SysY2022E 扩展语言编辑器

SysY2022E Language Support

编码规范和标准说明书

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. 引言

1.1. 编写目的

编码规范与标准文档的编写目的在于为开发团队提供一套统一的编码标准和最佳实践，确保代码的可读性、一致性和可维护性。通过制定和遵循编码规范，可以提高代码质量、减少错误、增强团队协作效率，并为代码的后续维护和扩展奠定基础。

1) 确保代码一致性：提供统一的编码风格和格式，使所有开发人员编写的代码在风格上保持一致，便于理解和审查。
2) 提高代码可读性：定义清晰的命名约定、注释规范和代码结构，使代码易于阅读和理解，减少沟通成本和理解障碍。
3) 增强代码可维护性：提供编写高质量、易维护代码的指导原则，帮助开发人员避免常见的编程错误和陷阱，提高代码的稳定性和可维护性。
4) 促进团队协作：通过制定统一的编码标准，减少开发人员之间的风格差异，促进团队协作和代码共享，提高开发效率。
5) 支持代码复用和扩展：定义模块化、可扩展的编码实践，促进代码的复用和扩展，降低重复劳动和开发成本。
6) 规范化开发流程：结合软件工程的最佳实践，规范化开发流程中的编码环节，确保开发过程的标准化和规范化。

1.2. 范围

1.2.1.覆盖范围

1）命名约定：

- 变量、函数、类、接口、文件、目录等的命名规则。
- 常量、全局变量、局部变量等不同作用域下的命名规范。

2）代码格式：

- 缩进与空格：统一的缩进规则（如使用空格或制表符）和每层缩进的空格数 量。
- 换行与换页：代码行长度限制，代码块、函数、类之间的换行规则。
- 括号与注释：括号的使用位置和风格，单行注释和多行注释的书写规范。

3）代码结构：

- 函数和类的组织方式。
- 模块的分层与分包策略，项目目录结构。
- 文件头部注释和版权声明。

4）编程实践：

- 编写可读性高的代码，使用自解释代码和注释的最佳实践。
- 错误处理与异常捕获的标准做法。
- 单元测试和代码覆盖率要求。

5）代码审查：

- 代码审查的流程和规范。
- 代码提交和合并的最佳实践。
- 代码审查工具的使用方法。

6）版本控制：

- 版本控制系统的使用规范（如 Git）。
- 分支管理策略（如 GitFlow、GitHub Flow）。
- 提交信息的书写规范。

1.2.2. 角色与解决的问题

1）提升代码质量：

通过统一的编码规范和最佳实践，减少代码中的错误和漏洞，提高代码的健壮性和稳定性。

2）提高开发效率：

规范化的代码风格和结构使开发人员在阅读和理解代码时更为高效，减少沟通成本，促进团队协作。

3）促进代码维护：

规范的代码格式和清晰的注释使代码更容易维护和扩展，降低后期维护成本。

4）确保一致性：

统一的命名约定和代码格式确保整个项目代码的一致性,提高代码的可读性和可维护性。

5）规范开发流程：

结合软件工程的最佳实践，规范化开发流程中的编码环节，确保开发过程的标准化和规范化。

6）支持持续集成与交付：

通过规范的代码审查和版本控制流程，支持持续集成和持续交付，提高开发和交付的效率和质量。

7）增强团队协作：

提供统一的编码标准，减少开发人员之间的风格差异，促进团队成员之间的代码共享和协作。

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] Google JavaScript Style Guide

