**源代码说明**

1. 如该软件源代码超过3400行，则提供软件**初始1700行**+**末尾1700行(或某模块前、后各1700行。）**
2. 如该软件源代码少于3400行则需提供**完整源代码**；
3. 源代需转化过后用WORD或者记事本提交；
4. 请用以下方式分隔初始源代码和末尾源代码；

**\*\*主要源代码文件统计：\*\***

**- `code/main.py`: 1805行（主入口文件）**

**- `code/stub\_processor.py`: 912行**

**- `code/core/stub\_processor.py`: 833行**

**- `code/core/stub\_parser.py`: 473行**

**- `code/ui/app\_controller.py`: 636行**

**- `code/ui/app\_ui.py`: 426行**

**- `code/handlers/yaml\_handler.py`: 672行**

**- `code/utils/logger.py`: 550行**

**- `code/utils/config.py`: 333行**

**- 其他工具模块：约500行**

**\*\*总计：约7100+行源代码\*\***

**---**

**## 初始1700行：**

**\*\*YAMLWeave项目 -> code/main.py\*\***

**```python**

**#!/usr/bin/env python**

**# -\*- coding: utf-8 -\*-**

**"""**

**YAMLWeave 主入口模块**

**支持两种模式：基于注释的传统插桩和基于YAML配置的锚点插桩**

**本模块是YAMLWeave工具的主入口，实现了note.md中描述的功能：**

**1. 传统模式 - 基于代码注释插桩:**

**原始代码:**

**```c**

**void process\_data(int data) {**

**// TC001 STEP1: 数据边界检查**

**// code: if (data < 0 || data > 100) { log\_error("无效数据: %d", data); return; }**

**perform\_data\_processing(data);**

**}**

**```**

**插入后的代码:**

**```c**

**void process\_data(int data) {**

**// TC001 STEP1: 数据边界检查**

**// code: if (data < 0 || data > 100) { log\_error("无效数据: %d", data); return; }**

**if (data < 0 || data > 100) { // 通过桩插入**

**log\_error("无效数据: %d", data); // 通过桩插入**

**return; // 通过桩插入**

**} // 通过桩插入**

**perform\_data\_processing(data);**

**}**

**```**

**2. 分离模式 - 基于锚点与YAML配置的插桩:**

**YAML配置:**

**```yaml**

**TC001:**

**STEP1:**

**segment1:**

**- if (data < 0 || data > 100) {**

**- " log\_error(\"无效数据: %d\", data);"**

**- " return;"**

**- }**

**```**

**原始代码:**

**```c**

**void process\_data(int data) {**

**// TC001 STEP1 segment1**

**perform\_data\_processing(data);**

**}**

**```**

**插入后的代码:**

**```c**

**void process\_data(int data) {**

**// TC001 STEP1 segment1**

**if (data < 0 || data > 100) { // 通过桩插入**

**log\_error("无效数据: %d", data); // 通过桩插入**

**return; // 通过桩插入**

**} // 通过桩插入**

**perform\_data\_processing(data);**

**}**

**```**

**"""**

**import os**

**import sys**

**import tkinter as tk**

**from tkinter import filedialog, messagebox**

**import importlib.util**

**import site**

**import inspect**

**import logging**

**from pathlib import Path**

**import tempfile**

**import uuid**

**import datetime**

**import shutil**

**import glob**

**# 获取应用根目录的函数**

**def get\_application\_root():**

**"""**

**获取应用程序根目录，支持普通运行和PyInstaller打包后的场景**

**返回:**

**str: 应用程序根目录的绝对路径**

**"""**

**# PyInstaller打包后，\_MEIPASS变量包含应用程序的根目录**

**if hasattr(sys, '\_MEIPASS'):**

**# PyInstaller打包后的路径**

**return sys.\_MEIPASS**

**# 普通Python运行 - 获取当前文件所在目录的父目录（项目根目录）**

**current\_file = os.path.abspath(inspect.getfile(inspect.currentframe()))**

**code\_dir = os.path.dirname(current\_file)**

**return os.path.dirname(code\_dir) # 返回code目录的父目录作为项目根目录**

**# 设置应用根目录**

**APP\_ROOT = get\_application\_root()**

**# 配置模块导入路径**

**def setup\_import\_paths():**

**"""**

**配置模块导入路径，确保能够正确导入项目模块**

**无论是直接运行还是打包后运行**

**"""**

**# 将项目根目录添加到Python路径**

**if APP\_ROOT not in sys.path:**

**sys.path.insert(0, APP\_ROOT)**

**# 将code目录添加到Python路径（如果存在）**

**code\_dir = os.path.join(APP\_ROOT, "code")**

**if os.path.isdir(code\_dir) and code\_dir not in sys.path:**

**sys.path.insert(0, code\_dir)**

**# 打印当前应用根目录和sys.path (调试用)**

**print(f"应用根目录: {APP\_ROOT}")**

**print(f"sys.path: {sys.path}")**

**# 设置导入路径，确保后续导入的模块可用**

**setup\_import\_paths()**

**from utils.logger import setup\_global\_logger**

**setup\_global\_logger()**

**def safe\_ui\_log(ui, message, tag=None):**

**"""Safely call ui.log with optional tag support."""