SysY2022E 扩展语言编辑器

SysY2022E Language Support

系统概要说明书

成员信息： 李晓坤　信息安全　信安 211　U202141863

王若凡　信息安全　信安 211　U202141852

王宠爱　信息安全　信安 211　U202141853

成豪　信息安全　信安 211　U202141880

指导教师： 崔 晓 龙

目录

1. 导言

1.1. 编写目的

本文档旨在为 SysY2022E 扩展语言编辑器的开发项目提供一个清晰、全面的系统概要说明。通过本文档，明确编辑器的系统体系结构、接口定义、模块设计以及故障监测和处理机制，从而为项目开发团队提供一个明确的开发指南，测试人员也可根据本文档对编辑器进行全面测试以确保产品满足预期要求，为未来的系统维护和扩展奠定基础。同时，本文档也旨在为潜在用户提供必要的产品信息和操作指南，以帮助他们更好地理解和使用 SysY2022E 扩展语言编辑器。

1.2. 范围

系统概要说明全面覆盖了 SysY2022E 扩展语言编辑器的系统结构设计及规格分析、接口定义、模块设计、故障监测和处理机制以及开发平台等关键信息。在整个项目中，该文档充当了规划蓝图和操作指南的角色，不仅为开发团队提供了明确的需求和开发方向，也为测试团队提供了测试依据，同时还帮助系统管理员和用户了解和使用系统。通过这份文档，项目团队能够确保高效开发，减少沟通误解，提升产品质量，并有效解决从项目规划到实施过程中的技术选型、功能实现及用户交互等核心问题。

1.3. 术语定义

表 1 术语定义表

术语	定义
SysY2022E 语言	SysY2022 是 C 语言的一个子集，最初是为“全国大学生系统能力大赛-编译器大赛”设计的迷你编程语言。SysY2022E 语言是在其基础上拓展的语言。
LSP	微软指定的 Language Server Protocol，它标准化了语言工具和代码编辑器之间的通信。
ANTLR4	ANTLR4 是一款功能强大的解析器生成器，用于读取、处理、执行和翻译结构化文本或二进制文件。

1.4. 引用标准

1）IEEE 830-1998（已被 IEEE 29148 取代）：软件需求规格说明标准

提供了编写高质量软件需求规格说明书的指南，确保需求文档的完整性和一致性。

2）IEEE 1016-2009：软件设计描述标准

规范了软件系统设计文档的内容和格式，提供了系统设计的高级视图结构。

3）ISO/IEC/IEEE 29148-2018：系统与软件工程——生命周期过程——需求工程

结合了需求工程的最佳实践，适用于需求开发和管理，提供了编写需求和系统描述文档的详细指南。

4）IEEE 1233-1998（已被 IEEE 29148 取代）：系统需求规格说明标准

涉及系统级别的需求和设计，提供了编写系统需求说明文档的结构和指南。

5）ISO/IEC 25010:2011：系统与软件工程——系统和软件质量模型

提供了软件和系统质量的标准模型，用于评估系统的非功能性需求，如性能、安全性、可用性和可维护性。

6）ISO/IEC/IEEE 15288:2015：系统与软件工程——系统生命周期过程

定义了系统工程和软件工程中的生命周期过程，为系统开发和维护提供了标准化的指南。

1.5. 参考资料

[1] IEEE 830-1998. IEEE Recommended Practice for Software Requirements Specifications. Institute of Electrical and Electronics Engineers.

[2] IEEE 1016-2009. IEEE Standard for Information Technology—Systems Design—Software Design Descriptions. Institute of Electrical and Electronics Engineers.

[3] ISO/IEC/IEEE 29148-2018. Systems and software engineering — Life cycle processes — Requirements engineering. International Organization for Standardization / Institute of Electrical and Electronics Engineers.

[4] IEEE 1233-1998. IEEE Guide for Developing System Requirements Specifications. Institute of Electrical and Electronics Engineers.

[5] ISO/IEC 25010:2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. International Organization for Standardization.

[6] ISO/IEC/IEEE 15288:2015. Systems and software engineering — System life cycle processes. International Organization for Standardization / Institute of Electrical and Electronics Engineers.

1.6. 版本更新信息

在项目开发过程中，使用 github 进行项目版本管理，共迭代了 5 个版本。在每一次版本迭代时，关于具体的修改时间、修改位置、修改内容均在 github 仓库中有详细记录。由于记录内容繁杂，不便于在报告中展示，因此在文档中省略关于版本更新信息的说明，如若想要查看相关版本更新信息，请查看本项目的 github 仓库。

表 2 版本更新表

修改编号	修改日期	修改后版本	修改位置	修改内容概述
U005	2024.06.08	v5.0	见仓库	见仓库

2. 概述

2.1. 设计采用的标准和方法

2.1.1. 设计标准

1）IDE 集成标准：遵循主流 IDE（如 vscode、Eclipse）的插件开发标准和接口，确保无缝集成和兼容性。

2）编程语言规范：严格遵循 SysY2022E 语言的语法和语义规范，确保编辑器的准确性和可靠性。

3）用户界面设计标准：遵循现代用户界面设计原则，提供直观、易用的操作界面。

2.1.2. 设计方法

1）敏捷开发：采用敏捷开发方法，快速迭代，及时响应用户反馈和需求变更。

2）面向对象设计：利用面向对象的设计原则，提高代码的可维护性和可扩展性。

3）设计模式：应用设计模式（如观察者模式、工厂模式等），优化代码结构，提高系统灵活性。

2.2. 系统结构

1）模块化设计：将整个系统划分为多个独立、可复用的模块，便于开发和维护。

2）扩展性：设计开放式的系统架构，允许未来添加新的功能或模块。

3）性能优化：在系统结构设计时考虑性能因素，使用高效的数据结构和算法，减少不必要的计算和内存消耗。

3）安全性：确保系统结构的安全性，防止潜在的安全漏洞和恶意攻击。

2.3. 错误处理机制

1）错误分类与定位：将错误详细分类，分为语法错误、语义错误、系统错误等，并提供精确的错误定位信息，帮助用户快速识别和解决问题。

2）错误提示与修复建议：为用户提供友好的错误提示信息，同时提供可能的修复建议或自动修复功能，降低用户解决错误的难度。

3）日志记录与分析：建立完善的日志记录机制，记录系统运行过程中的关键信息和错误信息，便于后续的问题分析和系统优化。

4）异常处理：在系统中实现完善的异常处理机制，确保在遇到异常情况时能够优雅地处理，避免系统崩溃或数据丢失。

综上所述，设计采用的标准和方法、系统结构的考虑以及错误处理机制的考虑都是为了构建一个功能强大、稳定可靠、易于维护和扩展的 SysY2022E 语言编辑器。

3. 规格分析

3.1. IDE 集成

- 功能需求：编辑器需要与 vscode 无缝集成。
- 分析：为实现这一功能，需要开发符合 vscode 插件开发规范的接口，确保编辑器可以作为 vscode 的一个扩展被安装和加载。允许用户通过 vscode 直接编辑、运行和调试程序。集成后，编辑器应提供与 vscode 一致的界面风格和用户体验，同时保留 SysY2022E 语言的特定功能。

