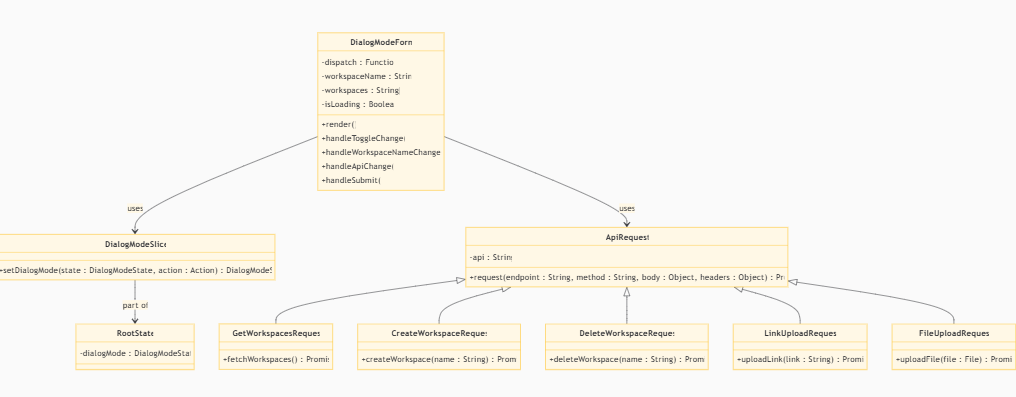
·模型由状态管理和API请求类组成。

DialogModeState定义了应用的状态，包括是否启用对话框模式、工作区名称、模式和API URL。API请求类（如ApiRequest、GetWorkspacesRequest、CreateWorkspaceRequest等）负责与后端进行通信，执行具体的业务逻辑，如获取工作区列表、创建工作区、删除工作区、上传文件和链接等。

DialogModeForm组件是视图部分，负责展示用户界面和接收用户输入。它显示了对话框模式的配置选项，如开关、工作区名称选择器、模式选择器和API URL输入框。

而控制器逻辑分布在DialogModeForm组件的事件处理函数中。

事件处理函数（如handleToggleChange、handleModeChange、handleWorkspaceNameChange等）响应用户的交互操作，如切换开关状态、选择工作区、更改模式等。同时，控制器还负责调用模型中的API请求方法，并将结果显示在视图中，如更新工作区列表、显示操作成功的提示。



* 1. **支持快捷调用VSCode外部工具**

**在侧边栏新增功能按钮以快捷调用VSCode工具实现下同开发，整体设计与实现思路如下：**

利用useState钩子来记忆vscodePath和isVscodeModalOpen状态，确保在组件重新渲染时，用户输入的路径和模态框状态不会被重置；通过定义handleVscodeButtonClick事件处理函数来打开模态框，让用户输入项目路径。

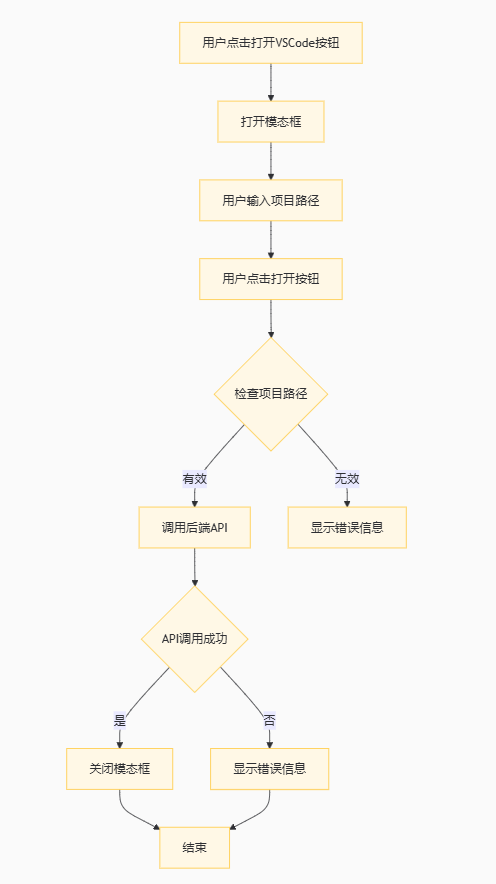
通过handleVscodeInputChange事件处理函数来更新用户输入的项目路径状态；使用handleVscodeOpen事件处理函数来调用后端API，传递用户输入的项目路径，并处理API调用的结果。

实现VSCode调用逻辑 在用户输入项目路径并点击打开按钮后，通过handleVscodeOpen函数调用后端API，传递项目路径给后端服务，后端服务负责打开VSCode。在这里使用async/await模式来处理异步API调用，确保在API调用完成前，用户界面能够正确显示加载状态。

在API调用成功后，使用setIsVscodeModalOpen函数来关闭模态框，并给用户反馈操作成功的消息。使用toast组件来显示操作成功或失败的提示信息，提高用户体验。

通过上述设计思路，支持VSCode快捷调用功能能够在用户需要时快速提供操作界面，允许用户直接从Web应用中打开VSCode项目，极大地提高了开发效率和用户体验。

**以下流程图描述了用户在使用此功能的过程。**



**本功能设计体现的设计思想包括：**

1. 前后端分离：

这样的好处是使得职责清晰：前端负责用户界面和用户体验，后端负责业务逻辑和数据管理，职责划分清晰；易于维护和扩展：前后端分离使得系统各部分可以独立开发和维护，便于扩展和重构；增强可测试性，使得测试可以更加专注和独立，前端可以进行单元测试和集成测试，后端可以进行API测试。