**

**if not ui or not hasattr(ui, "log"):**

**return**

**try:**

**import inspect**

**sig = inspect.signature(ui.log)**

**if tag is not None and "tag" in sig.parameters:**

**ui.log(message, tag=tag)**

**else:**

**ui.log(message)**

**except Exception as e:**

**logging.getLogger(\_\_name\_\_).error(f"UI log failed: {e}")**

**# 配置日志记录器**

**def setup\_logger():**

**"""配置并返回日志记录器"""**

**try:**

**# 创建日志记录器**

**logger = logging.getLogger(\_\_name\_\_)**

**logger.setLevel(logging.INFO)**

**# 配置控制台处理器**

**console\_handler = logging.StreamHandler()**

**console\_handler.setFormatter(logging.Formatter('%(levelname)s: %(message)s'))**

**logger.addHandler(console\_handler)**

**print(f"日志将写入: {APP\_ROOT}")**

**return logger**

**except Exception as e:**

**# 最小日志配置**

**print(f"创建日志记录器时出错: {str(e)}")**

**try:**

**import traceback**

**print(f"详细错误: {traceback.format\_exc()}")**

**except:**

**pass**

**logger = logging.getLogger(\_\_name\_\_)**

**logger.setLevel(logging.INFO)**

**handler = logging.StreamHandler()**

**handler.setFormatter(logging.Formatter('%(levelname)s: %(message)s'))**

**logger.addHandler(handler)**

**logger.warning(f"使用基本日志配置，原因: {str(e)}")**

**return logger**

**# 设置日志记录器**

**logger = setup\_logger()**

**# 尝试导入核心模块**

**try:**

**# 默认从 code 包导入**

**from code.core.stub\_processor import StubProcessor**

**from code.ui.app\_ui import YAMLWeaveUI**

**from code.ui.app\_controller import AppController**

**logger.info("成功导入核心模块 (code 前缀)")**

**except ImportError as e:**

**logger.warning(f"从 code 前缀导入失败: {str(e)}，尝试无前缀导入")**

**try:**

**# 尝试无需 code 前缀的导入以兼容旧的打包结构**

**from core.stub\_processor import StubProcessor**

**from ui.app\_ui import YAMLWeaveUI**

**from ui.app\_controller import AppController**

**logger.info("成功导入核心模块 (无前缀)")**

**except ImportError as inner\_e:**

**logger.error(f"无前缀导入仍然失败: {str(inner\_e)}")**

**# 尝试调整 sys.path 后再次导入 code 前缀模块**

**try:**

**sys.path.insert(0, os.path.join(APP\_ROOT, "code"))**

**from code.core.stub\_processor import StubProcessor**

**from code.ui.app\_ui import YAMLWeaveUI**

**from code.ui.app\_controller import AppController**

**logger.info("成功在调整路径后导入核心模块")**

**except Exception as final\_e:**

**logger.error(f"调整路径后导入核心模块仍然失败: {str(final\_e)}")**

**messagebox.showerror("导入错误", "无法导入核心处理模块，请确保项目结构完整")**

**sys.exit(1)**

**except Exception as e:**

**logger.error(f"导入核心模块失败: {str(e)}")**

**import traceback**

**logger.error(traceback.format\_exc())**

**messagebox.showerror("导入错误", f"导入核心模块时发生错误: {str(e)}")**

**sys.exit(1)**

**def show\_message(message):**

**"""显示简单的消息对话框，同时记录日志"""**

**logger.info(f"显示消息: {message}")**

**try:**

**messagebox.showinfo("YAMLWeave", message)**

**except Exception as e:**

**logger.error(f"显示消息对话框失败: {str(e)}")**

**print(f"[YAMLWeave] {message}")**

**def ensure\_data\_directory():**

**"""确保数据目录存在，包括示例和配置文件"""**

**try:**

**# 获取应用根目录**

**logger.info(f"应用根目录: {APP\_ROOT}")**

**# 创建示例目录**

**samples\_dir = os.path.join(APP\_ROOT, "samples")**

**try:**

**os.makedirs(samples\_dir, exist\_ok=True)**

**logger.info(f"确保示例目录存在: {samples\_dir}")**

**except Exception as e:**

**logger.warning(f"创建示例目录失败: {samples\_dir}, 错误: {str(e)}")**

**# 尝试在临时目录创建**

**samples\_dir = os.path.join(tempfile.gettempdir(), "yamlweave\_samples")**

**os.makedirs(samples\_dir, exist\_ok=True)**

**logger.info(f"在临时目录创建示例目录: {samples\_dir}")**

**# 返回目录路径供后续使用**

**return {**

**"samples\_dir": samples\_dir**

**}**

**except Exception as e:**

**logger.error(f"创建数据目录失败: {str(e)}")**

**# 记录详细错误**

**import traceback**

**logger.error(f"详细错误: {traceback.format\_exc()}")**

**# 回退到临时目录**

**fallback\_dir = os.path.join(tempfile.gettempdir(), "yamlweave\_samples")**

**logger.warning(f"回退到临时目录: {fallback\_dir}")**

**return {**

**"samples\_dir": fallback\_dir**

**}**

**def create\_example\_files(dirs):**

**"""创建基础示例文件和配置"""**

**try:**

**samples\_dir = dirs["samples\_dir"]**

**# 创建基础示例子目录，直接在samples\_dir下创建module1和module2**

**module1\_dir = os.path.join(samples\_dir, "module1")**

**module2\_dir = os.path.join(samples\_dir, "module2")**

**created\_dirs = []**

**for module\_dir in [module1\_dir, module2\_dir]:**

**try:**

**os.makedirs(module\_dir, exist\_ok=True)**

**created\_dirs.append(module\_dir)**

**logger.info(f"确保模块目录存在: {module\_dir}")**

**except Exception as e:**

**logger.warning(f"创建目录失败: {module\_dir}, 错误: {str(e)}")**

**if not created\_dirs:**

**logger.error("所有示例目录创建都失败，将尝试继续但可能会有问题")**

**# 创建module1中的示例文件**

**demo1\_1\_file = os.path.join(module1\_dir, "Demo1.1.c")**

**if not os.path.exists(demo1\_1\_file):**

**try:**

**with open(demo1\_1\_file, "w", encoding="utf-8") as f:**

**f.write('''/\*\***

**\* 模块1示例文件1**

**\* 用于测试多文件场景下的插桩功能**

**\*/**

**#include <stdio.h>**

**#include <string.h>**

**#include <stdlib.h>**

**/\*\***

**\* 模块1函数1: 数据验证与边界检查**

**\*/**

**int validate\_data(int value) {**

**// TC001 STEP1 segment1**

**printf("验证数据: %d\\n", value);**

**// TC101 STEP1 segment1**

**// TC201 STEP1 benchmark\_start**

**return value > 0 ? 1 : 0;**

**}**

**/\*\***

**\* 模块1函数2: 数据处理与转换**

**\*/**

**void process\_data(int data) {**

**// TC001 STEP2 segment1**

**printf("处理数据: %d\\n", data);**

**// TC102 STEP1 format\_check**

**// TC102 STEP1 segment1**

**// TC102 STEP1 check\_business\_rules**

**}**

**/\*\***

**\* 模块1函数3: 系统初始化与资源分配**

**\*/**

**int initialize\_system(void\* config) {**

**// TC101 STEP1 log\_init**

**// TC101 STEP1 segment2**

**printf("系统初始化中...\\n");**

**return 0;**

**}**

**''')**

**logger.info(f"创建示例文件: {demo1\_1\_file}")**

**except Exception as e:**

**logger.error(f"创建示例文件失败: {demo1\_1\_file}, 错误: {str(e)}")**

**# 创建module1中的第二个示例文件**

**demo1\_2\_file = os.path.join(module1\_dir, "Demo1.2.c")**

**if not os.path.exists(demo1\_2\_file):**

**try:**

**with open(demo1\_2\_file, "w", encoding="utf-8") as f:**

**f.write('''/\*\***

**\* 模块1示例文件2**

**\* 用于测试多文件场景下的插桩功能**

**\*/**

**#include <stdio.h>**

**#include <string.h>**

**/\*\***

**\* 模块1函数3: 消息处理**

**\*/**

**void process\_message(const char\* message) {**

**// TC001 STEP3 segment1**

**printf("处理消息: %s\\n", message);**

**// TC102 STEP2 segment1**

**// TC102 STEP2 verify\_permissions**

**// TC102 STEP2 after\_verification**

**}**

**/\*\***

**\* 模块1函数4: 系统状态检查**

**\*/**

**int check\_system\_status() {**

**// TC001 STEP4 segment1**

**printf("检查系统状态...\\n");**

**// TC103 STEP1 segment1**

**// TC103 STEP1 close\_connections**

**// TC103 STEP1 rollback**

**return 1;**

**}**

**/\*\***

**\* 模块1函数5: 错误处理与报告**

**\*/**

**void handle\_error(int error\_code, const char\* error\_description, const char\* error\_details) {**

**// TC103 STEP2 log\_error**

**// TC103 STEP2 segment2**

**// TC103 STEP2 completed**

**printf("处理错误: %d, %s\\n", error\_code, error\_description);**

**}**

**''')**

**logger.info(f"创建示例文件: {demo1\_2\_file}")**

**except Exception as e:**

**logger.error(f"创建示例文件失败: {demo1\_2\_file}, 错误: {str(e)}")**

**# 创建module2中的示例文件**

**demo2\_1\_file = os.path.join(module2\_dir, "Demo2.1.c")**

**if not os.path.exists(demo2\_1\_file):**

**try:**

**with open(demo2\_1\_file, "w", encoding="utf-8") as f:**

**f.write('''/\*\***

**\* 模块2示例文件1**

**\* 用于测试多文件场景下的插桩功能**

**\*/**

**#include <stdio.h>**

**#include <stdlib.h>**

**/\*\***

**\* 模块2函数1: 初始化模块**

**\*/**

**void init\_module() {**

**// TC002 STEP1 segment1**

**printf("初始化模块2...