

计算机软件著作权登记用软件说明书

软件名称：左右道飞MCP服务系统

英文名称：Daofy MCP Server

版本号：V2026.06.08.1

开发完成日期：2026年6月8日

著作权人：吉林省左右软件开发有限公司

编程语言：Python

运行环境：Windows 7+/Server 2012+, Python 3.10-3.14, 需安装 Edge/Chrome 浏览器 (PDF生成用), 可选安装 Delphi 编译器

软件类型：应用软件 — AI集成开发工具

第一章 引言

1.1 背景

Delphi 作为成熟的原生 Windows 开发平台，在企业管理软件、工业控制、金融系统等领域仍有广泛部署。然而 Delphi 开发者长期面临以下痛点：在 AI 辅助编程场景中，AI 助手无法直接理解 Delphi 项目结构和编译器参数，无法在对话中编译项目、查询 RTL/VCL 类的 API 定义，也无法自动修改 `.pas` 或 `.dfm` 文件。现有的 MCP (Model Context Protocol) 生态中缺少专门面向 Delphi 开发工具链的桥梁软件，导致 AI 的能力无法延伸到 Delphi 工程环境中。

左右道飞MCP服务系统 (Daofy MCP Server) 针对上述问题，以 MCP 协议为通信标准，将 Delphi 编译器、知识库搜索引擎、文件管理、Git 版本控制、组件管理等一系列开发工具封装为 AI 可调用的标准化工具接口，使 AI 助手能够像人类开发者一样编译项目、搜索 API、编辑源码、管理版本。"道飞"取"为创意插上翅膀"之意，旨在通过 AI 增强 Delphi 开发效率。

1.2 目的

本说明书旨在全面介绍左右道飞MCP服务系统的功能特性、系统架构、安装配置和操作使用方法。用户通过阅读本说明书，能够了解系统的整体设计思路和分层架构，掌握各模块的核心功能和 workflows，熟悉 14 个 MCP 工具的用途和调用方式，并能够独立完成系统的部署、配置和日常使用。

1.3 范围

本说明书共分七章，涵盖内容如下：

- 第一章 引言：背景、目的、范围和术语定义

- **第二章 系统概述**: 软件简介、功能总览和运行环境
- **第三章 系统架构**: 总体架构和模块划分, 用 mermaid 流程图展示分层设计
- **第四章 模块详细设计**: 对配置模块、数据模型、入口服务、业务逻辑、工具实现和工具类六个模块逐一展开, 包含功能概述、核心类与函数、核心流程和关键特性
- **第五章 安装与配置**: 系统要求、安装方式和配置方法
- **第六章 操作说明**: 操作模式概述和 13 个操作示例, 覆盖全部 14 个 MCP 工具的使用场景
- **第七章 测试与验收**: 测试方案和验收标准

1.4 术语与定义

术语	定义
MCP	Model Context Protocol, AI 模型与外部工具之间的标准化通信协议, 由 Anthropic 提出
MCP Server	实现 MCP 协议的服务端程序, 通过 JSON-RPC 接口向 AI 客户端暴露工具和资源
知识库 (KB)	系统内置的语义搜索引擎, 包括 Delphi 官方源码知识库、项目知识库、第三方库知识库和通用文档知识库
DPROJ	Delphi 项目文件 (XML 格式), 包含编译器参数、搜索路径、编译事件等完整项目配置
DFM	Delphi 窗体文件, 描述窗体和组件的布局、属性, 支持二进制和文本两种格式
编译器事件	Delphi 编译过程中的 PreBuildEvent、PreLinkEvent、PostBuildEvent 等钩子, 支持环境变量替换
pasfmt	Delphi 源码格式化工具, 用于自动规范化 .pas 文件的缩进和排版

第二章 系统概述

2.1 软件简介

左右道飞MCP服务系统 (Daofy MCP Server) 是一款基于 MCP 协议的桥梁软件, 让 AI 助手能够直接操作 Delphi 开发工具链和知识库。

系统共包含 **71 个源文件** (约 37,271 行代码), 按功能划分为以下模块:

| 配置 (config/) | 1 | 592 | | 数据模型 (models/) | 6 | 477 | | 入口 (server.py) (root/) | 3 | 1497 | | 业务逻辑 (services/) | 21 | 15817 | | 工具实现 (tools/) | 28 | 15489 | | 工具类 (utils/) | 12 | 3399 |

测试覆盖: 45 个测试文件

- **AI 驱动编译**: AI 助手直接调用 MSBuild 或 dcc32 编译 Delphi 项目, 无需手动切换 IDE
- **语义知识库**: 内置 Delphi RTL/VCL/FMX 源码知识库, 支持类名、函数名和自然语言语义搜索
- **智能文件编辑**: 自动检测 .pas / .dfm 编码格式, 写入前备份, 支持批量写入和行号偏移自适应
- **代码托管集成**: 通过统一接口操作 Gitea/GitHub/GitLab/Gitee/GitCode 平台和 Git 本地操作

2.2 功能总览

系统围绕 Delphi 开发全流程, 提供编译器集成、知识库检索、文件管理、代码托管、组件管理、环境诊断、异步任务、经验记忆、编码规范查询、更新管理等功能类型。所有功能通过 MCP 协议以标准化工具接口暴露给 AI 客户端 (如 Claude Desktop、CodeArts Agent), AI 通过调用工具名称和参数完成对应操作。

功能类别	功能模块	说明
项目全生命周期管理	project	编译 Delphi 项目文件，支持 MSBuild 和 dcc32 两种编译模式，支持单文件编译、项目配置查看与修改、源码审计和运行时检查
环境检查	check_environment	检测 Delphi 编译器是否可用、搜索已安装的编译器版本、安装 pasfmt 代码格式化工具及其 RAD Studio 插件
知识库搜索与管理	delphi_kb	按类名、函数名或自然语言搜索 Delphi 源码知识库；构建、重建或增量更新项目/第三方库/文档知识库
Delphi 文件操作	delphi_file	读取和写入 .pas / .dfm / .dproj 文件，支持编码自动检测、写入前自动备份、按行号部分替换和批量编辑
组件管理	manage_component	在 DFM 文件中添加、删除、修改组件属性，或通过 Pascal 代码动态生成组件 DFM 描述，自动同步 PAS 声明
代码托管	code_hosting	统一管理 Gitea/GitHub/GitLab/Gitee/GitCode 五大平台的工单和仓库操作，同时支持 Git 本地 add/commit/push
异步任务管理	async_task	启动、查询、取消后台长时间运行任务（如知识库构建），支持长轮询进度查询
组件包管理	package	编译并安装 Delphi 组件包（.dproj / .dpc），支持 win32/win64 双平台，可选 IDE 自动注册
编码规则查询	get_coding_rules	获取 Delphi 编码规范文档，支持按章节分段获取以减少 token 消耗
工具帮助	tool_help	按需返回任意工具的完整帮助文档，包含参数说明、使用示例和触发词
经验记忆管理	experience	保存 AI 解决问题的做法、语义搜索已有经验、合并去重、定期清理低价值条目
版本更新管理	daofy_update	检查 Daofy 自身是否有新版本发布，通过 git pull 执行更新
软著文档生成	generate_copyright	自动生成软件著作权登记所需的源代码文档、说明书和摘要
Delphi 自动化	automate_delphi	通过 JSON 脚本驱动 Delphi 程序执行 UI 自动化操作，支持截图捕获