3.2. 语法高亮和悬浮提示

- 功能需求：编辑器应对代码进行语法高亮，并为标识符提供悬浮提示。
- 分析：需实现一个语法分析器，用于解析代码，并根据语法规则进行高亮显示。同时，需要构建一个符号表来存储标识符及其相关信息，以便在鼠标悬停时提供提示。通过不同颜色来对不同的标识符进行高亮并显示相应的信息提示。

3.3. 语法检查与错误提示

- 功能需求：编辑器应检查程序中的语法错误，并提供错误位置和信息。
- 分析：在语法分析器的基础上，增加错误检测机制。当遇到不符合语法规则的代码时，记录错误位置并生成相应的错误信息。提供实时的语法检查与错误提示，以确保代码符合语言规范。具体功能包括：实时检测代码的语法错误，如变量声明、函数定义等问题；在发现错误时及时在相应位置进行标记，并提供详细的错误提示信息；通过代码高亮帮助开发者更好地识别和理解代码结构；同时，提供

自动补全功能以减少语法错误，并包括代码格式化工具以调整代码布局和风格，使其符合编码规范。该功能旨在提高代码质量和开发效率，减少语法错误和调试时间。

3.4. 静态语义检查

- 功能需求：编辑器需进行静态语义检查，包括类型和作用域检查。
- 分析：在语法分析的基础上，进行语义分析。静态语义检查是代码质量保障的重要一环，SysY2022-Editor 通过内置的静态语义分析器，在编译之前对 SysY2022E 程序进行详细的语义检查。这些检查不仅限于语法层面，还包括对代码逻辑、类型、作用域等方面的深入分析。

3.5. 修复建议

- 功能需求：对于检测到的语法和静态语义错误，编辑器应提供修复建议，并支持自动修复。
- 分析：自动检测和分析程序中的语法和静态语义错误，并提供针对性的修复建议。该功能在检测到错误时，会基于内置的错误分析引擎和最佳实践数据库，给出包括修改语法、调整变量类型等智能修复方案，并提供详细解释和示例代码，帮助用户理解和应用。此外，还支持自动修复和用户反馈，以提高修复效率和准确性。此功能旨在提升用户的编程效率和代码质量。

3.6. 程序错误检查

- 功能需求：编辑器需要检查程序中的常见逻辑错误，如死循环、无用变量等。
- 分析：需要具备一套全面的实时错误检查机制，可自动检测 SysY2022E 程序中的常见错误，如死循环、无用变量、死代码段和数组越界等，旨在帮助用户提升代码质量和减少运行时问题。编辑器不仅提供详细的错误信息，包括错误类型、位置和可能的修复建议，还可能为某些错误提供自动修复或代码重构的建议，从而显著提高开发效率，减少代码错误导致的运行问题。

3.7. 代码重构

- 功能需求：编辑器应支持代码重构功能，如变量/函数改名、抽取新函数等。
- 分析：帮助用户优化和重构 SysY2022E 程序，这些功能包括变量/函数改名、抽取

新函数、内联函数、提取局部变量以及其他多种重构操作。代码重构功能需要深入理解代码的语义结构。实现时，可以利用抽象语法树（AST）来表示代码结构，并基于 AST 进行各种重构操作。同时，需要确保重构后的代码保持原有的功能和性能。

4. 系统体系结构

4.1. 系统模块划分

4.1.1. 用户界面层（User Interface Layer）

- IDE 集成：提供与主流 IDE（如 Visual Studio Code, IntelliJ IDEA 等）的无缝集成，允许用户在熟悉的开发环境中使用 SysY2022 扩展语言。
- 语法高亮和悬浮提示：实现代码的语法高亮显示，以及当用户将鼠标悬停在代码元素上时，显示相关的提示信息。

4.1.2. 编辑器核心层（Editor Core Layer）

- 语法检查与错误提示：实时监控用户输入，进行语法分析，并在发现错误时立即提示用户。
- 静态语义检查：对代码进行静态语义分析，确保代码的语义正确性，并在发现问题时提供反馈。

4.1.3. 分析与修复层（Analysis and Fix Layer）

- 修复建议：当检测到错误或潜在问题时，提供修复建议，帮助用户快速修正代码。
- 程序错误检查：深入分析代码逻辑，识别潜在的运行时错误，并给出警告。

4.1.4. 重构引擎层（Refactoring Engine Layer）

- 代码重构：提供一系列重构操作，如重命名、提取方法、内联变量等，帮助用户改善代码结构，提高代码质量。

4.1.5. 后端服务层（Backend Service Layer）

- 集成编译器：与 SysY2022 语言的编译器集成，实现代码的编译功能，并提供编译错误反馈。

4.1.6. 数据存储层（Data Storage Layer）

- 代码仓库：提供代码版本控制功能，支持用户管理代码的历史版本。
- 配置存储：存储用户的个性化配置，如主题、快捷键设置等。

4.1.7. 插件系统（Plugin System）

插件支持：开放插件接口，允许第三方开发者为编辑器开发扩展功能，如额外的代码分析工具、集成其他服务等。

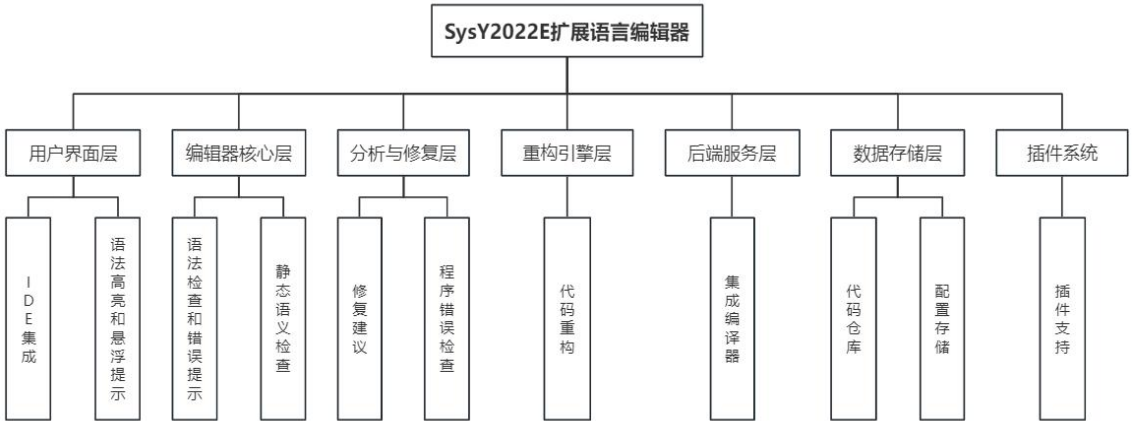

图 1 系统模块划分图

4.2. 系统工作流程

1） 启动编辑器：

- 用户启动代码编辑器应用程序。
- 应用程序加载用户界面层，准备接收用户输入。

2） 代码编辑：

- 用户在用户界面层编写或修改代码。
- 编辑器核心层实时监控用户输入，提供语法高亮、代码完成建议等功能。

3） 代码保存和版本控制：

- 用户选择保存代码。
- 编辑器核心层将代码变更传输给数据存储层，持久化存储到文件系统或数据库。
- 如集成有版本控制系统，编辑器将触发对应的版本控制操作（如提交、推送等）。