\\n");**

**// TC101 STEP2 network\_init**

**// TC101 STEP2 before\_db\_init**

**// TC101 STEP2 segment3**

**}**

**/\*\***

**\* 模块2函数2: 释放资源**

**\*/**

**void release\_resources() {**

**// TC002 STEP2 segment1**

**printf("释放模块2资源...\\n");**

**// TC202 STEP1 test\_min\_max**

**// TC202 STEP1 test\_special\_chars**

**}**

**/\*\***

**\* 模块2函数3: 性能测试与基准测量**

**\*/**

**void performance\_test(int iterations) {**

**// TC201 STEP1 benchmark\_start**

**printf("性能测试中...\\n");**

**// TC203 STEP1 complex\_structures**

**// TC201 STEP1 benchmark\_end**

**}**

**''')**

**logger.info(f"创建示例文件: {demo2\_1\_file}")**

**except Exception as e:**

**logger.error(f"创建示例文件失败: {demo2\_1\_file}, 错误: {str(e)}")**

**# 创建module2中的第二个示例文件**

**demo2\_2\_file = os.path.join(module2\_dir, "Demo2.2.c")**

**if not os.path.exists(demo2\_2\_file):**

**try:**

**with open(demo2\_2\_file, "w", encoding="utf-8") as f:**

**f.write('''/\*\***

**\* 模块2示例文件2**

**\* 用于测试多文件场景下的插桩功能**

**\*/**

**#include <stdio.h>**

**#include <math.h>**

**/\*\***

**\* 模块2函数3: 处理原始数据**

**\*/**

**float process\_raw\_data(int raw\_data) {**

**// TC002 STEP3 segment1**

**printf("处理原始数据: %d\\n", raw\_data);**

**return (float)raw\_data \* 1.5f;**

**}**

**/\*\***

**\* 模块2函数4: 验证结果**

**\*/**

**int verify\_result(float result, float expected) {**

**// TC002 STEP4 segment1**

**printf("验证结果: 实际 %.2f vs 期望 %.2f\\n", result, expected);**

**// TC202 STEP1 test\_min\_max**

**return fabs(result - expected) < (expected \* 0.05f) ? 1 : 0;**

**}**

**/\*\***

**\* 模块2函数5: 复杂数据处理**

**\*/**

**void process\_complex\_data(void\* data\_array, int array\_size) {**

**// TC203 STEP1 complex\_structures**

**printf("处理复杂数据结构...\\n");**

**}**

**''')**

**logger.info(f"创建示例文件: {demo2\_2\_file}")**

**except Exception as e:**

**logger.error(f"创建示例文件失败: {demo2\_2\_file}, 错误: {str(e)}")**

**# 为多文件测试创建YAML配置文件**

**basic\_yaml\_content = '''# YAMLWeave 基础测试配置\n# 用于验证在多个文件场景下成功插入代码\n\n# 模块1 测试用例\nTC001:\n STEP1:\n segment1: |\n if (value < 0) {\n printf("模块1-文件1: 检测到无效值 %d\\n", value);\n return 0;\n }\n STEP2:\n segment1: |\n int processed\_data = data \* 2;\n printf("模块1-文件1: 数据已处理为 %d\\n", processed\_data);\n STEP3:\n segment1: |\n printf("模块1-文件2: 开始处理消息 '%s'\\n", message);\n if (message == NULL || \*message == '\\0') {\n printf("模块1-文件2: 无效消息!\\n");\n return;\n }\n STEP4:\n segment1: |\n int system\_status = 1;\n if (system\_status != 1) {\n printf("模块1-文件2: 系统状态异常\\n");\n return 0;\n }\n\n# 模块2 测试用例\nTC002:\n STEP1:\n segment1: |\n static int initialized = 0;\n if (initialized) {\n printf("模块2-文件1: 已经初始化过了\\n");\n return;\n }\n initialized = 1;\n STEP2:\n segment1: |\n static int resources\_released = 0;\n if (resources\_released) {\n printf("模块2-文件1: 资源已释放\\n");\n return;\n }\n resources\_released = 1;\n STEP3:\n segment1: |\n if (raw\_data < 0) {\n printf("模块2-文件2: 无效原始数据 %d\\n", raw\_data);\n return 0.0f;\n }\n STEP4:\n segment1: |\n float diff = result - expected;\n float tolerance = expected \* 0.05f;\n printf("模块2-文件2: 差异 %.2f, 容差 %.2f\\n", diff, tolerance);'''**