2.3 运行环境

2.3.1 硬件环境

项目	最低配置	推荐配置
CPU	双核处理器，2.0 GHz	四核处理器，2.5 GHz
内存	4 GB	8 GB
磁盘空间	500 MB（系统本体）	2 GB（含知识库存储）
显示器分辨率	1366×768	1920×1080

2.3.2 软件环境

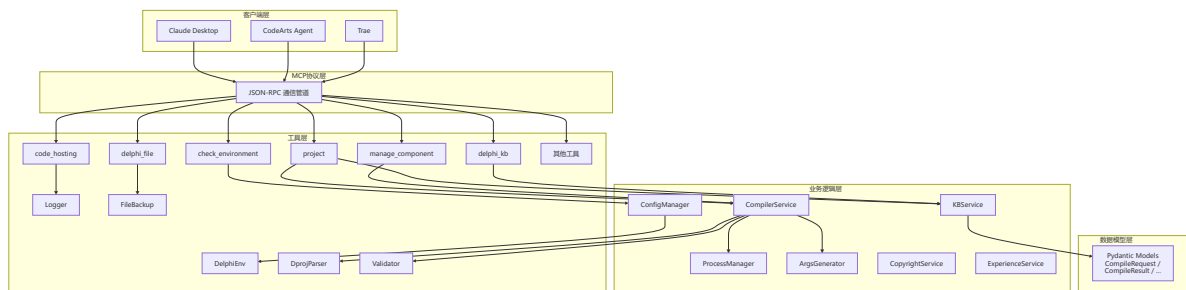
项目	要求
操作系统	Windows 7 SP1 及以上 / Windows Server 2012 R2 及以上
Python 运行时	3.10 — 3.14
浏览器（PDF 生成用）	Microsoft Edge 或 Google Chrome（最新稳定版）

项目	要求
Delphi 编译器 (可选)	Embarcadero Delphi 2005 — 13 任一版本
7-Zip (可选, CHM 解压用)	7-Zip 18.0 及以上
Git (可选, 更新用)	Git 2.30 及以上

第三章 系统架构

3.1 总体架构

系统采用五层分层架构, 从上到下依次为: MCP 协议层、工具层、业务逻辑层、数据模型层和工具层。每一层仅与其相邻层发生依赖, 层次边界清晰。用户请求从 MCP 协议层进入, 经工具层路由到业务逻辑层处理, 结果沿原路径返回。



各层的职责如下:

层次	职责说明
MCP 协议层	通过标准输入/输出建立 JSON-RPC 双工通信管道, 接收 AI 发送的工具调用请求并返回结构化响应
工具层	将系统能力封装为 14 个 MCP 工具, 每个工具负责参数校验、请求路由和结果格式化
业务逻辑层	实现各领域的核心业务逻辑, 包括编译调度、知识库构建与检索、配置文件管理、进程管理等
数据模型层	定义 Pydantic 数据模型, 统一请求参数和返回结果的数据结构, 提供序列化/反序列化能力
工具层	提供跨模块共享的基础能力, 包括 Delphi 环境检测、DPROJ 文件解析、路径验证和日志记录

3.2 模块划分

系统根据职责边界划分为六大模块: 配置模块、数据模型模块、入口模块、业务逻辑模块、工具实现模块和工具类模块。各模块代码分布如下表:

模块名	路径	文件数	代码行数	职责说明
配置模块	config/	1	592	管理编译器路径配置和工具文档元数据, 提供编译器路径自愈和注册表回退逻辑
数据模型模块	models/	6	477	定义编译请求、编译结果、编译器配置、命令参数等 Pydantic 数据模型
入口模块	root/	3	1497	MCP Server 启动入口, 注册 14 个工具并初始化各服务实例

模块名	路径	文件数	代码行数	职责说明
业务逻辑模块	services/	21	15817	实现编译器调度、知识库构建与搜索、配置文件管理、进程管理、版权生成和自动化测试等核心业务
工具实现模块	tools/	28	15489	将业务逻辑封装为 MCP 工具接口，处理参数校验、结果格式化和错误处理
工具类模块	utils/	12	3399	提供 Delphi 环境检测、DPROJ 文件 XML 解析、路径验证、编码检测和日志记录等通用能力

第四章 模块详细设计

4.1 配置模块 (src/config/, 1 文件, 592 行)

功能概述:

配置模块是系统的基础支撑模块，负责管理编译器配置文件的读取、回退和自愈，同时维护工具文档元数据（工具名称、简要描述）。该模块在系统启动时被 ConfigManager 调用，从 `compilers.json` 加载已安装的 Delphi 编译器信息；若配置文件路径异常或注册表读取失败，模块自动回退到项目根目录下的 `config/` 路径，确保系统不因路径问题崩溃。输出为编译器版本列表和工具描述表。

核心类与函数:

核心组件	所属文件	职责
TOOL_NAMES / TOOL_SHORT_DESC	tool_docs.py	定义 14 个工具的标准名称和单行描述，用于 MCP list_tools 响应
COMPILERS_CONFIG (常量)	compilers.json	记录各 Delphi 版本的注册表键名、默认安装路径和 dcc32/dcc64 可执行文件名

关键特性:

- **路径自愈**: 检测到 `compilers.json` 不在 `src/config/` 时自动回退到项目根目录下的 `config/` 目录
- **注册表回退**: 注册表读取失败时使用硬编码默认路径列表，仍可识别 Delphi XE 至 Delphi 13 共十几个版本
- **工具元数据集中管理**: 所有工具的名称和 `short_desc` 集中在 `tool_docs.py` 维护，新增工具时只需修改一处

4.2 数据模型模块 (src/models/, 6 文件, 477 行)

功能概述:

数据模型模块定义系统内部所有结构化数据的 Schema，使用 Pydantic 实现类型校验和序列化。该模块在业务逻辑层和工具层之间传递数据时确保格式正确。主要模型包括：编译请求（源文件路径、输出路径、条件编译符号、搜索路径）、编译结果（返回码、输出消息、生成文件路径）、编译器配置（版本号、可执行文件路径、平台）和命令参数（`dcc32` 命令行参数结构）。输入为原始参数字典，输出为经过类型校验的 Pydantic 模型实例。

核心类与函数:

核心组件	所属文件	职责
CompileRequest	compile_request.py	定义编译请求的完整参数结构，包含源文件、输出文件、搜索路径和条件编译符号

核心组件	所属文件	职责
CompileResult	compile_result.py	定义编译返回结果，包含返回码、标准输出、标准错误和生成文件列表
CompilerConfig	compiler_config.py	定义编译器配置，包含版本号、可执行文件路径、平台架构和注册表键名
CommandArgs	command_args.py	定义 dcc32 命令行参数结构，用于 args_generator 生成完整的 dcc32 参数串
CompileHistory	compile_history.py	定义编译历史记录，包含编译时间、用时、结果和项目路径

关键特性：

- **类型安全：**所有字段使用 Python type hints，Pydantic 自动执行类型校验和转换
- **序列化统一：**模型内置 model_dump() 方法，统一工具层和业务层的数据序列化格式
- **向后兼容：**字段定义使用别名机制，兼容旧版本的字段命名差异