* 1. **代码注释生成**

**在原有的Monaco Editor组件中增添部分代码以实现此功能，整体设计与实现思路如下：**

原组件通过useCallback钩子定义了一个handleEditorDidMount回调函数，这个函数在编辑器挂载完成后被调用，用于设置编辑器的主题和注册代码注释生成的事件监听器。

在编辑器配置中，使用editor.addCommand注册了一个快捷键命令，设定为Shift+Alt+P，通过editor.addCommand将快捷键与注释生成函数绑定，使得用户可以通过键盘快捷操作来生成注释。

在用户触发快捷键后，获取当前编辑器选中的代码文本，并按\n分割成多行,然后对每一行代码使用AI模型来生成对应的注释。此处通过editor.getSelection和editor.getModel().getValueInRange获取用户选中的代码文本。

使用fetchAICompletion函数异步获取AI模型返回的注释建议，通过Promise.all并行处理每一行代码的注释生成，提高整体注释生成的效率。

生成注释后，将注释与原代码合并，并构造一个新的代码字符串，其中包含原始代码和新增的注释。使用editor.executeEdits方法将新的代码字符串替换编辑器中的选中内容，实现注释的实时展示。

此外，为了防止用户在输入时频繁触发注释生成请求，使用setTimeout实现了防抖逻辑，通过全局变量debounceTimeout来跟踪是否有正在进行的注释生成请求，如果用户在防抖时间间隔内再次触发快捷键，之前的请求会被取消，只有最后一次请求会实际发送给AI。

**以下流程图描述了用户在选中一段代码并按下Alt+Shift+P快捷键时，如何触发代码注释功能，并替换更新代码注释版结果给用户的过程。**



**本功能设计体现的设计模式包括：**

A.命令模式：

通过 editor.addCommand 添加快捷键命令，体现了命令模式，它将操作封装成对象，使得可以用不同的方式请求操作，在此处用户作为命令的发出者，可通过快捷键触发。

从editor.addCommand(keybinding, handler, context)本身的方法签名来看，它包括keybinding（指定触发该命令的快捷键或其他输入事件）、handler（命令的处理函数，当命令被触发时执行的逻辑）和context（命令的上下文，用于指定特定条件下是否执行命令）。而这其中，命令的行为被封装在 handler 函数中，通过 editor.addCommand，命令的行为可以在不同的上下文中执行而不需要直接修改主程序逻辑。故用户可以根据特定的输入条件绑定不同的 handler 函数，而无需关心命令是如何被触发的。

因此在设计代码编辑区代码注释生成、代码优化建议等功能时，都采用了上述命令模式的思想,命令的行为（handler）和命令的触发条件（keybinding）被解耦，修改时更容易维护。

* 1. **代码优化建议**

**在原有的Monaco Editor组件中增添部分代码以实现此功能，整体设计与实现思路如下：**

原组件通过钩子定义 handleEditorDidMount 回调函数，在编辑器挂载完成后被调用，用于设置编辑器的主题和注册代码优化建议的事件监听器。

使用 editor.addCommand 注册了一个快捷键命令Alt+Shift+L，当用户在编辑器中按下这个快捷键时，会触发代码优化建议的生成。

editor.addCommand 将快捷键与优化建议生成函数绑定，使得用户可以通过键盘快捷操作来获取代码优化建议。

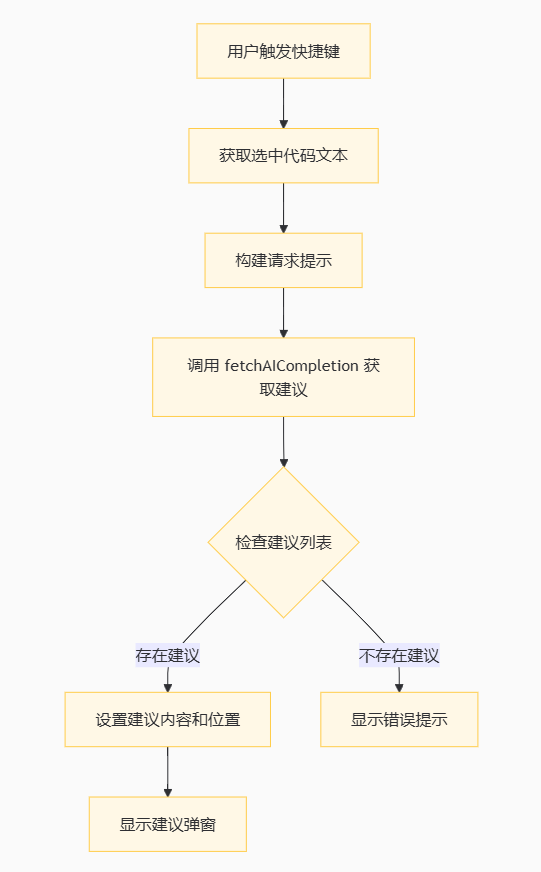
editor.getSelection 和 editor.getModel().getValueInRange 函数用来获取用户选中的代码文本。

构建一个请求提示（prompt），格式为 “为以下代码提供一些建议：\n${selectedText}”，传递给 AI 模型。

使用 fetchAICompletion 函数异步获取 AI 模型返回的优化建议，并将其存储在组件状态中。

将 AI 生成的优化建议展示给用户，通过弹出窗口或者编辑器内部的浮动层来展示；如果没有获取到建议或者发生错误，通过 toast 组件显示相应的提示信息。

**以下流程图描述了用户在选中一段代码并按下Alt+Shift+L快捷键时，如何触发代码优化建议功能，并以浮窗形式反馈给用户的过程。**



**本功能设计体现的设计模式包括：**

1. 命令模式：

通过 editor.addCommand 添加快捷键命令，体现了命令模式，将操作封装成对象，使得可以用不同的方式请求操作，在此处用户作为命令的发出者，可通过快捷键触发（与5.3思路相似）。