**return {**

**"example\_project\_path": samples\_dir,**

**"simple\_examples\_dir": samples\_dir, # 将simple\_examples\_dir直接指向samples\_dir**

**"simple\_yaml\_path": "",**

**"basic\_yaml\_content": basic\_yaml\_content**

**}**

**except Exception as e:**

**logger.error(f"创建示例文件时出错: {str(e)}")**

**# 返回最小必要的结果，防止后续代码崩溃**

**return {**

**"example\_project\_path": dirs["samples\_dir"],**

**"simple\_examples\_dir": dirs["samples\_dir"], # 将simple\_examples\_dir直接指向samples\_dir**

**"simple\_yaml\_path": "",**

**"basic\_yaml\_content": ""**

**}**

**def create\_advanced\_test\_cases(dirs):**

**"""创建更全面的测试用例配置，覆盖所有测试功能点"""**

**samples\_dir = dirs["samples\_dir"]**

**logger.info(f"开始创建高级测试用例: {samples\_dir}")**

**# 创建高级测试YAML配置内容**

**advanced\_yaml\_content = '''# YAMLWeave 高级测试配置文件**

**# 演示不同命名风格的segment，包括序号型和功能描述型**

**# ====== 系统初始化测试 ======**

**# TC101: 初始化测试 - 混合使用不同命名风格**

**TC101:**

**# STEP1: 系统初始化**

**STEP1:**

**# 序号型命名**

**segment1: |**

**if (config == NULL) {**

**printf("错误: 配置为空\\n");**

**return INIT\_ERROR\_NULL\_CONFIG;**

**}**

**printf("配置检查通过\\n");**

**# 功能描述型命名**

**log\_init: |**

**int log\_status = init\_log\_system(config->log\_level);**

**if (log\_status != SUCCESS) {**

**printf("警告: 日志系统初始化失败 (状态码: %d)\\n", log\_status);**

**} else {**

**log\_message(LOG\_INFO, "日志系统已初始化");**

**}**

**# 序号型命名**

**segment2: |**

**system\_context = (SystemContext\*)malloc(sizeof(SystemContext));**

**if (system\_context == NULL) {**

**log\_message(LOG\_ERROR, "致命错误: 无法分配系统上下文内存");**

**return INIT\_ERROR\_MEMORY\_ALLOC;**

**}**

**memset(system\_context, 0, sizeof(SystemContext));**

**# STEP2: 模块初始化**

**STEP2:**

**# 功能描述型命名**

**network\_init: |**

**system\_context->network\_handle = network\_initialize(config->net\_params);**

**if (system\_context->network\_handle == NULL) {**

**log\_message(LOG\_ERROR, "网络模块初始化失败");**

**return INIT\_ERROR\_NETWORK;**

**}**

**log\_message(LOG\_INFO, "网络模块已初始化");**

**# 位置型命名**

**before\_db\_init: |**

**log\_message(LOG\_INFO, "准备初始化数据库连接...");**

**if (config->db\_host == NULL) {**

**log\_message(LOG\_ERROR, "数据库主机未配置");**

**return INIT\_ERROR\_DATABASE\_CONFIG;**

**}**

**# 序号型命名**

**segment3: |**

**DatabaseConfig db\_config;**

**db\_config.host = config->db\_host;**

**db\_config.port = config->db\_port;**

**db\_config.username = config->db\_user;**

**db\_config.password = config->db\_password;**

**system\_context->db\_handle = database\_connect(&db\_config);**

**if (system\_context->db\_handle == NULL) {**

**log\_message(LOG\_ERROR, "数据库模块初始化失败");**

**return INIT\_ERROR\_DATABASE;**

**}**

**log\_message(LOG\_INFO, "数据库模块已初始化");**

**# ====== 数据验证测试 ======**

**# TC102: 数据验证测试 - 混合使用不同命名风格**

**TC102:**

**# STEP1: 输入数据验证**

**STEP1:**

**# 功能描述型命名**

**format\_check: |**

**if (data == NULL || data->format != DATA\_FORMAT\_V1) {**

**log\_message(LOG\_ERROR, "数据格式无效");**

**return VALIDATION\_ERROR\_FORMAT;**

**}**

**# 序号型命名**

**segment1: |**

**uint32\_t calculated\_checksum = calculate\_checksum(data->buffer, data->length);**

**if (calculated\_checksum != data->checksum) {**