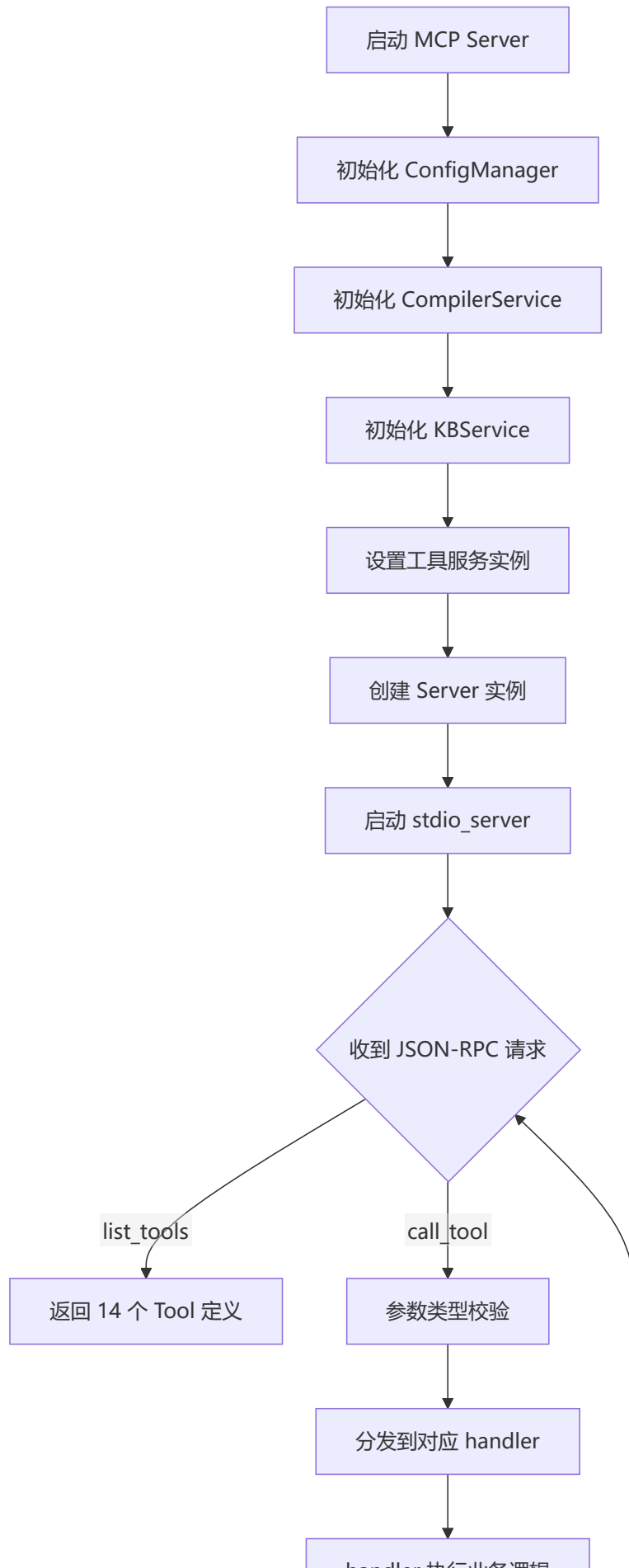
4.3 入口模块 (src/server.py + version + init, 3 文件, 1497 行)**功能概述：**

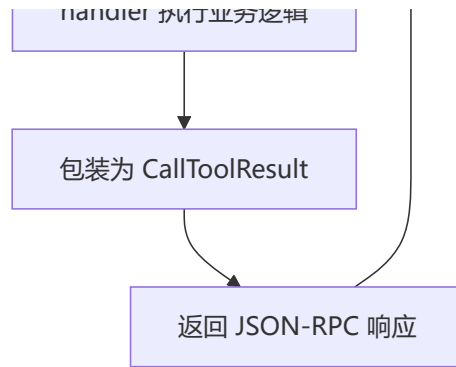
入口模块是系统的启动中心和 MCP 协议交互的总调度器。它负责创建 MCP Server 实例、初始化所有服务 (ConfigManager、CompilerService、DelphiKnowledgeBaseService、ThirdPartyKnowledgeBase)、注册 14 个 MCP 工具、实现参数类型校验的辅助函数，以及分发工具调用请求到对应的 handler。输入为 MCP 客户端通过 stdin 发送的 JSON-RPC 请求 (list_tools 或 call_tool)，输出为 Tool 列表或工具调用结果 (CallToolResult)。

核心类与函数：

核心组件	所属文件	职责
run_server()	server.py	异步主入口函数，初始化服务和 MCP Server 实例，启动 IO 事件循环
list_tools()	server.py	返回 14 个 Tool 对象的完整列表，每个对象包含名称、描述和 JSON Schema 输入定义
call_tool()	server.py	接收工具名和参数，分发到对应 handler 处理，统一包装返回 CallToolResult
_coerce_bool/int/list()	server.py	参数类型安全转换函数，防止客户端传错类型（如字符串代替布尔值）导致异常
_get_smart_hint()	server.py	智能提示生成器，在工具返回后根据上下文给出下一步操作建议

核心流程：





步骤 1: 服务初始化 系统启动后首先初始化 ConfigManager, 从 `compilers.json` 加载编译器配置。然后初始化 CompilerService, 将其与 ConfigManager 绑定。接着初始化 DelphiKnowledgeBaseService 和 ThirdPartyKnowledgeBase, 用于后续知识库查询。最后将各服务实例通过 setter 函数注入到对应的工具模块中, 使工具层可以调用业务逻辑层的实例。

步骤 2: DFM 编译器路径设置 从 ConfigManager 获取最新的 Delphi 编译器路径, 若路径合法且可执行文件存在, 则注入到 `dfm_utils` 和 `create_component_dfm` 模块中, 使 DFM 格式转换功能可用; 若文件不存在, 记录警告日志。

步骤 3: 启动 IO 事件循环 创建 `mcp.server.Server` 实例, 注册 `list_tools` 和 `call_tool` 回调, 通过 `stdio_server()` 建立标准输入/输出的 JSON-RPC 双工通信管道。此后持续等待客户端发送请求。

步骤 4: 请求分发与响应 收到 `list_tools` 请求时返回预定义的 Tool 列表。收到 `call_tool` 请求时先校验参数类型 (字符串转布尔值、字符串转整数等), 然后根据工具名称分发到对应的异步 handler 函数, handler 执行业务逻辑后返回结果。系统在返回前调用 `_get_smart_hint()` 智能提示器, 根据返回内容生成下一步操作建议 (如知识库搜索命中后提示使用 `delphi_file` 读取源码)。

关键特性:

- **子进程保护:** 检测到 `__name__ == '__mp_main__'` 时跳过所有服务导入, 避免 Windows spawn 模式下子进程重复加载全部模块 (885 个模块)
- **智能提示:** 工具返回后自动分析结果状态, 生成上下文相关的下一步操作建议, 减少用户思考时间
- **编码统一:** 启动时强制设置 `PYTHONIOENCODING=utf-8` 和 `PYTHONUTF8=1`, 确保中文路径和 Unicode 字符正确处理

4.4 业务逻辑模块 (src/services/, 21 文件, 15817 行)