* 1. **代码内联补全**

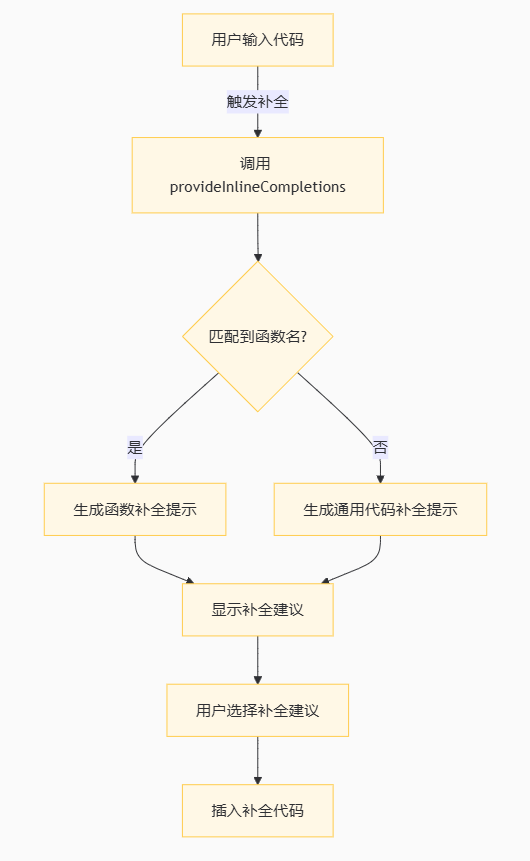
**在原有的Monaco Editor组件中增添部分代码以实现此功能，整体设计与实现思路如下：**

原组件通过useCallback钩子定义了一个handleEditorDidMount回调函数，这个函数在编辑器挂载完成后被调用，用于设置编辑器的主题、注册内联补全提供者。

使用Monaco Editor的API来定义内联补全提供者，使用registerInlineCompletionsProvider来注册，通过provideInlineCompletions方法来异步获取补全建议，provideInlineCompletions是每次 Monaco 编辑器触发内联补全时调用的回调函数。在实际补全过程中，其实现逻辑大致为捕获光标所在行的内容并传给大模型，同时预设了一部分补全要求，如果通过正则表达式/#\s\*(\S.\*)/来匹配用户可能输入的函数名，并基于此来生成补全提示。

此外，使用setTimeout来实现防抖逻辑，避免在用户输入时频繁发送请求。

**以下流程图描述了用户输入代码时，如何触发内联补全功能，并显示补全建议给用户。**



**本功能设计体现的设计模式包括：**

1. 观察者模式：

编辑器的事件监听（onDidChangeModelContent）体现了观察者模式，编辑器作为主题，组件作为观察者，对编辑器的状态变化做出响应。

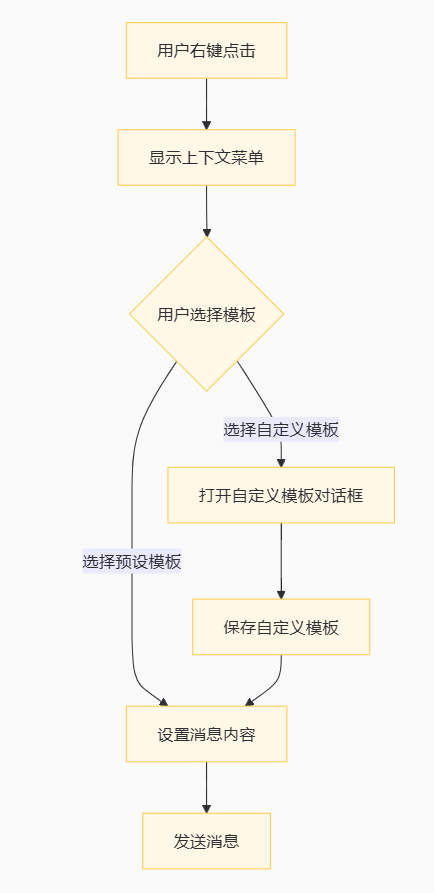
1. 工厂模式：

GoogleGenerativeAI类可被视为一个工厂，它负责创建和配置AI模型实例。这种模式允许代码在不直接依赖具体实现的情况下，生成和使用AI模型。

* 1. **有效的Prompt模板预设与自定义**

**在原有的ChatInput组件中集成新组件以实现此功能，加入prompt预设模板类管理，整体设计与实现思路如下：**

**以下流程图描述了用户使用此功能的过程。**



**本功能设计体现的设计模式包括：**

1. 简单工厂模式：

简单工厂模式的体现主要通过TemplateManager类来展示。

工厂模式是一种创建型设计模式，其核心思想是将对象的创建与使用分离，通过一个共同的接口来创建一系列相关或相互依赖的对象，而不需指定它们具体的类。

TemplateManager 类扮演了“工厂”的角色，它包含了一系列与模板相关的操作，包括获取模板列表、获取单个模板、保存新模板或更新现有模板。这个类封装了模板的创建和查找逻辑，对外提供了简单的接口来访问模板数据。