提供了 JavaScript 代码的命名、格式、注释、代码组织、错误处理等方面的详细规范和最佳实践。

[2] The Clean Code Book by Robert C. Martin

提供了编写高质量、可维护代码的原则和实践，是代码规范和质量保证的重要参考资料。

[3] The Pragmatic Programmer by Andrew Hunt and David Thomas

涉及软件开发中的最佳实践和编码技巧，提供了编写高质量代码的实际建议。

[4] Effective Java by Joshua Bloch

提供了 Java 编程中的最佳实践和建议，有助于编写高质量的 Java 代码。

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. 程序书写格式规范

严格要求书写格式是为了使程序整齐美观、易于阅读、风格统一，程序员必须对规范书写的必要性要有明确认识。

2.1. 代码风格

2.1.1.代码文件结构

表 3 代码文件结构

```
/project-root
├── /src                    // 源代码
|   ├── /components         // 组件
|   |   ├── ComponentA.ts
|   |   └── ComponentB.ts
|   ├── /utils              // 工具函数
|   |   └── helper.ts
|   ├── /models             // 数据模型
|   |   └── model.ts
|   ├── /services           // 服务层
|   |   └── apiService.ts
|   ├── /styles             // 样式文件
|   |   └── main.css
|   ├── app.ts              // 入口文件
|   └── index.html          // HTML 模板
├── /test                   // 测试文件
|   ├── /unit               // 单元测试
|   ├── /integration        // 集成测试
|   └── /e2e                // 端到端测试
├── /scripts                // 脚本文件
|   ├── build.sh
```

```
|   ├── deploy.sh
|   └── test.sh
├── package.json
├── tsconfig.json
└── README.md
```

2.1.2.导入/包含语句顺序

1）导入顺序

- 核心模块（如 fs, path）
- 第三方库（如 react, lodash
- 自定义模块（如 @/components/ComponentA）

2）导入语句格式

表 4 导入语句格式示例

```
// 核心模块
import fs from 'fs';
// 第三方库
import React from 'react';
import _ from 'lodash';
// 自定义模块
import ComponentA from '@/components/ComponentA';
import { helper } from '@/utils/helper';
```

2.1.3.类和函数的排列顺序

1）类的排列顺序

表 5 类的排列顺序示例

```
class MyClass {
  // 私有属性
  private property1: string;
  private property2: number;
  // 公有属性
```

```
  public property3: boolean;
  // 构造函数
  constructor(property1: string, property2: number) {
    this.property1 = property1;
    this.property2 = property2;
    this.property3 = false;
  }
  // 公共方法
  public method1(): void {
    // 方法体
  }
  public method2(): number {
    // 方法体
    return 0;
  }
  // 私有方法
  private method3(): void {
    // 方法体
  }
}
```

2）函数的排列顺序

- 工具函数
- 回调函数
- 事件处理函数
- 渲染函数

表 6 函数排列顺序示例

```
// 工具函数
function utilFunction1(): void {
  // 函数体
}
function utilFunction2(): number {
```

```
  // 函数体
  return 0;
}
// 回调函数
function callbackFunction(err: Error, data: any): void {
  if (err) {
    // 错误处理
  } else {
    // 处理数据
  }
}
// 事件处理函数
function handleClick(event: MouseEvent): void {
  // 处理点击事件
}
// 渲染函数
function renderComponent(): JSX.Element {
  return (
    <div>
      {/* JSX 元素 */}
    </div>
  );
}
```

2.2. 代码格式

2.2.1. 缩进

1）使用 2 个空格进行缩进，不使用制表符。

2）所有代码块（函数、类、对象字面量等）均应缩进。

表 7 缩进示例

```
function exampleFunction() {
  const exampleObject = {
    property1: 'value1',
    property2: 'value2',
  };
  if (exampleObject.property1) {
    console.log('Property1 exists');
  }
}
```

2.2.2. 行宽

每行不超过 80 个字符。若需要换行，请使用合理的换行规则。

2.2.3. 空格使用规则

（1）操作符前后各保留一个空格。

（2）关键字（如 if, for, while）后保留一个空格。

（3）函数名和括号之间不留空格，函数参数之间留一个空格。

表 8 空格使用规则示例

```
const sum = a + b;
if (a > b) {
  console.log('a is greater than b');
}
function exampleFunction(arg1: string, arg2: number): void {
  // 函数体
}
```