功能概述:

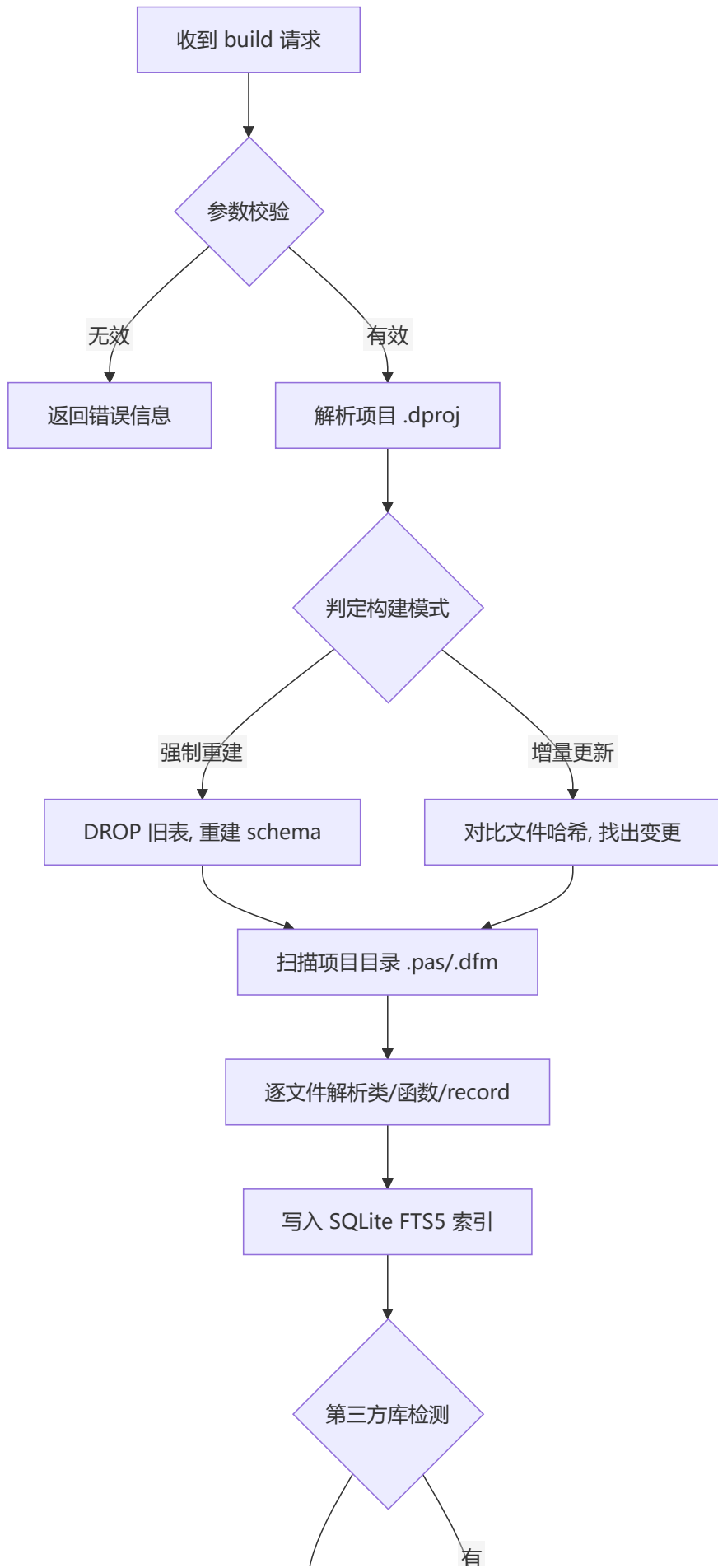
业务逻辑模块是系统的核心处理层, 包含 7 个子服务, 覆盖编译调度、知识库构建与检索、配置文件管理、子进程管理、版权文档生成、经验记忆和自动化测试。该模块接收来自工具层的参数, 调用下游的工具层能力 (如 DprojParser、DelphiEnv) 完成具体操作, 返回结构化结果。输入为经过校验的参数字典, 输出为业务处理结果。

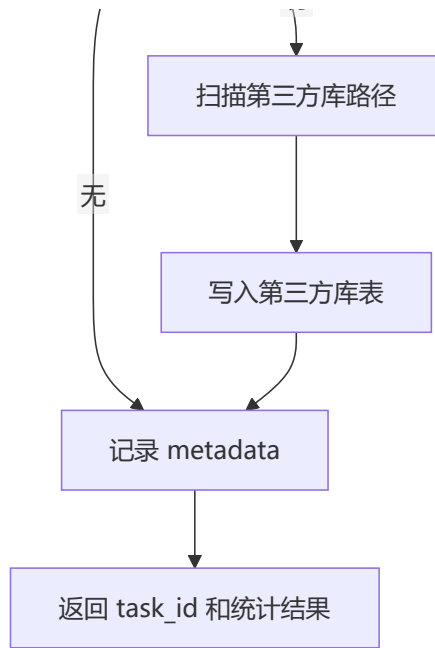
核心类与函数:

核心组件	所属文件	职责
CompilerService	<code>compiler_service.py</code>	编译调度核心, 管理编译器列表选择逻辑, 提供 <code>find/compile/detect</code> 等方法
ConfigManager	<code>config_manager.py</code>	编译器配置持久化管理, 从 JSON 文件读写编译器列表, 支持注册表检测和路径自愈
ArgsGenerator	<code>args_generator.py</code>	根据 <code>CompileRequest</code> 生成完整的 <code>dcc32</code> 命令行参数串, 处理搜索路径、条件符号和输出目录

核心组件	所属文件	职责
ProcessManager	process_manager.py	子进程生命周期管理，封装 subprocess.Popen，支持 30 秒长轮询超时时降级为短轮询
DelphiKnowledgeBaseService	knowledge_base/service.py	知识库统一入口，管理关联数据库连接，按库类型路由搜索请求到对应子模块
ProjectKnowledgeBase	knowledge_base/project_knowledge_base.py	项目级知识库，自动扫描项目目录和第三方库路径，增量更新源码索引
EmbeddingService	knowledge_base/embedding_service.py	语义向量服务，加载 ONNX 或 sentence-transformers 模型，为知识库搜索提供向量化支持
CopyrightService	copyright_service.py	软著文档自动生成，读取源码目录结构生成源代码文档、说明书和摘要，支持 mermaid PDF 渲染
AutomationService	automation_service.py	Delphi 程序 UI 自动化驱动器，通过 JSON 脚本指令 (goto/click/capture/drag/type/key) 驱动目标程序执行操作并截图
ExperienceService	experience_service.py	AI 经验记忆持久化管理，基于 SQLite 存储，支持语义去重 (相似度 > 0.85 合并)
Schema	knowledge_base/schema.py	统一 DDL 管理，所有知识库的建表语句集中在此文件，确保数据结构一致性

核心流程 — 项目知识库构建：





步骤 1: 接收构建请求 工具层调用 `delphi_kb(action="build", kb_type="project")` 后, 请求到达业务逻辑层。先校验参数完整性, 检查 `project_path` 是否有效、`.dproj` 文件是否存在。无效参数直接返回错误信息。