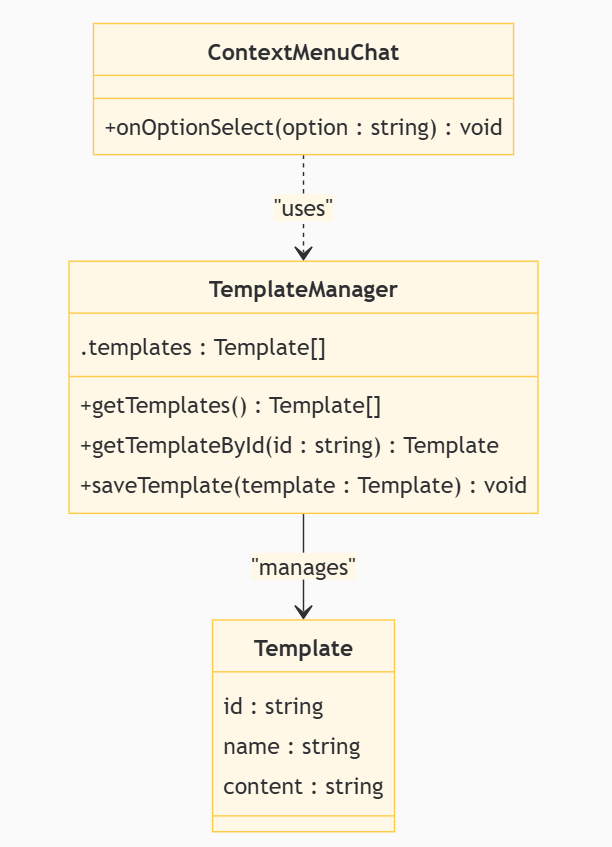
在TemplateManager类中，模板的创建和修改不是由外部代码直接操作，而是通过这个类提供的公共方法来实现。比如当需要添加一个新的模板或更新现有模板时，外部代码只需调用saveTemplate方法，传入模板的数据，而具体的添加或更新逻辑都封装在TemplateManager内部。

在工厂模式中，客户端代码不需要知道对象是如何被创建和组装的，只需要知道如何使用它们。ContextMenuChat组件通过TemplateManager获取模板列表，并展示给用户，而无需关心模板数据的具体来源和存储细节，提高了代码的灵活性和可维护性。

本类的相关代码如下图所示：



涉及的类图如下图所示：



* 1. **代码翻译转换**

**在侧边栏新增功能按钮以快速进行不同编程语言的代码转换，整体设计与实现思路如下：**

**本功能使用简单工厂模式进行设计，涉及工厂模式的部分代码如下：**

基础语言接口

export interface Language {

name: string; // 语言显示名称，如 "Python", "Java" 等

value: string; // 语言标识值，用于系统内部识别

getDescription(): string; // 获取语言描述的方法

}

具体语言实现

以 Python 和 Java 为例：

export class Python implements Language {

name = "Python";

value = "python";

getDescription(): string {

return "Python is a high-level, interpreted programming language.";

}

}

export class Java implements Language {

name = "Java";

value = "java";

getDescription(): string {

return "Java is a high-level, class-based, object-oriented programming language.";

}

}

工厂类实现

export class LanguageFactory {

// 根据类型创建具体语言实例

public static createLanguage(type: string): Language | null {

switch (type) {

case "python": return new Python();

case "java": return new Java();

case "javascript": return new JavaScript();

case "cpp": return new Cpp();

case "csharp": return new CSharp();

default: return null;

}

}

// 获取所有支持的语言列表

public static getLanguages(): Language[] {

return [

new Python(),

new Java(),

new JavaScript(),

new Cpp(),

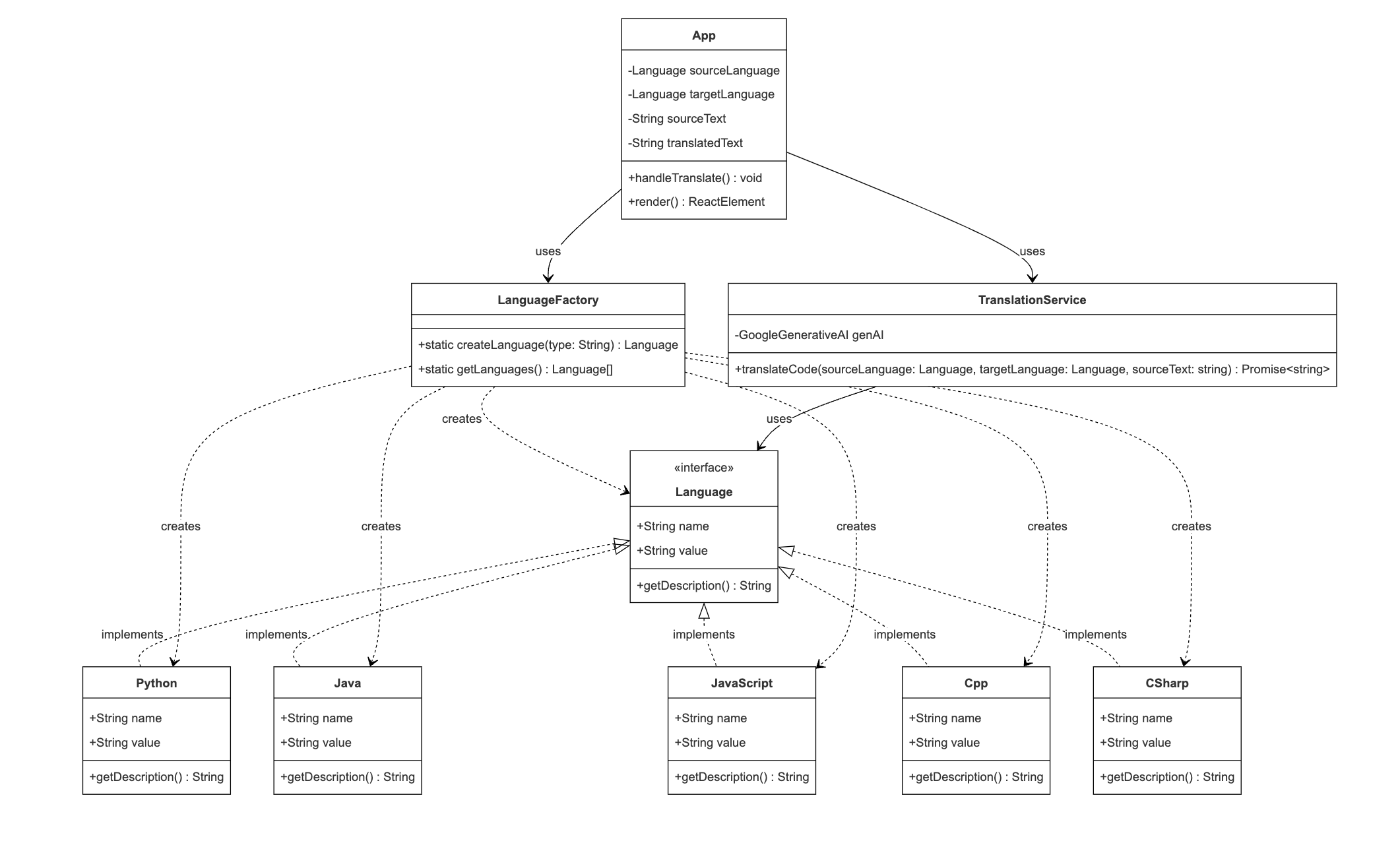
new CSharp()