**log\_message(LOG\_WARNING, "数据校验和不匹配: 期望 %u, 实际 %u",**

**data->checksum, calculated\_checksum);**

**return VALIDATION\_ERROR\_CHECKSUM;**

**}**

**# 组合型命名**

**check\_business\_rules: |**

**if (data->timestamp < get\_system\_time() - MAX\_DATA\_AGE) {**

**log\_message(LOG\_WARNING, "数据已过期");**

**return VALIDATION\_ERROR\_EXPIRED;**

**}**

**if (data->value < MIN\_ACCEPTABLE\_VALUE || data->value > MAX\_ACCEPTABLE\_VALUE) {**

**log\_message(LOG\_WARNING, "数据值超出可接受范围: %f", data->value);**

**return VALIDATION\_ERROR\_VALUE\_RANGE;**

**}**

**# STEP2: 权限验证**

**STEP2:**

**# 序号型命名**

**segment1: |**

**UserContext\* user = get\_current\_user();**

**if (user == NULL) {**

**log\_message(LOG\_ERROR, "未找到用户上下文");**

**return VALIDATION\_ERROR\_NO\_USER;**

**}**

**# 功能描述型命名**

**verify\_permissions: |**

**if ((user->permissions & PERMISSION\_DATA\_WRITE) == 0) {**

**log\_message(LOG\_WARNING, "用户 %s 没有数据写入权限", user->username);**

**return VALIDATION\_ERROR\_PERMISSION;**

**}**

**log\_message(LOG\_INFO, "用户 %s 权限验证通过", user->username);**

**# 后处理命名**

**after\_verification: |**

**log\_message(LOG\_INFO, "验证完成，更新用户访问时间戳");**

**user->last\_access\_time = get\_system\_time();**

**# ====== 错误处理测试 ======**

**# TC103: 错误处理测试 - 混合使用不同命名风格**

**TC103:**

**# STEP1: 资源清理**

**STEP1:**

**# 序号型命名**

**segment1: |**

**if (resource != NULL) {**

**if (resource->buffer != NULL) {**

**free(resource->buffer);**

**resource->buffer = NULL;**

**}**

**free(resource);**

**resource = NULL;**

**}**

**# 功能描述型命名**

**close\_connections: |**

**if (connection\_is\_open(conn)) {**

**log\_message(LOG\_INFO, "关闭连接...");**

**int close\_status = connection\_close(conn);**

**if (close\_status != SUCCESS) {**

**log\_message(LOG\_WARNING, "关闭连接时出错: %d", close\_status);**

**return close\_status;**

**}**

**}**

**# 动作型命名**

**rollback: |**

**if (transaction\_is\_active(transaction)) {**

**log\_message(LOG\_INFO, "回滚事务...");**

**int rollback\_status = transaction\_rollback(transaction);**

**if (rollback\_status != SUCCESS) {**

**log\_message(LOG\_ERROR, "事务回滚失败: %d", rollback\_status);**

**// 即使回滚失败也继续处理**

**}**

**}**

**# STEP2: 错误报告**

**STEP2:**

**# 功能描述型命名**

**log\_error: |**

**log\_message(LOG\_ERROR, "操作失败: %s (错误码: %d)", error\_description, error\_code);**

**if (error\_details != NULL) {**

**log\_message(LOG\_DEBUG, "错误详情: %s", error\_details);**

**}**

**# 序号型命名**

**segment2: |**

**if (error\_code >= CRITICAL\_ERROR\_THRESHOLD) {**

**send\_admin\_notification("系统错误", error\_description);**

**log\_message(LOG\_INFO, "已发送管理员通知");**

**}**

**# 状态型命名**

**completed: |**

**log\_message(LOG\_INFO, "错误处理流程已完成");**

**return ERROR\_HANDLED;**

**# ====== 性能与边界测试 ======**

**# TC201: 性能测试 - 测试桩代码对性能的影响**

**TC201:**

**# STEP1: 性能基准测试**

**STEP1:**

**benchmark\_start: |**

**clock\_t start\_time = clock();**

**long long iteration\_counter = 0;**

**printf("性能测试开始\\n");**

**benchmark\_end: |**

**clock\_t end\_time = clock();**

**double elapsed\_seconds = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;**

**printf("性能测试完成: 耗时 %.6f 秒, 迭代次数 %lld\\n", elapsed\_seconds, iteration\_counter);**

**printf("每秒处理 %.2f 次操作\\n", iteration\_counter / elapsed\_seconds);**

**# TC202: 边界测试 - 测试各种边界情况**

**TC202:**

**# STEP1: 边界值测试**

**STEP1:**

**test\_min\_max: |**

**// 测试整型边界值**

**if (value == INT\_MIN || value == INT\_MAX) {**

**printf("检测到整型边界值: %d\\n", value);**

**return BOUNDARY\_CASE;**

**}**

**// 测试浮点数边界值**

**if (fabs(double\_value) < 1e-10 || fabs(double\_value) > 1e10) {**

**printf("检测到浮点数边界值: %e\\n", double\_value);**

**return BOUNDARY\_CASE;**

**}**

**test\_special\_chars: |**

**// 测试特殊字符输入**

**if (strchr(input, '\\\\') || strchr(input, '\\"') || strchr(input, '\\'')) {**

**printf("检测到特殊字符输入\\n");**

**// 对特殊字符进行转义处理**

**char\* escaped = escape\_string(input);**

**printf("转义后: %s\\n", escaped);**

**free(escaped);**

**}**

**# TC203: 复杂代码段测试 - 测试多级缩进和循环**

**TC203:**

**STEP1:**

**complex\_structures: |**

**// 多级嵌套的数据结构和控制流测试**

**for (int i = 0; i < array\_size; i++) {**

**if (data\_array[i].type == TYPE\_COMPOSITE) {**

**CompositeData\* composite = (CompositeData\*)data\_array[i].data;**

**for (int j = 0; j < composite->child\_count; j++) {**

**ChildData\* child = &composite->children[j];**

**if (child->flags & FLAG\_REQUIRES\_PROCESSING) {**

**switch (child->category) {**

**case CATEGORY\_A:**

**process\_category\_a(child);**

**break;**

**case CATEGORY\_B:**

**if (child->priority > HIGH\_PRIORITY) {**

**process\_high\_priority\_b(child);**

**} else {**

**process\_normal\_priority\_b(child);**

**}**

**break;**

**default:**

**log\_message(LOG\_WARNING, "未知类别: %d", child->category);**

**continue;**

**}**

**// 统计处理结果**

**processed\_count++;**

**total\_size += child->size;**

**}**

**}**

**}**

**}**

**printf("处理完成: %d 项, 总大小: %lu 字节\\n", processed\_count, total\_size);'''**

**# 返回高级测试YAML内容**

**return advanced\_yaml\_content**

**def create\_combined\_yaml(dirs, basic\_content, advanced\_content):**

**"""创建合并的YAML配置文件，包含基础测试和高级测试的所有用例"""**

**try:**

**samples\_dir = dirs["samples\_dir"]**

**combined\_yaml\_file = os.path.join(samples\_dir, "all\_tests.yaml")**

**# 确保内容不为None**

**basic\_content = basic\_content or "# 基础测试内容创建失败"**

**advanced\_content = advanced\_content or "# 高级测试用例创建失败"**

**try:**

**with open(combined\_yaml\_file, "w", encoding="utf-8") as f:**

**f.write('''# YAMLWeave 综合测试配置文件**

**# 包含所有基础测试和高级测试用例**

**# ---------------------------------------**

**# =========== 基础测试用例 ===========**

**''')**

**f.write(basic\_content)**

**f.write('''**

**# =========== 高级测试用例 ===========**

**''')**

**f.write(advanced\_content)**

**logger.info(f"创建综合测试YAML配置: {combined\_yaml\_file}")**

**except Exception as e:**

**logger.error(f"写入综合测试YAML配置失败: {str(e)}")**

**# 尝试在临时目录创建**

**import tempfile**

**temp\_dir = tempfile.gettempdir()**

**combined\_yaml\_file = os.path.join(temp\_dir, "yamlweave\_all\_tests.yaml")**

**try:**

**with open(combined\_yaml\_file, "w", encoding="utf-8") as f:**

**f.write('''# YAMLWeave 临时综合测试配置文件**

**# 包含所有基础测试和高级测试用例**

**# ---------------------------------------**

**''')**

**f.write(basic\_content)**