步骤 2: 解析项目配置 使用 `DprojParser` 解析 `.dproj` 文件, 提取编译器版本、搜索路径、第三方库路径 (`DCC_UnitSearchPath`) 等信息。解析失败时返回 XML 解析错误详情。

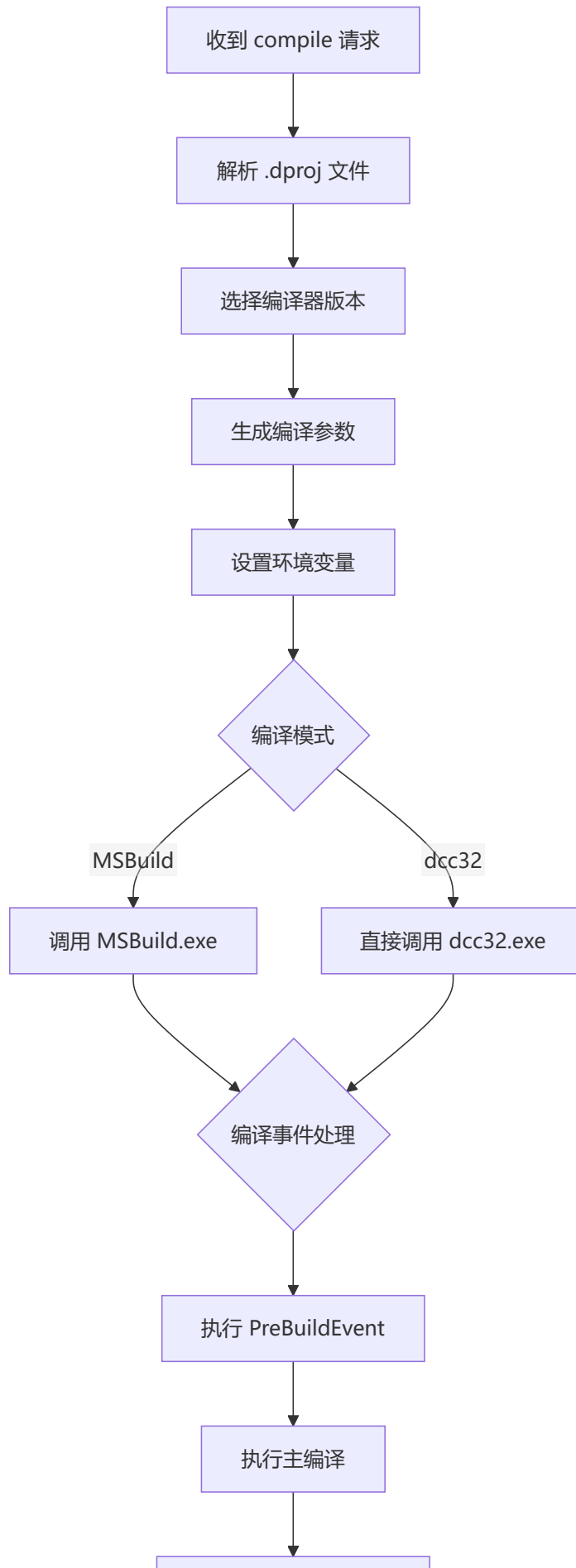
步骤 3: 判定构建模式 若 `rebuild=True` 执行强制重建, 删除现有表结构后重建 `schema`; 若 `incremental=True` 执行增量更新, 对比文件修改时间和内容哈希, 仅处理发生变更的文件。

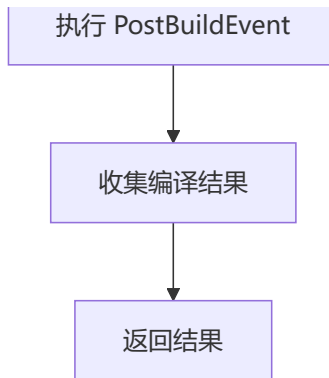
步骤 4: 扫描与索引 使用 `scan_delphi_sources.py` 中的扫描器遍历项目目录下所有 `.pas` 和 `.dfm` 文件, 提取类定义 (TClass)、函数/过程 (function/procedure)、record 定义和接口声明。结果写入 SQLite 数据库的 FTS5 全文索引表和实体表。

步骤 5: 第三方库处理 若项目配置了第三方库搜索路径, 使用 `ThirdPartyKnowledgeBase` 扫描并索引这些路径下的源码。索引完成后记录构建元数据 (文件数、类数量、函数数量和构建用时)。

步骤 6: 返回结果 返回 `task_id` 和统计信息 (处理文件数、新增实体数、构建用时)。异步模式下立即返回 `task_id`, 由客户端通过 `async_task` 轮询进度。

核心流程 — Delphi 项目编译:





步骤 1: 解析项目文件 收到编译请求后, 先解析 `.dproj` 文件提取编译器参数: 目标平台 (Win32/Win64)、配置文件 (Debug/Release)、搜索路径 (`DCC_UnitSearchPath`)、条件编译符号 (`DCC_Define`) 和编译事件脚本。

步骤 2: 选择编译器 根据项目目标平台和版本号, 从 ConfigManager 的编译器列表中选择合适的 `dcc32` (Win32) 或 `dcc64` (Win64) 可执行文件路径。若未找到匹配版本则返回错误。

步骤 3: 生成参数 ArgsGenerator 根据项目配置和版本执行环境变量替换 (如 `$(Platform)`、`$(Config)`、`$(OutDir)`) , 生成完整的编译器命令行参数串。智能过滤冗余的第三方库路径以防止命令行过长。

步骤 4: 执行编译 通过网络 MSBuild 编译 (推荐) 或直接 `dcc32` 编译。ProcessManager 启动子进程, 通过长轮询 (≤ 30 秒) 监控编译进度, 超时后自动切换短轮询模式。编译过程中执行 PreBuildEvent、PreLinkEvent 和 PostBuildEvent。

步骤 5: 返回结果 收集编译输出 (stdout/stderr)、返回码、生成文件列表和用时统计。若编译成功返回生成文件路径; 若失败返回详细的错误信息列表。

关键特性:

- **双重编译模式:** 支持 MSBuild (项目级依赖管理) 和直接 `dcc32` (单文件快速编译) 两种模式
- **智能库路径:** 自动分析项目依赖路径, 去重后按需添加, 避免命令行参数超过 Windows 限制
- **循环事件保护:** 在环境变量替换过程中检测循环引用, 防止无限递归
- **WAL 模式统一:** 所有知识库 SQLite 连接统一使用 WAL journal 模式, 避免运行中锁冲突
- **Worker print 禁令:** 多进程 worker 内部禁用 `print()`, 防止 MCP stdout 通信管道被污染

4.5 工具实现模块 (src/tools/, 28 文件, 15489 行)

功能概述:

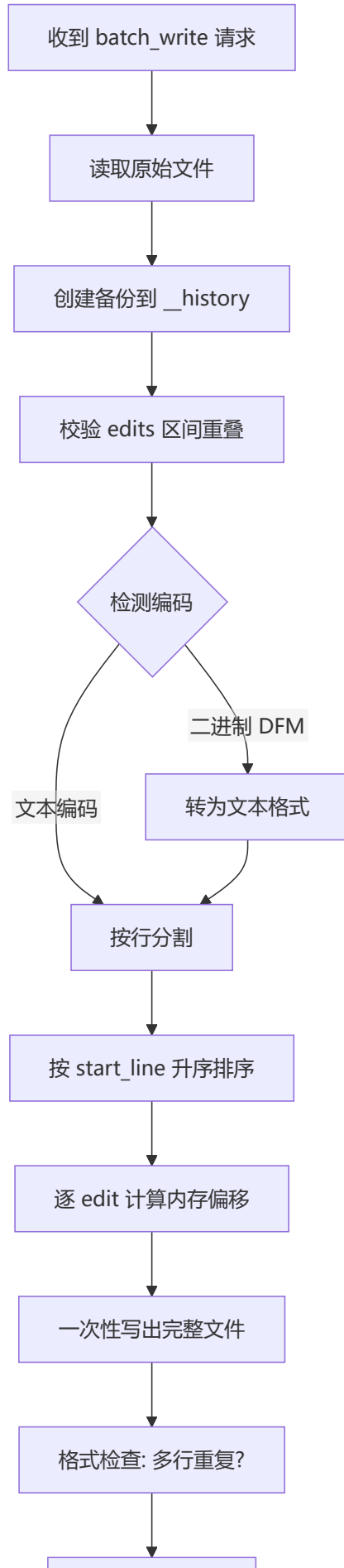
工具实现模块是系统对外的接口层, 将业务逻辑层的服务能力封装为 14 个 MCP 标准工具, 每个工具对应一个 handler 函数。该模块负责参数校验、请求路由、错误捕获和结果格式化。输入为 AI 客户端传入的参数字典, 输出为 `CallToolResult` (包含 `TextContent` 列表和 `isError` 标识)。28 个文件中包含核心工具、辅助解析器 (DFM 解析/PAS 声明解析) 和后台任务 worker。

核心类与函数:

核心组件	所属文件	职责
handle_project	project.py	编译/审计/AST/运行时检查的统一入口, 调度到 compile_project/compile_file/audit 等子模块
search_knowledge	knowledge_base.py	知识库搜索路由, 按 kb_type 分发到 Delphi/项目/第三方库/文档四个引擎
file_read/write/batch_write	file_tool.py	Delphi 文件读写核心实现, 包含编码自动检测、DFM 二进制自动转换和行号偏移计算

核心组件	所属文件	职责
manage_component	manage_component.py	DFM 组件增/删/改/查, 自动同步到 PAS 文件中的声明部分
code_hosting	code_hosting.py	五大代码托管平台 API 请求统一封装, Git 本地操作的消息格式化输出
check_environment	environment.py	编译器环境诊断, 通过注册表和文件系统检测 Delphi 安装状态和 pasfmt 可用性
handle_async_task	async_tasks.py	后台任务管理接口, 提交任务到 ProcessPoolExecutor, 返回值中附带 task_id 供轮询
handle_package	install_package.py	组件包编译安装, 调用 dcc32 编译 .dpc 文件并注册到 IDE
get_coding_rules	coding_rules.py	按 project_path 查找 CODING_RULES.mdc, 支持 section 参数分段返回
experience	experience.py	经验记忆处理器, 调用 ExperienceService 实现语义搜索、去重保存和定期清理
get_tool_help	tool_help.py	按需返回工具完整帮助文档, 减少 token 消耗
handle_generate_copyright	copyright_service.py	软著文档生成器, 支持 generate/validate/update_config 等 8 个 action
automate_delphi	automation_service.py	Delphi UI 自动化驱动器, 通过命名管道与控制程序通信, 逐条执行 JSON 脚本指令
DfmParser	dfm_parser.py	DFM 文件到 JSON 的结构化解析器, 支持签名识别和事件绑定分析
PasDeclParser	pas_decl_parser.py	PAS 文件类型声明提取器, 用于 manage_component 的 PAS 同步
ProjectKBWorker	project_kb_worker.py	项目知识库构建的 multiprocessing worker, 内部禁用 print 保护 MCP 通信

核心流程 — Delphi 文件批量编辑:



返回 diff 预览

步骤 1: 校验与备份 收到 `batch_write` 请求后先读取原始文件内容, 创建备份到 `__history/` 目录 (版本号累加)。校验 `edits` 列表中各编辑区间是否重叠 (相邻 `edit` 区间可以是连续的, 但不能交叉)。

步骤 2: 编码检测与格式转换 自动检测文件编码 (BOM 检测 → `utf-8/utf-16/gbk` 判定)。对于二进制 DFM 文件, 先转换为文本格式, 编辑完成后再转回二进制。

步骤 3: 合并编辑与检查 将 `edits` 按 `start_line` 升序排列后, 按行操作, 每次操作后累加偏移量。最终一次性写出完整文件。写入后做格式检查: 若新内容中存在连续相同的行 (可能因行号偏移误算导致), 输出警告文本, 提示 AI 核对 diff。

步骤 4: 输出结果 返回每处编辑的前后行范围变换 (如 `[5, 10) → [5, 13)`), 附带 `- / + diff` 预览, 帮助 AI 确认编辑准确性。

关键特性:

- **编码自动检测:** 支持 UTF-8、UTF-16 LE、GBK 等多种编码, 读取时自动识别, 写入时保持原始编码
- **备份管理:** 每次写入自动备份, 支持 `backup(action=list)` 查看历史版本和 `backup(action=restore)` 恢复指定版本
- **DFM 透明处理:** 二进制 DFM 自动转为文本编辑, 编辑完成后自动转回二进制, 对 AI 完全透明
- **RWLock 并发保护:** 所有文件操作使用读写锁, 并发写入返回错误并引导使用 `batch_write` 合并

4.6 工具类模块 (src/utils/, 12 文件, 3399 行)**功能概述:**

工具类模块提供跨模块共享的基础能力, 不依赖任何业务逻辑或工具实现模块。包含 Delphi 环境检测 (通过注册表读取 BDS 版本路径)、DPROJ XML 文件解析、路径安全性校验、编码检测与文件备份、日志初始化、单元依赖分析等功能。该模块被业务逻辑层和工具层共同引用, 是系统最底层的支撑模块。

核心类与函数:

核心组件	所属文件	职责
<code>get_delphi_version / get_delphi_path</code>	<code>delphi_env.py</code>	从注册表 HKLM/HKCU 读取 Embarcadero BDS 键, 枚举已安装的 Delphi 版本路径
<code>DprojParser</code>	<code>dproj_parser.py</code>	解析 <code>.dproj</code> XML 文件, 提取 <code>MSBuild</code> 属性组、编译事件和搜索路径列表
<code>Validator</code>	<code>validator.py</code>	验证项目文件路径可访问性和编码合法性, 统一返回 (是否有效, 错误信息) 元组
<code>init_default_logger / log_api_call</code>	<code>logger.py</code>	初始化 logging 配置, 提供 API 调用的结构化日志记录和性能计时
<code>FileBackup</code>	<code>file_backup.py</code>	文件版本管理, 自动创建带版本号后缀 (<code>~1~</code> , <code>~2~</code>) 的备份, 支持列表和恢复
<code>Parser (encoding detect)</code>	<code>parser.py</code>	检测文件编码 (BOM 自动识别), 支持 <code>utf-8/utf-16-le/utf-16/gbk/cp950</code> 等
<code>PathValidator</code>	<code>path_validator.py</code>	路径安全性检查, 防止路径遍历攻击, 限制文件操作在许可目录范围内
<code>ProgressTracker</code>	<code>progress_tracker.py</code>	后台任务进度跟踪器, 提供增量百分比计算和多阶段进度聚合
<code>UnitDependencyAnalyzer</code>	<code>unit_dependency_analyzer.py</code>	分析 PAS 文件中的 <code>uses</code> 依赖关系, 检测循环引用和缺失单元