4) **代码分析:**

- 用户请求代码分析，例如查找潜在错误或代码质量问题。
- 编辑器核心层将代码传送到分析与修复层。
- 分析与修复层对代码进行分析，检测问题并返回分析结果给编辑器核心层。
- 编辑器核心层将分析结果展示给用户。

5) **代码重构:**

- 用户选择应用一个或多个重构操作，例如重命名变量或更改代码结构。
- 编辑器核心层将用户的重构请求发送给重构引擎层。
- 重构引擎层执行请求的操作并返回重构后的代码。
- 用户界面层展示重构后的代码并等待用户确认。

6) **关闭编辑器:**

- 用户完成代码编辑工作，关闭编辑器。
- 用户界面层触发关闭事件，提示用户保存未保存的工作。
- 编辑器核心层确保所有数据都保存到数据存储层。
- 编辑器应用程序释放资源并成功关闭。

4.3. 各层及其模块之间的关系和通信

4.3.1. 用户界面层与编辑器核心层

- 关系: 用户界面层接收用户的输入，并将这些输入传输到编辑器核心层进行处理。编辑器核心层分析用户的输入，执行语法检查和静态语义检查。当核心层发现错误或需要显示提示时，将这些信息传回给用户界面层，用户界面层再将这些信息展示给用户。
- 通信: 此通信通常是通过内部 API 调用实现的，可以是同步或异步的，这取决于具体实现和用户操作的实时或非实时反馈需求。

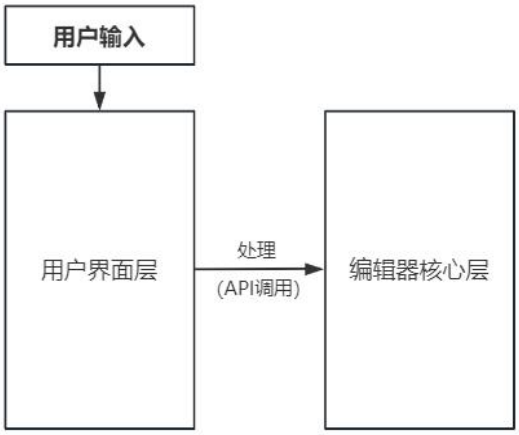

图 2 用户界面层与编辑器核心层关系图

4.3.2. 编辑器核心层与分析与修复层

- 关系： 当编辑器核心层检测到潜在的代码问题时，会请求分析与修复层进行更深入的分析，并提出修复建议。分析与修复层处理这些请求，并返回一系列可能的解决方案。
- 通信： 通过内部调用进行，分析与修复层可能会作为一个独立的服务运行，使用消息队列或者 RPC（远程过程调用）机制与编辑器核心层交互。

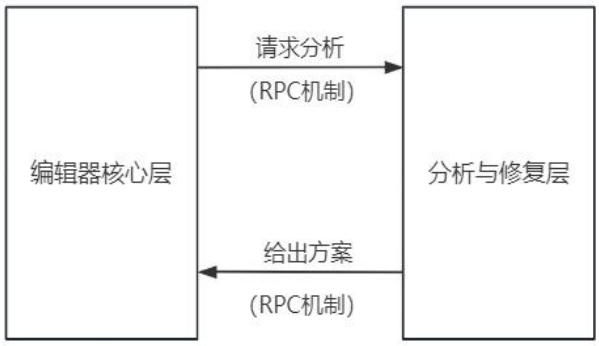

图 3 编辑器核心层与分析与修复层关系图

4.3.3. 编辑器核心层与重构引擎层

- 关系： 当用户希望对代码进行结构性改变时，编辑器核心层会调用重构引擎层的功能。重构引擎层负责应用重构操作，并返回重构后的代码。
- 通信： 类似于分析与修复层的通信机制，重构引擎层可以通过内部 API 进行调用，也可能使用更复杂的通信协议，以支持重构操作的复杂性和计算需求。

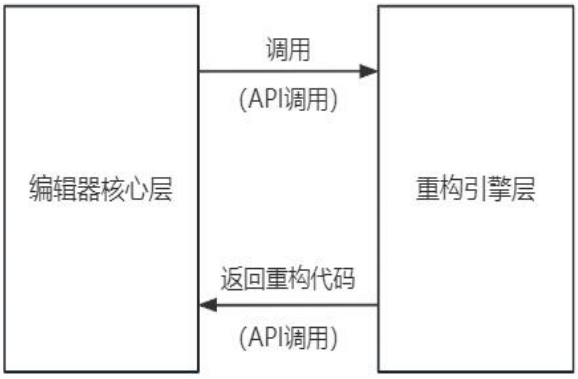

图 4 编辑器核心层与重构引擎层关系图

4.3.4. 各层与后端服务层

- 关系： 后端服务层为其他层提供支持服务，如编译和调试。当用户请求编译或调试操作时，相应的层会与后端服务层通信，执行需要的服务。
- 通信： 服务层可能与其他层通过网络协议通信，特别是在分布式系统中。这可能包括 HTTP 请求或使用专用的编译器/调试器服务协议。

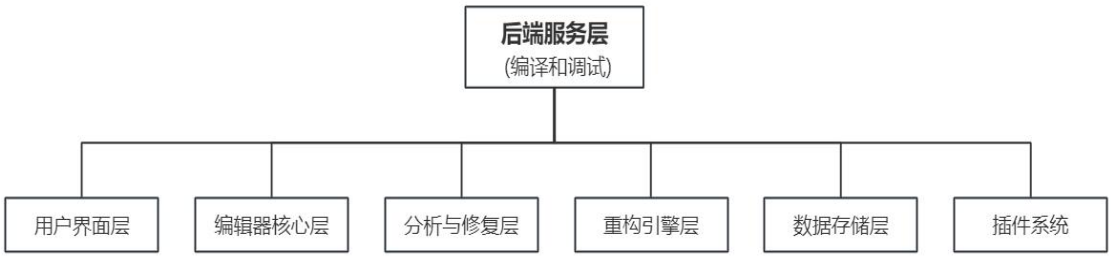

图 5 各层与后端服务层关系图

4.3.5. 各层与数据存储层

- 关系： 数据存储层负责持久化用户的代码、配置和其他重要信息。当用户保存或加载代码时，相应层会与数据存储层通信。
- 通信： 数据存储层的通信通常使用数据库连接和访问协议，如 SQL，或者文件系统的 API 调用。