**f.write('''**

**# =========== 高级测试用例 ===========**

**''')**

**f.write(advanced\_content)**

**logger.info(f"在临时目录创建综合测试YAML配置: {combined\_yaml\_file}")**

**except Exception as inner\_e:**

**logger.error(f"在临时目录创建综合测试YAML配置也失败: {str(inner\_e)}")**

**# 回退到使用内存中的临时路径**

**combined\_yaml\_file = ""**

**return combined\_yaml\_file**

**except Exception as e:**

**logger.error(f"创建合并YAML配置文件时出错: {str(e)}")**

**return "" # 返回空字符串，表示没有创建成功**

**def create\_fallback\_controller(ui\_instance):**

**"""**

**创建后备控制器类，在无法加载原始AppController时使用**

**提供与原始控制器相同的基本功能**

**"""**

**logger.info("创建后备AppController类...")**

**class FallbackAppController:**

**"""**

**后备控制器实现，提供基本的文件处理和插桩功能**

**在无法导入原始AppController时使用**

**"""**

**def \_\_init\_\_(self, ui):**

**self.ui = ui**

**self.processor = None**

**# 添加锚点存储**

**self.anchors\_found = 0**

**self.anchor\_details = []**

**# 设置进程回调函数**

**ui.set\_process\_callback(self.process\_files)**

**logger.info("后备AppController已初始化")**

**# 尝试导入StubProcessor**

**try:**

**# 尝试多种可能的导入路径**

**stub\_processor\_module = None**

**import\_paths = [**

**"core.stub\_processor",**

**"code.core.stub\_processor",**

**"YAMLWeave.core.stub\_processor",**

**"stub\_processor" # 添加直接导入路径**

**]**

**for path in import\_paths:**

**try:**

**logger.info(f"尝试从{path}导入StubProcessor")**

**\_\_import\_\_(path)**

**stub\_processor\_module = sys.modules[path]**

**logger.info(f"成功从{path}导入模块")**

**# 设置控制器级别为完整功能**

**self.controller\_level = 0**

**break**

**except ImportError as e:**

**logger.warning(f"从{path}导入失败: {str(e)}")**

**continue**

**except Exception as e:**

**logger.error(f"从{path}导入时发生非导入错误: {str(e)}")**

**continue**

**if stub\_processor\_module and hasattr(stub\_processor\_module, "StubProcessor"):**

**self.StubProcessor = stub\_processor\_module.StubProcessor**

**logger.info("成功导入StubProcessor类")**

**else:**

**# 如果无法导入，尝试手动查找文件**

**logger.warning("无法通过常规方式导入StubProcessor，尝试手动查找文件...")**

**search\_paths = [**

**APP\_ROOT,**

**os.path.dirname(APP\_ROOT),**

**os.path.join(APP\_ROOT, "code"),**

**os.path.join(APP\_ROOT, "core"),**

**os.path.join(APP\_ROOT, "code", "core"),**

**os.path.join(os.path.abspath(os.curdir), "code", "core") # 添加当前工作目录**

**]**

**if hasattr(sys, '\_MEIPASS'):**

**search\_paths.append(sys.\_MEIPASS)**

**search\_paths.append(os.path.join(sys.\_MEIPASS, "core"))**

**stub\_processor\_file = None**

**for path in search\_paths:**

**if os.path.exists(path):**

**logger.info(f"搜索路径: {path}")**

**file\_candidates = [**

**"stub\_processor.py",**

**"core/stub\_processor.py",**

**"core\\stub\_processor.py",**

**"code/core/stub\_processor.py",**

**"code\\core\\stub\_processor.py"**

**]**

**for filename in file\_candidates:**

**filepath = os.path.join(path, filename)**

**if os.path.exists(filepath):**

**stub\_processor\_file = filepath**

**logger.info(f"找到StubProcessor文件: {stub\_processor\_file}")**

**break**

**if stub\_processor\_file:**

**break**

**if stub\_processor\_file:**

**# 手动导入StubProcessor类**

**try:**

**spec = importlib.util.spec\_from\_file\_location("stub\_processor", stub\_**