2.2.4. 注释

1）使用单行注释 (//) 和多行注释 (/* ... */)。

2）注释应尽量简洁明了，说明代码的作用和逻辑。

表 9 注释示例

```
// 这是单行注释
/*
 * 这是多行注释
 * 可以用于详细解释代码逻辑
 */
// 函数用于计算两个数的和
function add(a: number, b: number): number {
  return a + b; // 返回相加结果
}
```

2.2.5. 大括号位置

1）大括号 { 放在控制语句和函数定义的同一行。

2）结束大括号 } 单独占一行。

表 10 大括号位置示例

```
if (a > b) {
  console.log('a is greater than b');
}


function exampleFunction() {
  // 函数体
}
```

2.2.6. 空行使用

1）文件开头和结尾不应有多余的空行。

2）不同代码块之间使用空行分隔，保持代码清晰。

3）注释与代码之间使用空行分隔。

表 11 空行使用示例

```
class ExampleClass {
  private property: string;
  constructor(property: string) {
```

```
    this.property = property;
  }
  // 获取属性值
  public getProperty(): string {
    return this.property;
  }
  // 设置属性值
  public setProperty(property: string): void {
    this.property = property;
  }
}
// 实例化对象
const example = new ExampleClass('value');
```

3. 命名规范

命名规范使程序更易读，从而更易于理解，本节主要描述包、类、接口、方法、各类变量和常量等如何进行规范的命名。

3.1. 文件内容

3.1.1 包名

1）使用小写字母，单词间用点分隔。

2）包名应具有描述性，通常为域名的反写形式。

表 12 包名示例

```
// 示例
com.example.myproject.utils
```

3.1.2 类名

1）使用帕斯卡命名法（PascalCase），每个单词的首字母大写。

2）类名应具有描述性，通常为名词。

表 13 类名示例

```
// 示例
class UserManager {
  // 类体
}
```

3.1.3. 接口名

1）使用帕斯卡命名法（PascalCase），每个单词的首字母大写。

2）接口名通常以 I 开头，表示这是一个接口。

表 14 接口名示例

```
// 示例
interface IUser {
  // 接口体
}
```

3.1.4. 方法名

1）使用骆驼命名法（camelCase），第一个单词首字母小写，后续单词首字母大写。

2）方法名应具有描述性，通常为动词或动词短语。

表 15 方法名示例

```
// 示例
class UserManager {
  createUser() {
    // 方法体
  }
}
```

3.1.5. 变量名

1）使用骆驼命名法（camelCase），第一个单词首字母小写，后续单词首字母大写。

2）变量名应具有描述性，尽量避免使用单字符变量名。

表 16 变量名示例

```
// 示例
let userName: string;
let userAge: number;
```

3.1.6. 常量名

1）使用全大写字母，单词间用下划线分隔。

2）常量名应具有描述性，通常为名词或名词短语。

表 17 常量名示例

```
// 示例
const MAX_USERS = 100;
const API_URL = "https://api.example.com";
```

3.1.7. 枚举名

1）使用帕斯卡命名法（PascalCase），每个单词的首字母大写。

2）枚举名应具有描述性，通常为名词。

表 18 枚举名示例

```
// 示例
enum UserRole {
  Admin,
  User,
  Guest
}
```

3.1.8. 属性名

1）类属性名使用骆驼命名法（camelCase），第一个单词首字母小写，后续单词首字母大写。

2）私有属性名应以 _ 开头，表示这是一个私有属性。

表 19 属性名示例

```
// 示例
class User {
  private _userName: string;
  private _userAge: number;
}
```

3.2. 文件命名

3.2.1. 文件名

1）文件名应与类名或主要内容相匹配，使用帕斯卡命名法（PascalCase）。

2）扩展名根据文件类型使用 .ts, .js 等。

表 20 文件名示例

```
// 示例
UserManager.ts
IUser.ts
```

3.2.2. 命名空间

1）使用帕斯卡命名法（PascalCase），每个单词的首字母大写。

2）命名空间应具有描述性，通常为名词或名词短语。

表 21 命名空间示例

```
// 示例
namespace MyProject {
  export class UserManager {
    // 类体
  }
}
```