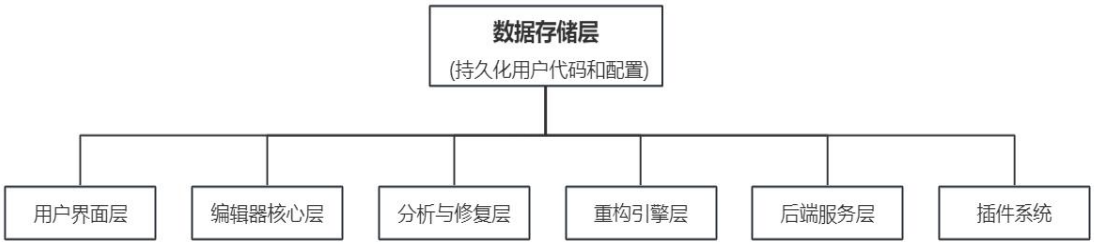

图 6 各层与数据存储层关系图

4.3.6. 用户界面层与插件系统

- 关系： 插件系统允许第三方功能扩展与编辑器的核心功能集成。用户通过用户界面层管理和使用这些插件。
- 通信： 插件系统可能会通过定义良好的接口（例如，使用事件、钩子或者回调函数）与用户界面层和其他系统层交互。

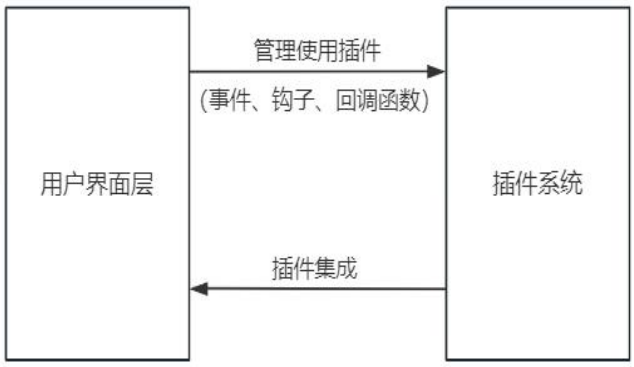

图 7 用户界面层与插件系统关系图

5. 界面设计定义

5.1. 界面布局

表 3 界面布局规定表

界面	规定
编辑器窗口	中心工作区，用户可以在此编写和查看代码。支持标签页，可以打开多个文件进行编辑。
侧边栏	包含多个视图：资源管理器、搜索、源码管理、扩展，可通过图标进行切换。
状态栏	显示有关打开的项目和文件的状态信息，如编程语言、分支、错误和警告。
活动栏	位于侧边栏最左侧，包含用于快速切换主要视图的图标。

面板	包括输出、问题、终端和调试控制台，可以横向或纵向分割，展示项目运行期间的输出或日志信息。

5.2. 交互设计

表 4 交互设计规定表

交互部件	规定
命令面板	允许通过键盘输入执行编辑器命令，极大提升用户的工作效率。
快捷键	提供了一系列快捷键，允许用户快速执行常用功能而无需使用鼠标。
多光标编辑	通过键盘快捷键可以轻松添加多个光标，进行批量编辑操作。

5.3. 可视化设计

表 5 可视化设计规定表

可视化部件	规定
语法高亮	根据不同的语言语法提供色彩编码，使代码更易于阅读和理解。
主题	支持多种配色主题，包括亮色和暗色，用户也可以安装或创建自己的主题。
图标	使用图标来表示文件类型、编辑器操作和状态信息，直观表明其功能。

5.4. 可定制性

1) **用户和工作区设置**：用户可以配置全局设置或特定于工作区的设置，包括快捷键、编辑器偏好、语言环境等。
2) **扩展市场**：通过安装来自 VSCode 市场的扩展，用户可以根据需要添加新功能或增强已有功能。

5.5. 功能性设计

表 6 功能性设计规定表

功能性部件	规定

版本控制集成	内置版本控制功能，如 Git 支持，使得代码同步和历史追踪更加便捷。
调试器	强大的调试工具，支持断点、步进、变量检查和日志输出。
集成终端	内嵌命令行终端，可以在不离开编辑器的情况下运行命令和脚本。

5.6. 辅助工具

1) **代码自动完成**：智能提示和代码自动完成功能，帮助用户快速编写代码。
2) **代码片段**：用户可以快速插入常用代码段，提高编码效率。
3) **代码折叠**：允许用户折叠代码区块，方便查看和编辑大型文件。

6. 接口定义

6.1. 客户端接口定义

6.1.1. 人机交互接口

1）扩展激活（Extension Activation）：

- 接口：export function activate(context: ExtensionContext)
- 功能：初始化语言服务器客户端，配置事件监听器，启动语言服务。
- 触发条件：VSCode 加载扩展时自动触发。

2）语言服务器客户端创建与配置（Language Client Creation & Configuration）：

- 接口：
  a) LanguageClient
  b) LanguageClientOptions
  c) ServerOptions
  d) TransportKind
- 功能：配置语言服务器的运行和调试选项，文档选择器，同步设置等。
- 触发条件：在 activate 函数中配置。

3）命令注册（Command Registration）：

- 接口：vscode.commands.registerCommand
- 功能：注册自定义命令，例如代码分析命令 extension.runAnalysis。
- 触发条件：用户执行命令或通过快捷键触发。

4）事件订阅（Event Subscription）：

- 接口：context.subscriptions.push
- 功能：管理订阅的事件和命令，确保它们在适当的时候被调用。
- 触发条件：在 activate 函数中订阅。

5）扩展停用（Extension Deactivation）：

- 接口：export function deactivate()
- 功能：清理资源，停止语言服务器客户端。
- 触发条件：VSCode 停用扩展时自动触发。

6） 代码分析执行（Run Analysis Execution）：

- 接口：runAnalysis()
- 功能：执行代码分析，调用外部脚本或程序。
- 触发条件：用户通过命令面板或快捷键触发 extension.runAnalysis 命令。

7） 命令行执行（Command Line Execution）：

- 接口：exec
- 功能：在终端中执行外部批处理脚本 run_analysis.bat。
- 触发条件：runAnalysis 函数中调用。

8） 用户消息提示（User Message Display）：

- 接口：
  a) vscode.window.showErrorMessage
  b) vscode.window.showWarningMessage
  c) vscode.window.showInformationMessage
- 功能：向用户显示错误、警告和信息消息。
- 触发条件：根据执行结果和分析输出向用户展示不同的提示信息。

6.1.2. 系统与外部接口

1） 文件系统监视器（FileSystem Watcher）：

- 接口：workspace.createFileSystemWatcher
- 功能：监视工作区中.sy 文件的变化。
- 触发条件：文件系统发生变化时自动触发。

2) 文件读写操作（File Read/Write Operations）：
   - 接口：fs.readFileSync
   - 功能：读取外部脚本执行结果文件 check_output.txt。
   - 触发条件：runAnalysis 函数中调用，用于获取分析结果。
3) 命令行构建与执行（Command Line Construction & Execution）：
   - 功能：构建并执行包含必要参数的命令行，调用外部分析工具。
   - 触发条件：根据当前编辑的文件和工作区路径动态构建命令行。