核心组件	所属文件	职责
check_update	updater.py	通过 GitHub API 远程检查新版本，缓存检查结果避免高频请求

关键特性:

- **注册表安全读取**: delphi_env.py 仅读取 HKLM/HKCU 下 Embarcadero BDS 路径，不写入任何注册表内容
- **XML 命名空间感知**: DprojParser 正确处理 MSBuild XML 命名空间
`http://schemas.microsoft.com/developer/msbuild/2003`
- **结构化日志**: logger.py 提供统一的 API 调用日志格式，包含工具名、参数长度和耗时，便于调试和性能分析
- **备份版本管理**: FileBackup 自动在 `_history/` 目录维护历史版本，支持按版本号递增，与 delphi_file 备份操作一致

第五章 安装与配置

5.1 系统要求

- **操作系统**: Windows 7 SP1 及以上 (Windows 10/11 推荐), 或 Windows Server 2012 R2 及以上
- **Python 运行时**: 3.10、3.11、3.12 或 3.13 (64 位推荐)
- **浏览器 (PDF 生成用)**: Microsoft Edge 或 Google Chrome 最新稳定版 (用于软著文档 PDF 的 mermaid 图表渲染)
- **Delphi 编译器 (可选)**: Embarcadero Delphi 2005 至 Delphi 13 任一版本 (不安装时系统只能使用知识库和文件操作类功能)
- **7-Zip (可选)**: 7-Zip 18.0 及以上, 用于解压 Delphi CHM 帮助文档以构建文档知识库
- **Git (可选)**: Git 2.30 及以上, 用于版本更新和代码托管工具

5.2 安装方式

系统提供两种安装方式:

- **pip 安装 (推荐)**: 执行 `pip install daofy-for-delphi` 直接安装。安装完成后在 AI 客户端 MCP 配置文件中添加 daofy 条目并指定命令为 `daofy` 即可。
- **源码安装**: 执行 `git clone https://github.com/chinawsb/daofy.git`, 创建虚拟环境后执行 `pip install -r requirements.txt`, 然后在 AI 客户端配置中将启动命令指向 `src/server.py`。

安装后验证: 启动 AI 客户端, 调用 `check_environment(action="check")` 查看编译器检测状态; 调用 `delphi_kb(action="stats")` 查看知识库统计信息。

5.3 配置方法

系统的核心配置文件为 `config/compiler.json` (JSON 格式), 记录已安装的 Delphi 编译器信息。每个编译器条目包含版本号 (如 "23.0")、可执行文件路径、平台 (Win32/Win64) 和搜索路径列表。首次使用时, 系统自动通过 Windows 注册表检测已安装的 Delphi 版本并写入此文件。如需手动添加自定义编译器, 可编辑此文件添加条目, 或使用 `check_environment(action="detect", search_path=...)` 指定搜索路径重新检测。

客户端 MCP 配置需在 AI 助手配置文件中添加 daofy 条目，指定命令和路径（具体取决于 pip 安装或源码安装），并设置环境变量 `PYTHONIOENCODING=utf-8` 和 `PYTHONUTF8=1` 确保编码正确。

第六章 操作说明

6.1 操作模式概述

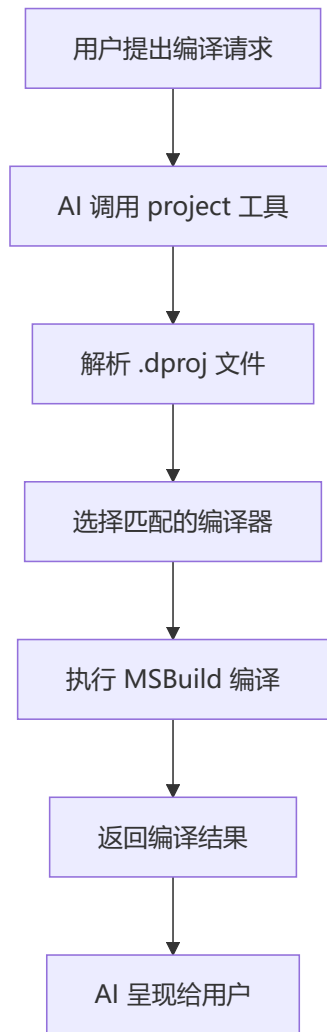
系统以 MCP Server 方式运行，无图形界面，通过标准输入/输出与 AI 客户端建立 JSON-RPC 双工通信。用户在 AI 助手的对话界面中以自然语言描述需求，AI 分析后自动调用对应的 MCP 工具（如 `project(action="compile")`）并传入参数。系统处理后返回结构化结果（文本或错误信息），AI 将结果转化为自然语言反馈给用户。整个交互过程对用户透明——用户只需描述“编译这个项目”或“搜索 TStringList 的用法”，无需记忆工具名和参数。

6.2 操作示例

示例 1：编译 Delphi 项目

操作描述：用户在 AI 助手说“帮我编译项目 D:\Projects\App1\App1.dproj，Release 配置，输出到 D:\Build\out”。

系统响应通过以下流程完成编译：



步骤 1: AI 调用 project 工具 AI 分析用户指令后, 调用 `project(action="compile", project_path="D:\\Projects\\App1\\App1.dproj", build_configuration="Release", output_dir="D:\\Build\\out")` 工具。系统进入业务逻辑模块的编译流程。

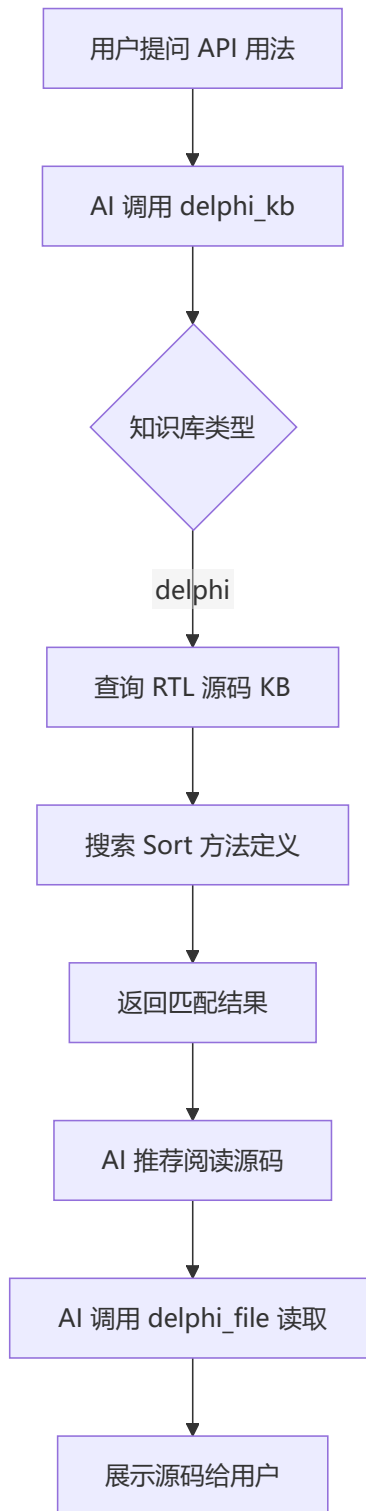
步骤 2: 解析项目配置 CompilerService 调用 DprojParser 解析 `.dproj` 文件, 提取目标平台 (如 Win64)、搜索路径 (包含 VCL、RTL 和第三方库路径)、条件编译符号 (如 RELEASE) 和编译事件脚本。解析失败时返回 XML 格式错误详情。