4. 声明规范

程序中定义的数据类型，在计算机中都要为其开辟一定数量的存储单元，为了避免造成资源的不必要浪费，所以按需定义数据的类型，声明包、类以及接口。

4.1. 变量初始化

1）在声明变量时，应该立即对其进行初始化。

2）多个变量声明应分行列出，每个变量独占一行。

表 22 变量初始化示例

```
let variable1: number = 10;
let variable2: string = "Hello";
```

4.2. 包的声明

1）包的声明应该放在文件的最顶部。

2）使用 namespace 或 module 关键字声明包。

3）包名应与文件路径相匹配。

表 23 包的声明示例

```
namespace MyPackage {
  // 包内内容
}
```

4.3. 类和接口的声明

1）使用 class 关键字声明类，使用 interface 关键字声明接口。

2）类和接口的声明应该遵循帕斯卡命名法（PascalCase）。

表 24 类和接口的声明示例

```
class MyClass {
```

```
  // 类的内容
}
interface MyInterface {
  // 接口的内容
}
```

4.4. 构造函数

1）类的构造函数应该在类的声明之后立即定义。

2）构造函数的参数应该按照顺序排列，并且类型应该明确指定。

表 25 构造函数示例

```
class MyClass {
  constructor(private _name: string, private _age: number) {
    // 构造函数内容
  }
}
```

4.5. 方法和属性声明

1）类的方法和属性声明应该遵循骆驼命名法（camelCase）。

2）方法和属性的访问修饰符应该明确指定。

表 26 方法和属性声明示例

```
class MyClass {
  private _propertyName: string;

  public getPropertyName(): string {
    return this._propertyName;
  }
}
```

5. 语句规范

规范的语句可以改善程序的可读性，可以让程序员尽快而彻底地理解新的代码。本节主要对简单语句和复合语句的格式原则进行描述。

5.1. 简单语句

1）简单语句应该只有一行，除非在可读性和清晰性上有必要分成多行。

2）在分成多行时，应该使用适当的缩进和换行，保持代码的可读性。

表 27 简单语句示例

```
// 单行简单语句
let x: number = 10;
// 多行简单语句
let y: number =
  20;
```

5.2. 复合语句

1）复合语句应该使用大括号 {} 包围。

2）复合语句内的语句应该缩进，通常是使用两个空格或四个空格进行缩进。

表 28 复合语句示例

```
// 复合语句示例
if (condition) {
  // 缩进的语句
}
```

5.3. 控制结构的格式

1）控制结构（如 if、for、while 等）的格式应该清晰易读，使用适当的缩进和换行。

2）控制结构内的代码应该按照逻辑结构进行排列，保持代码的清晰度。

表 29 控制结构的格式示例

```
// 控制结构示例
if (condition) {
  // 缩进的语句
} else {
  // 缩进的语句
}
```

5.4. 空格和换行

1）在适当的地方使用空格和换行可以提高代码的可读性。

2）在运算符、逗号等符号周围使用空格可以使代码更易于阅读。

表 30 空格和换行示例

```
// 使用空格和换行的示例
let result = a + b;

if (condition) {
  // 缩进的语句
}
```