6.1.3. 系统内模块之间的接口

1) 语言服务器启动（Language Server Start）：
   - 接口：client.start()
   - 功能：启动语言服务器客户端。
   - 触发条件：在 activate 函数中调用。
2) 编辑器与文档交互（Editor and Document Interaction）：
   - 接口：
     a) vscode.window.activeTextEditor
     b) editor.document
     c) document.fileName
   - 功能：获取当前活动的编辑器和文档对象，以及文档的文件路径。
   - 触发条件：runAnalysis 函数中调用，以确定分析目标。
3) 工作区检测（Workspace Detection）：
   - 接口：vscode.workspace.workspaceFolders
   - 功能：检测是否存在打开的工作区。
   - 触发条件：runAnalysis 函数中调用，以确定工作区上下文。
4) 错误和警告计数（Error and Warning Counting）：
   - 功能：解析外部脚本的输出，计算错误、警告等的数量。
   - 触发条件：在读取并解析 check_output.txt 文件后执行。

6.2. 服务端接口定义

6.2.1. 人机交互接口

1) 悬停提示（Hover）
   - 接口：connection.onHover
   - 功能：当用户将鼠标悬停在代码的某个符号上时，提供相关的信息提示。
   - 触发条件：用户鼠标悬停。
2) 自动完成（Completion）
   - 接口：connection.onCompletion
   - 功能：在用户编码时提供代码自动完成建议。
   - 触发条件：用户键入代码时触发。
   - 补充接口：connection.onCompletionResolve
   - 功能：为自动完成列表中的项目提供更详细的信息。
3) 代码操作（Code Action）
   - 接口：connection.onCodeAction
   - 功能：提供一组可执行的代码操作，如快速修复、重构等。
   - 触发条件：用户请求代码操作或触发自动代码操作。
4) 执行命令（Execute Command）
   - 接口：connection.onExecuteCommand
   - 功能：执行由客户端定义的命令，如自动修复代码问题。
   - 触发条件：用户执行特定命令或通过界面触发。
5) 符号定义（Definition）
   - 接口：connection.onDefinition
   - 功能：提供符号的定义位置，如变量、函数的定义。
   - 触发条件：用户需要查看符号的定义时触发。
6) 函数签名帮助（Signature Help）
   - 接口：connection.onSignatureHelp
   - 功能：在用户输入函数调用时提供参数信息和提示。
   - 触发条件：用户键入函数名后紧跟左括号时触发。
7) 所有引用（References）
   - 接口：connection.onReferences

- 功能：查找并提供文档中所有对某个符号的引用。
- 触发条件：用户需要查找符号的所有引用时触发。

8) 重命名（Rename）

- 接口：connection.onRenameRequest
- 功能：允许用户对代码中的符号进行重命名。
- 触发条件：用户执行重命名操作时触发。

6.2.2. 系统与外部接口

1) 配置变更（Configuration Change）

- 接口：connection.onDidChangeConfiguration
- 功能：响应客户端配置的变化，如用户更改设置。
- 触发条件：客户端配置发生变化。

2) 文件监听（File Watching）

- 接口：connection.onDidChangeWatchedFiles
- 功能：响应文件系统上的文件变化事件。
- 触发条件：监视的文件发生变化。

6.2.3. 系统内模块之间的接口

1) 文档管理（Document Management）

- 功能模块：documents
- 功能：管理文本文档的打开、更改和关闭事件。
- 相关事件：
  a) onDidOpen：文档打开时触发。
  b) onDidChangeContent：文档内容更改时触发。
  c) onDidClose：文档关闭时触发。

2) 诊断信息（Diagnostics）

- 功能模块：connection.languages.diagnostics
- 功能：提供文档的诊断信息，如语法错误、代码警告等。
- 触发条件：文档打开或内容更改后进行诊断分析。

3) 符号表生成（Symbol Table Generation）

- 功能模块：SymbolTableGenerator
- 功能：遍历解析树，生成当前文档的符号表。

4) 语法检查（Syntax Checking）
   - 功能模块：SyntaxChecker
   - 功能：检查文档的语法正确性，并生成错误列表。
5) 函数签名生成（Function Signature Generation）
   - 功能模块：FuncSignGenerator
   - 功能：生成函数签名信息，供 onSignatureHelp 接口使用。
6) 引用记录生成（Reference Record Generation）
   - 功能模块：ReferenceRecordGeneretor
   - 功能：生成文档中符号的引用记录，供 onReferences 接口使用。

7. 模块设计

7.1. IDE 集成模块

7.1.1. 模块功能

IDE 集成模块在 SysY2022E 语言编辑器中扮演着桥梁的角色，它负责将编辑器的核心功能与主流的 IDE（如 VSCode、Eclipse）进行无缝集成。该模块的主要功能是提供插件或扩展的安装与配置，确保编辑器能够在 IDE 环境中作为一个独立的插件或扩展运行，同时保持与 IDE 的交互和数据同步。

7.1.2. 模块对象（组件）

1) 输入/输出：
   - 输入：接收来自 IDE 的命令和参数，如打开文件、保存文件、执行语法检查等。
   - 输出：向 IDE 返回操作结果、错误信息或提示信息。
2) 用户界面：集成到 IDE 的用户界面中，提供与编辑器相关的菜单项、工具栏按钮等。
3) 对象或组件：
   - IDE 接口组件：负责与 IDE 进行通信，实现命令的接收和结果的返回。
   - 插件管理组件：负责插件的安装、更新和卸载。
   - 配置管理组件：负责编辑器的配置管理，如设置语法高亮颜色、自动补全选项等。
   - 对象或组件关系：IDE 接口组件是与其他组件交互的入口点，插件管理组件和配置

管理组件通过 IDE 接口组件与 IDE 进行通信。

7.1.3. 对象（组件）的触发机制

1) IDE 接口组件：
   - 当 IDE 启动时，自动加载并初始化 IDE 接口组件。
   - 当用户通过 IDE 执行编辑器相关命令时，IDE 接口组件接收命令并转发给相应的内部组件处理。
2) 插件管理组件：
   - 当用户通过 IDE 安装或更新编辑器插件时，触发插件管理组件的安装或更新操作。
   - 当用户卸载编辑器插件时，触发插件管理组件的卸载操作。
3) 配置管理组件：
   - 当用户通过 IDE 修改编辑器配置时，触发配置管理组件的配置更新操作。

7.1.4. 对象（组件）的关键算法

1) 插件安装与更新算法：下载插件或更新包，验证其完整性和兼容性，然后解压到指定目录，并在 IDE 中注册该插件。
2) 配置管理算法：读取用户的配置修改请求，验证配置的合法性和有效性，然后更新配置文件并通知相关组件重新加载配置。