];

}

}

**本功能完整的类图如下：**

****

**类图说明如下：**

Language 接口

1. 定义了语言对象的通用接口
2. 包含name和value属性
3. 定义getDescription()方法

具体语言类

1. Python, Java, JavaScript, Cpp, CSharp
2. 实现Language接口
3. 各自实现特定的getDescription()方法

LanguageFactory 类

1. 提供createLanguage静态工厂方法
2. 提供getLanguages方法返回所有支持的语言
3. 负责创建具体的语言对象

TranslationService类

1. 使用Language对象进行翻译服务
2. 集成了Google AI翻译功能

App 组件

1. 使用LanguageFactory创建语言对象
2. 通过TranslationService进行代码翻译
3. 管理UI状态和用户交互

**关系说明如下：**

1. 所有具体语言类实现Language接口
2. LanguageFactory创建具体的语言对象
3. App使用LanguageFactory和TranslationService
4. TranslationService依赖Language接口

**使用简单工厂模式进行设计的优点如下：**

封装性：

1. 语言创建逻辑集中在工厂类中
2. 具体语言实现细节对客户端透明

扩展性：

1. 可以轻松添加新的语言类型
2. 只需实现Language接口并在工厂中添加创建逻辑

维护性：

1. 清晰的类层次结构
2. 职责划分明确
3. 便于测试和修改

灵活性：

1. 支持运行时动态选择语言
2. 可以根据需要加载不同的语言实现

**开发过程中进行了多次迭代优化，提高了程序性能和可靠性，简化了后续开发的难度**：

1. LLM API调用优化

之前版本的方法：通过 Pyodide 调用 translate.py

在之前的版本中，为了在前端实现翻译功能，采用了 Pyodide。具体实现方式是在 root.tsx 中加载 Pyodide，进而调用 translate.py 进行翻译操作。此方法的优点在于，可以在不依赖后端的情况下实现翻译逻辑，但也带来了性能和复杂性方面的不足。

当前版本的改进：直接在前端调用 LLM API

直接在前端通过 API 请求调用 LLM，有以下几个显著的优势：

**性能**

1. 低延迟：直接调用 LLM API 避免了加载和运行 Pyodide 的步骤，减少了请求的往返时间，从而显著降低了延迟。
2. 高效资源利用：直接 API 调用不需要在前端加载 Pyodide，因此减少了内存和 CPU 的占用，提高了资源利用效率。

**简化开发**

1. 减少复杂性：省去了 Pyodide 的加载和初始化，前端代码逻辑得到简化，降低了开发复杂性。
2. 易于维护：API 调用依赖项减少，后续维护时无需关心 Pyodide 版本更新或其兼容性问题，降低了维护成本。

**可靠性**

1. 稳定性提升：避免了在浏览器中运行 Python 代码导致的不确定性，减少了潜在的浏览器兼容性问题，提高了系统稳定性。
2. 错误处理机制：API 通常提供更完善的错误处理和监控机制，使得错误捕获和处理更为便捷，降低了前端复杂错误处理的负担。
3. 优化推理速度：设置合理的 generationConfig 参数，控制生成的最大输出长度和温度。
4. 内存管理：在前端使用状态管理（useState）来管理翻译结果的状态。
5. 编程语言选择器优化

之前版本的方法：硬编码

在之前的硬编码方式中，语言列表通过对象数组的方式直接写在前端代码中，这种直接在列表中硬编码了语言描述信息的方式不易扩展和维护：

const languages = [

{ name: "Python", value: "python", description: "Python is a high-level, interpreted programming language." },

{ name: "Java", value: "java", description: "Java is a high-level, class-based, object-oriented programming language." },

{ name: "JavaScript", value: "javascript", description: "JavaScript is a high-level, just-in-time compiled, multi-paradigm programming language." },