6. 注释规范

本节主要对注释的方法和风格进行规范和说明。如块注释、单行注释、尾端注释和行末注释。

6.1. 块注释

1）块注释适用于多行注释，通常用于对代码段进行说明。

2）块注释应该以 /* 开始，以 */ 结束。

3）块注释的内容应该清晰明了，描述性强。

表 31 块注释示例

```
/*
  这是一个块注释的示例。
  块注释可以跨越多行。
*/
```

6.2. 单行注释

1）单行注释适用于对单行代码或代码片段进行说明。

2）单行注释应该以 // 开始，后跟注释内容。

表 32 单行注释示例

```
// 这是一个单行注释的示例。
```

6.3. 尾端注释

1）尾端注释通常位于代码行末尾，用于对代码行的说明。

2）尾端注释应该与代码之间有足够的空格隔开，以增加可读性。

表 33 尾端注释示例

```
let x: number = 10;  // 这是一个尾端注释的示例。
```

6.4. 行末注释

1）行末注释位于代码行之后，用于对代码行的补充说明。

2）行末注释应该与代码之间有足够的空格隔开，以增加可读性。

表 34 行末注释示例

```
let y: number = 20; // 这是一个行末注释的示例。
```

7. 开发目录规范

说明目录规范,开发人员在开发过程中按照开发的目录将相应的文件存放在指定目录下。

7.1. 根目录

1）根目录包含项目的主要文件和配置文件，如 package.json、tsconfig.json 等。

2）根目录下应该有一个名为 src 的目录，用于存放源代码。

7.2. src 目录

1）src 目录用于存放项目的源代码文件。

2）src 目录下可以包含多个子目录，用于组织不同功能模块的代码。

7.3. 源代码目录结构

1）modules: 存放各个功能模块的代码文件。

2）components: 存放可复用的 UI 组件的代码文件。

3）services: 存放与后端通信的服务代码文件。

4）utils: 存放各种工具函数的代码文件。

7.4. 其他目录

除了 src 目录外，项目还可以包含其他目录，如 node_module 目录用于存放静态资源文件。

7.5. 示例目录结构

表 35 目录结构示例

```
my-project/
├── package.json
├── tsconfig.json
├── public/
│   ├── index.html
```

```
└── src/
    ├── modules/
    |   ├── user/
    |   |   ├── User.ts
    |   |   ├── UserList.ts
    |   |   └── ...
    |   └── ...
    ├── components/
    |   ├── Button.tsx
    |   ├── Input.tsx
    |   └── ...
    ├── services/
    |   ├── ApiService.ts
    |   ├── AuthService.ts
    |   └── ...
    └── utils/
        ├── StringUtils.ts
        ├── ArrayUtils.ts
        └── ...
```

8. 代码的版本管理

说明代码版本管理的方法和规范，以便团队成员能够有效地管理和协作开发项目代码。

8.1. 版本控制系统

使用 Git 作为版本控制系统,将代码托管在远程仓库中(如 GitHub、GitLab、Bitbucket 等)。

8.2. 分支管理

1）主分支：main 分支用于存放稳定的、可发布的代码。

2）开发分支：develop 分支用于集成各个功能开发分支的代码。

3）功能分支：每个新功能或修复都应该在自己的分支上进行开发，并在开发完成后合并到 develop 分支。

8.3. 版本号规范

使用语义化版本号（Semantic Versioning）规范（例如 MAJOR.MINOR.PATCH）管理版本号。

1）主版本号（MAJOR）：当做了不兼容的 API 修改时增加。

2）次版本号（MINOR）：当添加了新功能，但是保持向后兼容时增加。

3）修订号（PATCH）：当做了向后兼容的 bug 修复时增加。

8.4. 提交信息规范

1）提交信息应该清晰、简洁，包含本次提交的目的和内容。

2）使用约定的提交格式（如 Angular 提交信息规范）：<type>(<scope>): <subject>。

8.5. 协作流程

1）每位开发人员在开始新任务之前应先拉取最新的代码。

2）完成任务后，先提交到本地仓库，再拉取远程仓库的最新代码，解决冲突后再推送到远程仓库。

8.6. 代码审查

1）所有的代码变更都应该经过代码审查后才能合并到主分支。

2）代码审查应该由团队中的其他成员进行，审查意见应该尽量提前给出。

8.7. 发布流程

1）发布前应该在 develop 分支上进行测试，确保没有明显的问题。

2）发布时应该更新版本号，并在发布说明中列出本次发布的变更内容。