7.2. 语法高亮和悬浮提示模块

7.2.1. 模块功能

语法高亮和悬浮提示模块是 SysY2022E 语言编辑器的重要组成部分，它负责实现代码的语法高亮显示和鼠标悬浮时的提示功能。该模块通过解析代码语法，为不同的语法元素着色，并在用户将鼠标悬浮在标识符上时显示相应的信息或提示。

7.2.2. 模块对象（组件）

1) 输入/输出：
   - 输入：接收用户打开的 SysY2022E 源代码文件。
   - 输出：高亮显示的源代码和悬浮提示信息。
2) 用户界面：无直接用户界面，但通过高亮显示和悬浮提示间接与用户交互。
3) 对象或组件：

- 语法解析器：负责解析 SysY2022E 源代码的语法结构。
- 高亮渲染器：根据语法解析结果，为不同的语法元素着色。
- 悬浮提示生成器：根据用户鼠标悬浮的标识符，生成相应的提示信息。
- 对象或组件关系：语法解析器解析源代码后，将解析结果传递给高亮渲染器和悬浮提示生成器进行处理。

7.2.3. 对象（组件）的触发机制

1) 语法解析器：
   - 当用户打开一个新的 SysY2022E 源代码文件时，触发语法解析器的解析操作。
   - 当用户修改源代码时，实时触发语法解析器的增量解析操作。
2) 高亮渲染器：
   - 当语法解析器完成解析后，触发高亮渲染器的着色操作。
   - 当用户滚动代码视图或切换选项卡时，触发高亮渲染器的重新着色操作。
3) 悬浮提示生成器：
   - 当用户将鼠标悬浮在标识符上时，触发悬浮提示生成器的提示信息生成操作。

7.2.4. 对象（组件）的关键算法

1) 语法解析算法：基于 SysY2022E 语言的语法规则，采用递归下降或 LL(1)等算法进行语法解析。
2) 高亮渲染算法：根据语法元素的类型和属性，选择合适的颜色和样式进行着色渲染。
3) 悬浮提示生成算法：根据鼠标悬浮的标识符类型（如变量名、函数名等），查询相应的符号表或类型信息，生成对应的提示信息。

7.3. 语法检查与错误提示模块

7.3.1. 模块功能

语法检查与错误提示模块负责检查 SysY2022E 源代码中的语法错误，并在发现错误时提供错误位置和相应的错误信息提示。该模块通过静态分析源代码来发现潜在的语法问题，帮助用户及时纠正错误并提高代码质量。

7.3.2. 模块对象（组件）

1) 输入/输出：

- 输入：接收用户打开的 SysY2022E 源代码文件。
- 输出：语法错误位置和相应的错误信息提示。

2) 用户界面：在源代码编辑器中标记语法错误位置，并显示错误信息提示框或下划线等视觉反馈。

3) 对象或组件：
   - 语法检查器：负责静态分析源代码并发现语法错误。
   - 错误提示生成器：根据语法错误类型和位置生成相应的错误信息提示。
   - 对象或组件关系：语法检查器将发现的语法错误传递给错误提示生成器进行处理和显示。

7.3.3. 对象（组件）的触发机制

1) 语法检查器：
   - 当用户打开一个新的 SysY2022E 源代码文件时，触发语法检查器的全面检查操作。
   - 当用户修改源代码时，实时触发语法检查器的增量检查操作。

2) 错误提示生成器：
   - 当语法检查器发现语法错误时，触发错误提示生成器的错误信息生成和显示操作。

7.3.4. 对象（组件）的关键算法

1) 语法检查算法：基于 SysY2022E 语言的语法规则，采用词法分析和语法分析相结合的方法进行源代码的静态检查。

2) 错误提示生成算法：根据语法错误的类型和位置，生成具有明确指向性和描述性的错误信息提示文本。

7.4. 静态语义检查模块

7.4.1. 模块功能

静态语义检查模块负责在编译时检查 SysY2022E 源代码中的静态语义错误，如类型不匹配、未定义的变量等。该模块通过静态分析源代码的语义信息来发现潜在的错误，提高代码的健壮性和可读性。

7.4.2. 模块对象（组件）

1) 输入/输出：

- 输入：经过语法检查的 SysY2022E 源代码文件。
- 输出：静态语义错误位置和相应的错误信息提示。

2) 用户界面：在源代码编辑器中标记静态语义错误位置，并提供错误信息提示或警告。

3) 对象或组件：

- 语义分析器：负责分析源代码的语义信息并发现静态语义错误。
- 错误报告生成器：根据静态语义错误类型和位置生成相应的错误信息提示或警告。
- 对象或组件关系：语义分析器将发现的静态语义错误传递给错误报告生成器进行处理和显示。

7.4.3. 对象（组件）的触发机制

1) 语义分析器：

- 当用户完成源代码的语法检查并请求进行静态语义检查时，触发语义分析器的全面分析操作。
- 当用户修改源代码并重新进行静态语义检查时，触发语义分析器的增量分析操作。

2) 错误报告生成器：

- 当语义分析器发现静态语义错误时，触发错误报告生成器的错误信息或警告生成和显示操作。

7.4.4. 对象（组件）的关键算法

1) 语义分析算法：基于 SysY2022E 语言的语义规则和数据流分析技术，对源代码进行静态语义检查。包括类型推断、变量和函数定义检查等步骤。

2) 错误报告生成算法：根据静态语义错误的类型和位置，生成具有明确描述性和指向性的错误信息提示或警告文本。

7.5. 修复建议模块

7.5.1. 模块功能

修复建议模块负责为语法错误和静态语义错误提供自动修复建议。该模块通过分析错误的类型和上下文信息，给出可能的修复方案供用户选择和执行。这有助于提高开发效率和代码质量。

7.5.2. 模块对象（组件）

1) 输入/输出：
   - 输入：语法错误和静态语义错误的位置及类型
   - 输出：针对特定错误的修复建议或修复代码片段
2) 用户界面：在错误提示旁边或下方显示修复建议，用户可以直接点击应用或手动修改代码。
3) 对象或组件：
   - 错误分析器：深入解析具体的错误类型和上下文。
   - 修复建议生成器：基于错误分析的结果，生成可能的修复建议。
   - 修复执行器：如果用户选择了自动修复，此组件将负责将修复建议应用到代码中。
   - 对象或组件关系：错误分析器首先分析错误，然后将结果传递给修复建议生成器。修复建议生成器根据这些信息产生修复建议，用户可以选择应用这些建议，此时修复执行器会介入执行。

7.5.3. 对象（组件）的触发机制

1) 错误分析器：当语法检查与错误提示模块或静态语义检查模块发现错误时，触发错误分析器进行深入分析。
2) 修复建议生成器：当错误分析器完成错误分析后，触发修复建议生成器生成相应的修复建议。
3) 修复执行器：当用户从修复建议中选择一个并点击自动修复时，触发修复执行器进行代码修复。