{ name: "C++", value: "cpp", description: "C++ is a general-purpose programming language created as an extension of the C programming language." },

{ name: "C#", value: "csharp", description: "C# is a modern, object-oriented programming language developed by Microsoft." }

];

当前版本的方法：工厂模式

使用工厂模式，将每种语言的逻辑封装到独立的类中，并通过工厂方法来实例化这些类的方式具有：

1. 扩展性：可以方便地添加新语言，而无需更改原有代码，符合开闭原则。
2. 代码结构清晰：将每种语言的描述封装在独立的类中，代码模块化且易于理解。
3. 便于维护：每种语言的更改仅限于其自身类中，减少了对整体代码的影响。
   1. **项目级代码生成**

**在侧边栏新增功能按钮以快速实现对项目级代码的生成，整体设计与实现思路如下：**

算法描述

1. 生成项目代码: 使用生成式AI模型，根据用户输入的项目描述生成代码。
2. 保存代码: 将生成的代码保存为Jupyter Notebook文件，并执行该文件。

复杂度分析

1. 生成代码的时间复杂度主要取决于AI模型的响应时间。
2. 保存和执行代码的时间复杂度为O(1)，因为它们是固定的文件操作。

**关键代码片段如下：**

const generateProject = async (projectPath: string, projectDescription: string, projectLanguage: string): Promise<string> => {

  try {

    const prompt = `Generate a project with the following details:\n\nPath: ${projectPath}\nDescription: ${projectDescription}\nLanguage: ${projectLanguage}`;

    const model = genAI.getGenerativeModel({

      model: "gemini-1.5-flash",

      generationConfig: { ... }

    });

    const result = await model.generateContent(prompt);

    const notebookContent = result.response.text();

    const notebookPath = `${projectPath}/generated\_project.ipynb`;

    await fetch('/api/save-notebook', { ... });

    await fetch('/api/execute-notebook', { ... });

    return notebookPath;

  } catch (error) {

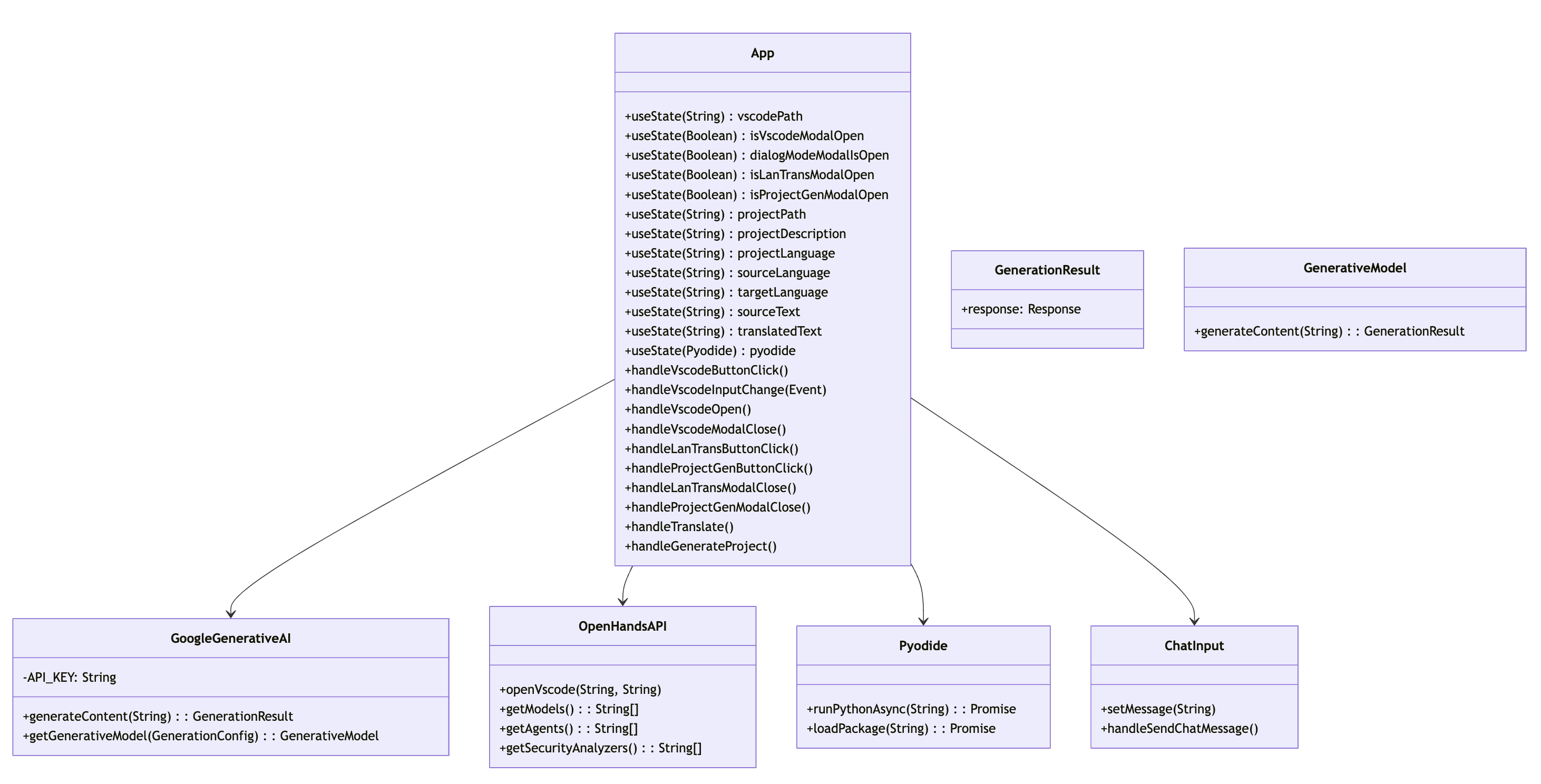
    console.error("Error generating project:", error);

    return "Generation failed";

  }

};

**代码被组织在src目录下，分为components、services、utils等子目录，本功能涉及类图如下：**



接口设计

API接口

1. /api/save-notebook: 保存生成的代码到指定路径。
2. /api/execute-notebook: 执行生成的Jupyter Notebook文件。

接口文档

1. POST /api/save-notebook: 接收path和content参数，保存文件。
2. POST /api/execute-notebook: 接收path参数，执行指定路径的Notebook。
   1. **潜在问题漏洞修复**
      1. make build环节报错

构建时多次卡在 Installing frontend dependencies with npm...经过多次尝试，最终选择在使用make build前：

·打开OpenHands文件夹下子文件夹frontend；使用 npm config set registry https://registry.npmmirror.com 更换npm下载的镜像源；退出文件夹并执行make build

·注：不可提前在frontend文件夹中先执行npm install命令，否则会发生msw等版本冲突，构建失败，多项报错其实质为提前下载的版本与frontend在package.json中指定版本冲突所致

* + 1. make run 环节报错 - UID冲突

·在使用windows wsl ubuntu进行部署时需注意尽量避免在root用户下进行部署，否则在使用make run命令运行项目后，可成功映射至端口并打开http://localhost:3002启动界面，但在与chat初次交互过程中，Client多次初始化失败，在系统中创建用户及其工作目录的过程中，源码出现与root用户冲突并无法创建的情况。

·漏洞修复：修改client.py文件中的def \_init\_user(self, username: str, user\_id: int)函数，当检测到创建用户与root冲突后，选择未使用的UID以避免冲突。



* + 1. make run报错 - 网络环境问题

·出现与在线语言模型的链接问题，程序有多次尝试机制但仍无法连接。

·问题修复：需修改DNS设置，使用公共DNS服务器，

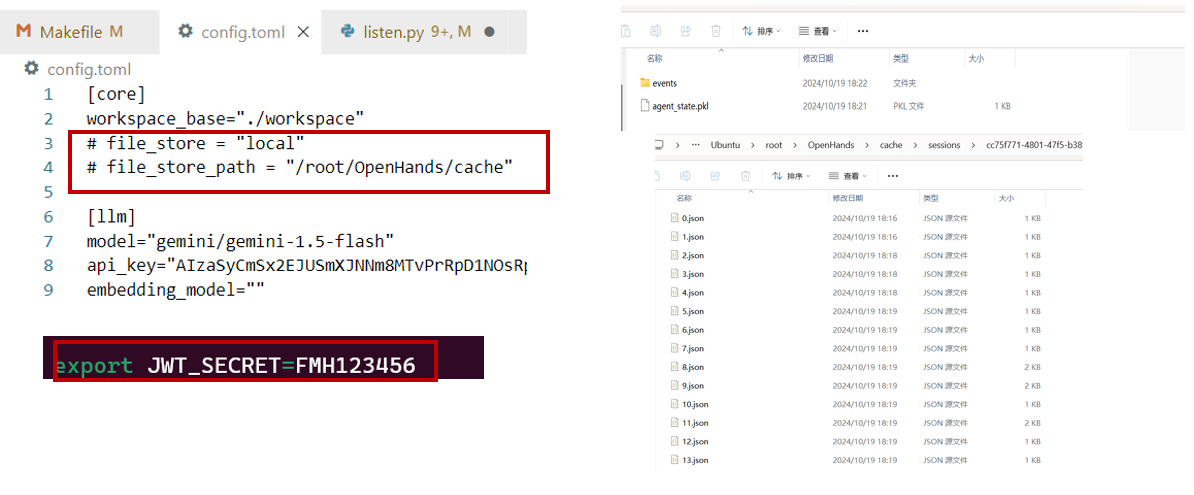
使用sudo nano /etc/resolv.conf命令并在对应文件末尾添加

nameserver 8.8.8.8 nameserver 8.8.4.4后保存关闭文件。请务必确保本机网络环境畅通，本系统在实现多代理交互问答时在调用国外大模型时严重依赖网络环境。若使用Gemini，可提前在终端测试，使用wget www.google.com命令能否成功响应。

* + 1. 历史对话无法重载导入 - 启动出现Invalid token弹窗

·使用 make run 命令启动应用程序后，在网页端会先从交互窗口停留一下并闪退回主界面，同时网页会弹出报错窗口Invalid token。

·问题修复：主要问题在于未能验证用户身份并获取会话信息。报错原因为找不到正确的sid与令牌，故无法恢复。用户需在config.toml文件中设置保存方式为本地存储并将路径设置为绝对路径，同时在.bashrc文件中新增固定的JWT密码，每次重启应用时便直接加载上次的会话文件，交互窗口与Jupyter内容都将保存。在本地也可见其以json与pickle的形式进行存储。