7.5.4. 对象（组件）的关键算法

1) 错误分析算法：详细解析错误类型和上下文，可能涉及控制流、数据流和类型信息的分析。
2) 修复建议生成算法：基于常见的编程模式和错误修复策略，结合错误的上下文，生成可能的修复方案。
3) 修复执行算法：对选定的修复建议进行代码转换和插入，确保修复后的代码仍然保持语法和语义的正确性。

7.6. 代码格式化与排版模块

7.6.1. 模块功能

代码格式化与排版模块负责自动调整代码的格式，使其符合一定的编程风格和排版规则。这有助于提高代码的可读性和维护性。

7.6.2. 模块对象（组件）

1) 输入/输出：
   - 输入：用户编写的 SysY2022E 源代码。
   - 输出：格式化后的源代码。
2) 用户界面：提供格式化选项供用户选择，如缩进风格、空格使用等。
3) 对象或组件：
   - 代码解析器：负责解析源代码的结构。
   - 格式化规则引擎：根据用户选择的格式化选项，确定如何调整代码格式。
   - 代码重写器：根据格式化规则引擎的输出，重写源代码以实现格式化。
   - 对象或组件关系：代码解析器首先解析源代码，然后格式化规则引擎确定如何格式化，最后代码重写器根据规则重写源代码。

7.6.3. 对象（组件）的触发机制

1) 代码解析器：当用户请求代码格式化时，触发代码解析器的解析操作。
2) 格式化规则引擎：当代码解析器完成解析后，触发格式化规则引擎确定格式化规则。
3) 代码重写器：当格式化规则引擎确定规则后，触发代码重写器进行源代码的重写操作。

7.6.4. 对象（组件）的关键算法

1) 代码解析算法：基于 SysY2022E 语言的语法规则进行源代码的解析。
2) 格式化规则确定算法：根据用户选择的格式化选项和编程风格指南，确定具体的格式化规则。
3) 代码重写算法：根据格式化规则，对源代码进行重写以实现预期的格式化效果。这可能涉及添加、删除或修改空格、换行符、缩进等。

8. 故障检测和处理机制

8.1. 故障检测触发机制

1) **IDE 集成故障检测**

- 插件/扩展兼容性检测：在与主流 IDE（如 VSCode、Eclipse）集成时，编辑器应定期检查插件或扩展的兼容性。若检测到版本不匹配或接口变更，将触发故障处理流程，以确保与 IDE 的无缝集成。
- IDE 接口调用监控：监控与 IDE 之间的接口调用情况，一旦出现调用失败、超时或异常返回，将触发故障检测机制。

2) **语法高亮和悬浮提示故障检测**

- 语法高亮规则验证：定期验证语法高亮规则的准确性，若检测到规则错误或更新不及时，将触发故障处理。
- 悬浮提示数据加载：监控悬浮提示功能的数据加载过程，若加载失败或数据不准确，将触发故障检测。

3) **语法检查与错误提示故障检测**

- 语法解析器状态监控：监控语法解析器的运行状态，确保其正常工作。若解析器崩溃或无响应，将触发故障处理流程。
- 错误提示准确性验证：定期验证错误提示的准确性，若检测到误报或漏报，将触发故障处理。

4) **静态语义检查故障检测**

- 类型和作用域检查规则验证：验证静态语义检查的规则是否准确，若检测到规则错误或更新不及时，将触发故障处理。
- 检查过程监控：监控静态语义检查过程的执行情况，若执行时间过长或出现异常，将触发故障检测机制。

5) **修复建议和程序错误检查故障检测**

- 修复建议准确性验证：验证提供的修复建议是否准确有效，若检测到无效或错误的建议，将触发故障处理流程。
- 程序错误检查规则更新：定期检查程序错误检查规则的更新情况，确保规则的时效性和准确性。若检测到规则未及时更新或存在错误，将触发故障处理。

6) **代码重构故障检测**

- 重构操作验证：在每次代码重构操作后，验证重构结果的正确性。若检测到重构

失败或引入新的问题，将触发故障处理流程。

- 重构引擎状态监控：监控代码重构引擎的运行状态，确保其正常工作。若引擎崩溃或无响应，将触发故障检测机制。

8.2. 故障处理机制

针对上述故障检测触发机制中可能发现的问题，我们设计了以下详尽的故障处理机制

1) **IDE 集成故障处理**

- 自动更新插件/扩展：当检测到插件或扩展不兼容时，尝试自动更新至最新版本以解决问题。若更新失败，则提示用户手动更新。
- 重新建立 IDE 连接：当与 IDE 的接口调用出现问题时，尝试重新建立连接。若多次尝试后仍然失败，则提示用户检查 IDE 设置或网络环境。

2) **语法高亮和悬浮提示故障处理**

- 修复或更新语法高亮规则：当检测到语法高亮规则错误时，尝试自动修复或提示用户更新规则。
- 重新加载悬浮提示数据：当悬浮提示数据加载失败时，尝试重新加载数据。若多次加载仍然失败，则提示用户检查数据源或网络环境。

3) **语法检查与错误提示故障处理**

- 重启语法解析器：当语法解析器崩溃或无响应时，尝试自动重启解析器。若重启后仍然无法正常工作，则提示用户检查系统资源或联系技术支持。
- 优化错误提示算法：当检测到错误提示存在误报或漏报时，尝试优化算法并提高准确性。同时，收集用户反馈以持续改进错误提示功能。

4) **静态语义检查故障处理**

- 更新静态语义检查规则：当检测到静态语义检查规则错误时，尝试自动更新规则。若更新失败，则提示用户手动更新或联系技术支持。
- 优化检查过程性能：当静态语义检查过程执行时间过长或出现异常时，尝试优化检查算法并提高性能。同时，监控系统资源使用情况以确保检查过程的顺利进行。

5) **修复建议和程序错误检查故障处理**

- 验证并修正修复建议：当检测到无效的修复建议时，尝试验证并修正建议。若无法自动修正，则提示用户谨慎选择修复方案或联系技术支持。
- 更新程序错误检查规则：定期检查并更新程序错误检查规则以确保其时效性和准确性。同时收集用户反馈以持续改进错误检查功能。

6） **代码重构故障处理**

- 回滚重构操作：当检测到重构失败或引入新的问题时，尝试自动回滚重构操作至原始状态。若回滚失败，则提示用户手动恢复或联系技术支持。
- 重启重构引擎：当重构引擎崩溃或无响应时，尝试自动重启引擎。若重启后仍然无法正常工作，则提示用户检查系统资源或联系技术支持。同时收集用户反馈以持续改进重构引擎的稳定性和性能。

9. 系统开发平台

9.1. 硬件平台

服务器硬件：对于本 SysY2022E 语言编辑器开发项目，用于后端服务层和数据存储层的服务器并不需要较高的计算能力和存储容量，因此对服务器硬件并无较高要求。

客户端硬件:用户界面层和编辑器核心层可以在各种客户端设备上运行,包括个人电脑、笔记本电脑、甚至是平板电脑和智能手机。这些设备需要足够的处理能力和内存来运行复杂的 IDE 功能。

9.1.1. 支持的平台

该插件支持以下主要硬件平台：

- Windows
- macOS
- Linux

9.1.2. 最低硬件要求

表 7 最低硬件要求表

	Windows	macOS	Linux
操作系统	Windows 10 及以上版本	macOS 10.14 Mojave 及以上版本	64-bit 版本的 Ubuntu 16.04 及以上 65-Debian 9 及以上 66-Fedora 30 及以上
处理器	1.6 GHz 或更高，双核	Intel Core i5 或更高，	1.6 GHz 或更高，双核

		或 Apple M1 芯片	
内存	4 GB RAM 或更高		
硬盘空间	可用空间>=200 MB		
显示	分辨率>=1366 x 768		
网络	互联网连接用于插件安装和更新		

9.1.3. 推荐硬件配置

表 8 推荐硬件配置表

	Windows	macOS	Linux
操作系统	Windows 11	macOS 12 Monterey	最新版本的 Ubuntu、Debian 或 Fedora
处理器	2.4 GHz 或更高，四核	Apple M1 芯片或更高	2.4 GHz 或更高，四核
内存	8 GB RAM 或更高		
硬盘空间	SSD，500 GB 可用空间		
显示	1920 x 1080 分辨率或更高		
网络	高速互联网连接		

9.1.4. 未来的硬件支持

随着硬件技术的发展，本插件将定期更新，以支持新的硬件平台和配置。定期评估和更新硬件要求，以确保插件的高效运行。

9.2. 操作系统

服务器操作系统：考虑到本项目的使用场景，服务器实际上是配置在客户端主机上的另外一个独立的进程，因此对于其对操作系统的要求，应具有可迁移性，即适用于 Windows、macOS 和 Linux。

客户端操作系统：用户界面层和编辑器核心层可以在多种操作系统上运行，包括 Windows、macOS 和 Linux，以满足不同用户的需求。

9.2.1. 支持的操作系统平台

该插件支持以下主要操作系统平台：

Windows

macOS

Linux

9.2.2. 最低操作系统要求

表 9 最低操作系统要求表

	Windows	macOS	Linux
操作系统版本	Windows 10 版本 1607 及以上	macOS 10.14 Mojave 及以上版本	64-bit 版本 Ubuntu: 16.04 及以上版本 Debian: 9 及以上版本 Fedora: 30 及以上版本 CentOS: 7 及以上版本
架构	x64 和 x86	x64 和 ARM64 (Apple Silicon)	x64 和 ARM64
依赖软件	.NET Framework 4.5.2 或更高版本	————	————

9.2.3. 推荐操作系统配置

表 10 推荐操作系统配置表

	Windows	macOS	Linux
操作系统版本	Windows 11	macOS 12 Monterey	最新版本 Ubuntu: 最新 LTS 版本 Debian: 最新稳定版本 Fedora: 最新稳定版本 CentOS: 最新稳定版本
架构	x64	ARM64 (Apple Silicon)	x64 和 ARM64
依赖软件	最新版本的 .NET Framework	————	————

9.2.4. 安装和配置说明

1) Windows

- 下载并安装最新版本的 Visual Studio Code。
- 确保系统已安装 .NET Framework 4.5.2 或更高版本。
- 在 Visual Studio Code 中通过扩展市场安装插件。

2) macOS

- 下载并安装最新版本的 Visual Studio Code。
- 确保 macOS 版本在 10.14 及以上。
- 在 Visual Studio Code 中通过扩展市场安装插件。

3) Linux

- 下载并安装最新版本的 Visual Studio Code。
- 确保操作系统为 64-bit 版本，版本满足最低要求。
- 在 Visual Studio Code 中通过扩展市场安装插件。

9.2.5. 未来的操作系统支持

随着操作系统技术的发展，本插件将定期更新，以支持新的操作系统版本和架构。定期评估和更新系统要求，以确保插件的高效运行。

9.3. 开发工具

- 集成开发环境（IDE）：用于开发系统的 IDE，Visual Studio Code，IDE 提供了代码编辑、调试、版本控制等功能。
- 版本控制系统：如 Git，用于管理代码版本和协作开发。
- 容器化和虚拟化工具：如 Docker 和 Kubernetes，用于部署和管理后端服务。
- 数据库管理系统：如 MySQL、PostgreSQL 或 MongoDB，用于数据存储层的实现。
- API 开发和文档工具：如 Swagger 或 Postman，用于设计和测试 API 接口。

9.3.1. 开发环境

为了保证插件开发的高效和顺利进行，推荐使用以下开发环境：

- 操作系统:开发人员可以选择适合自己的操作系统，包括 Windows、macOS 和 Linux；
- 文本编辑器/集成开发环境 (IDE):Visual Studio Code；
- 版本:最新稳定版；
- 扩展:一些常用的扩展如 ESLint、Prettier、TypeScript、npm Intellisense 等。

9.3.2. 编程语言和工具

1) TypeScript
   - 版本:最新稳定版;
   - 安装:可以通过 npm 安装 TypeScript。
2) Node.js 和 npm
   - 版本: Node.js 最新稳定版和 npm 最新版本;
   - 安装: 可以从 Node.js 官方网站下载和安装 Node.js 和 npm。
3) VS Code Extension API
   - 文档：参考官方文档。
4) 语言服务器协议 LSP
   - 文档：参考 LSP 官方文档。

9.3.3. 开发工具和库

1) Yeoman 和 Generator Code
   - 用于生成 VS Code 插件的脚手架;
   - 安装: 通过 npm 安装 Yeoman 和 Generator Code。
2) ANTLR4
   - 用于语法分析和语义分析;
   - 安装: 参考 ANTLR4 官方网站 下载和安装 ANTLR4;
   - VS Code 扩展: ANTLR4 Grammar Syntax Support。

9.3.4. 项目结构

```
your-vscode-extension/
├─── .vscode/
|    ├─── extensions.json
|    └─── settings.json
├─── src/
|    ├─── extension.ts
|    ├─── languageServer.ts
|    ├─── diagnosticProvider.ts
|    ├─── hoverProvider.ts
```

```
│   ├── ...
├── syntaxes/
│   └── sysY2022.tmLanguage.json
├── package.json
├── tsconfig.json
├── .eslintrc.json
├── .prettierrc
└── README.md
```

9.3.5. 版本控制

- 工具:Git
- 平台:GitHub、GitLab 或 Bitbucket
- 规范:使用 Git 分支模型进行开发，例如 Git Flow 或 GitHub Flow
- 文件:确保 .gitignore 文件包含不需要版本控制的文件和文件夹，例如 node_modules/

9.3.6. 文档和资源

- API 文档: 定期查阅并参考 VS Code API 文档 和 LSP 官方文档
- 教程和指南:参考 VS Code 扩展开发文档 以及各种在线教程和博客
- 社区和支持:加入相关的开发者社区,例如Stack Overflow和GitHub Discussions，获取帮助和支持