1. **部分测试用例示例**

|  |  |  |
| --- | --- | --- |
| 测试功能 | 测试用例 | 测试结果 |
| 单个文件导入 | 上传一个图像处理的简单的Java程序文件 | 能够成功上传并打开查看编辑 |
| 项目级代码导入 | 连接GitHub，上传整个图像处理课程大作业项目 | 能够成功上传并打开编辑 |
| 智能问答 | 询问“设计模式有什么”并查看回应 | 回答了个种设计模式并给出详细解释 |
| 上下文联系 | 首先让其“创建一个名为test的文件夹”，之后“删除这个文件夹”，看能否正确删除 | 成功创建并正确删除相应文件夹 |
| 单个代码文件生成 | 要求请在工作区生成一个简单的贪吃蛇小游戏代码文件 | 能够生成文件，并且能够查看编辑 |
| 项目级代码生成 | 让其生成一个扫雷小游戏的完整项目 | 需要描述很详细才能生成为项目的结构，有README等文件，有用户系统，前后端分离 |
| 生成注释 | 选中整个文件的所有代码后按Alt+shift+P，是否能成功生成注释 | 每行代码后面都有代码生成 |
| 代码优化建议 | 选中部分代码后按Alt+shift+L，是否能成功生成建议浮窗 | 成功生成代码优化建议 |
| 代码自动补全 | 编写冒泡排序法，写到一半停下，看能否自动补全 | 光标停留一段时间后有自动补全的代码出现，按tab能够补全 |
| 代码转换 | 按左侧转换按钮，将整个图像处理的Java文件输入并要求转化为python | 可自由选择转换前后的语言，并能够正确转换 |
| 对大模型的API接入，32K上下文支持 | 调用不同的API如Gemini或OpenAI测试能否使用 | 成功 |
| 支持交互模式修改、知识库问答 | 切换交互模式，输入AnyhingLLM网址作为知识库的一部分，上传文件，询问“什么是anythingLLM”；传入图像处理大作业询问某个函数的调用情况 | 成功上传文件与网址，可向量化，回答比之前更加详细准确；对于图像处理作业的函数调用回答内容准确 |

**7. 二次开发进展情况**

**7.1 时间线**

| 周数 | 关键任务 | 完成目标 |
| --- | --- | --- |
| 第1周 | 项目启动和团队组建 | 完成团队组建，分配任务，明确角色和责任 |
| 第2-3周 | 需求分析和确认 | 完成需求分析，确认需求优先级，制定详细设计计划 |
| 第4周 | 技术选型和架构设计 | 确定技术栈，完成系统架构设计 |
| 第5-8周 | 功能开发和模型集成 | 完成代码整库分析、文档分析增强、代码定制化交互式生成等功能开发 |
| 第9周 | 工具和知识库集成 | 完成工具库、代码库和知识库的集成 |
| 第10周 | API设计与模型优化 | 设计统一API接口，优化模型推理效率，提升生成速度和准确率 |
| 第11周 | 测试和性能调优 | 进行系统测试，根据反馈进行性能调优 |
| 第12周 | 项目总结和文档编写 | 完成项目总结报告，编写和更新项目文档 |

**7.2 关键里程碑**

·项目启动：完成团队组建，分配任务，明确角色和责任。

·需求确认：完成需求分析，确认需求优先级，制定详细设计计划。

·技术选型：确定技术栈，完成系统架构设计。

·功能开发完成：完成所要求的功能模块开发和模型集成。

·集成测试：完成工具和知识库的集成，进行系统测试。

·性能优化：优化系统性能，提升用户体验，根据测试反馈进行功能迭代和优化。

·项目总结：完成项目总结报告，确保所有功能达到预期目标。

1. **项目需求实现总结表**

| 需求内容 | 是否为二次开发 | 实现情况详述 |
| --- | --- | --- |
| 支持依托于开源的 LLM 与 Embedding 模型，实现基于开源模型的离线私有部署 | 否 | 源项目已实现，小组完成部署工作 |
| 支持统一的 API 调用，具备对各类开源大模型及模型 API 的扩充能力，具备不少于32K上下文支持 | 否 | 源项目已实现，所使用的大模型已满足此需求，支持通过统一接口对主流在线模型与本地模型的调用 |
| 支持集成工具库、代码库及知识库 | 是 | 在OpenHands前端实现集成anythingllm的知识库部分功能，调用其api，使得用户可以切换对话模式，连接自己的知识库，进行知识库文件与链接的上传，工作区的创建与删除，知识库两种对话模式的修改，延用OpenHands原有的交互窗口。测试对基础的文本知识库进行问答与传入图像处理大作业构建代码知识库进行问答，反馈的准确率与丰富度相比于仅与代理进行交互要更有高 |
| 支持工具调用能力 | 是 | 在代码编辑区组件对monaco-editor进行调用，将注释、优化、补全等功能封装为工具进行快捷调用；支持对VSCode工具的集成调用以协同开发；代码解释翻译等功能调用大模型并规定返回内容，实现工具的调用能力等等 |
| 支持实现代码库级别的代码理解、仓库项目级的代码文件优化、生成 | 是（50%） | OpenHands原生的问答功能可以实现一定程度的代码仓库级分析理解与优化，且可生成简单的项目级代码；现已对其进行扩展，通过连接知识库进行问答，相比于原生的智能问答效率更高，在理解与回复的速度方面更快，回答的准确率两模式不相上下（传入图像处理大作业进行测试，在迭代ppt中提及并演示）。同时，通过测试微调有效的prompt，实现了对一些常见prompt的封装，例如使用我组在系统中提供的创建项目级代码的prompt可以实现更高准确率且更完善的项目及代码。 |
| 具有代码补全、代码建议、注释生成、代码转换及代码智能问答功能 | 是 | OpenHands原生的编辑器组件仅支持快捷保存等，现已注册内联补全函数，快捷生成优化建议与注释，实现代码转换翻译等各项功能，代码智能问答其已实现，现扩充其问答模式。 |