步骤 3: 选择编译器并执行编译 ArgsGenerator 根据项目配置生成 MSBuild 命令行参数, ProcessManager 启动 MSBuild.exe 子进程。系统以 30 秒长轮询模式等待编译完成, 每 5 秒检查一次进程输出。若检测到 PreBuildEvent, 在编译前先执行该脚本中的命令。编译过程中实时收集 stdout/stderr 输出。

步骤 4: 返回编译结果 编译完成后返回 CompileResult, 包含返回码 (0 表示成功)、编译用时 (秒)、所有输出消息列表和生成的可执行文件路径。AI 将结果呈现给用户: "编译成功! 用时 12.3 秒, 生成文件: D:\\Build\\out\\App1.exe"。

示例 2: 搜索 Delphi 知识库

操作描述: 用户在 AI 助手中询问 "TStringList 的 Sort 方法怎么用?"



步骤 1: AI 调用 delphi_kb 搜索 AI 调用 `delphi_kb(action="search", query="TStringList.Sort", search_type="method", kb_type="delphi")`。系统在 Delphi RTL 源码知识库中搜索匹配的类和方法定义。

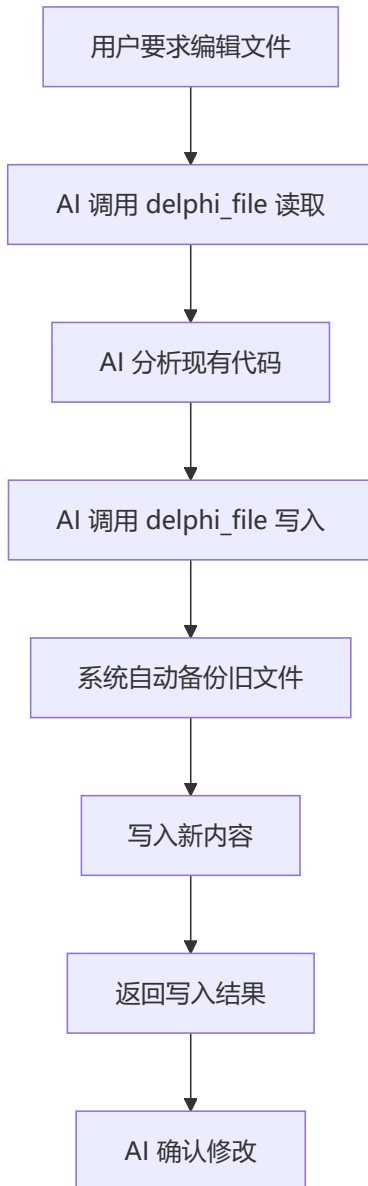
步骤 2: 知识库返回结果 知识库引擎在 FTS5 索引中全文搜索，返回命中记录的列表（包括类名、方法签名、所在文件和行号）。若同时满足语义搜索条件，还返回语义相似度评分。命中结果按匹配度降序排列。

步骤 3: AI 进一步读取源码 知识库返回的智能提示建议 AI 使用 `delphi_file` 读取完整源码。AI 接受建议，调用 `delphi_file(action="read", file_path="System.Classes.pas", search_type="function", function_name="Sort")` 读取 Sort 方法的实现代码。

步骤 4: 展示给用户 AI 将读取到的源码和方法说明整理后反馈给用户: "TStringList.Sort 在 System.Classes.pas 第 12345 行定义, 原型为 `procedure Sort;`, 调用字符串比较函数进行快速排序。需要自定义排序规则时请使用 `CustomSort` 方法。"附带源码片段。

示例 3: 编辑 Delphi 文件

操作描述: 用户要求"给 Unit1.pas 的 TForm1 类增加一个 Button1 点击事件, 弹出一个消息框"。



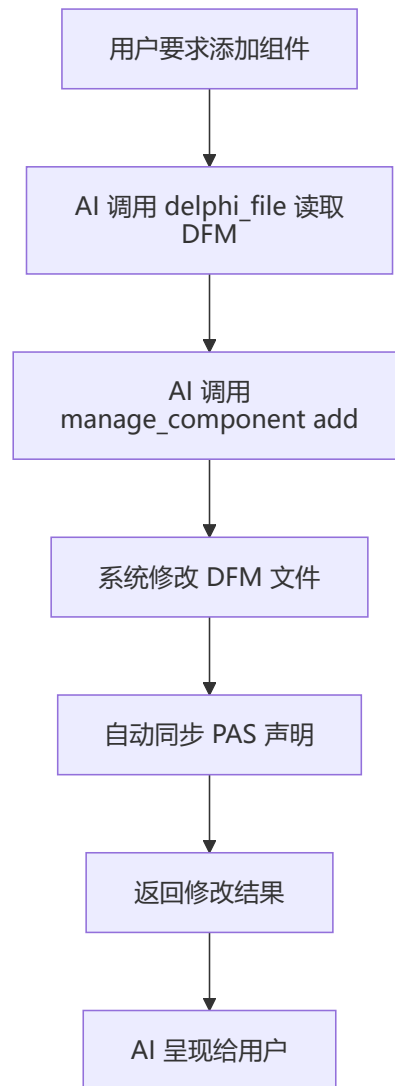
步骤 1: AI 读取当前文件 AI 调用 `delphi_file(action="read", file_path="Unit1.pas", search_type="class", type_name="TForm1")` 读取当前 TForm1 类的声明和实现。系统返回该类的完整定义, 包含已声明的字段和方法。

步骤 2: AI 分析并计算修改 AI 分析现有代码确定插入位置: 在 TForm1 类的 `published` 段增加 `Button1: TButton` 字段声明, 在 `implementation` 段增加 `procedure TForm1.Button1Click(Sender: TObject);` 事件处理实现。AI 使用 `delphi_file(action="batch_write", ...)` 一次性完成多处编辑, 系统自动备份旧文件到 `__history/`。

步骤 3: 写入并验证 系统写入完成后返回 `diff` 预览, AI 核对无误后呈现给用户: "已添加 Button1 按钮, 点击弹出消息框。文件已自动备份。"用户可随时通过 `delphi_file(action="backup", backup_action="list", file_path="Unit1.pas")` 查看备份版本并恢复。

示例 4: DFM 组件管理

操作描述：用户在 AI 助手中要求“在 MainForm 上添加一个 TEdit 控件，Name 为 EdSearch，放在 Panel1 上面”。



步骤 1: AI 读取 DFM 确定位置 AI 先调用 `delphi_file(action="read", file_path="MainForm.dfm")` 读取窗体描述，找到 Panel1 组件的定义内容和位置偏移。

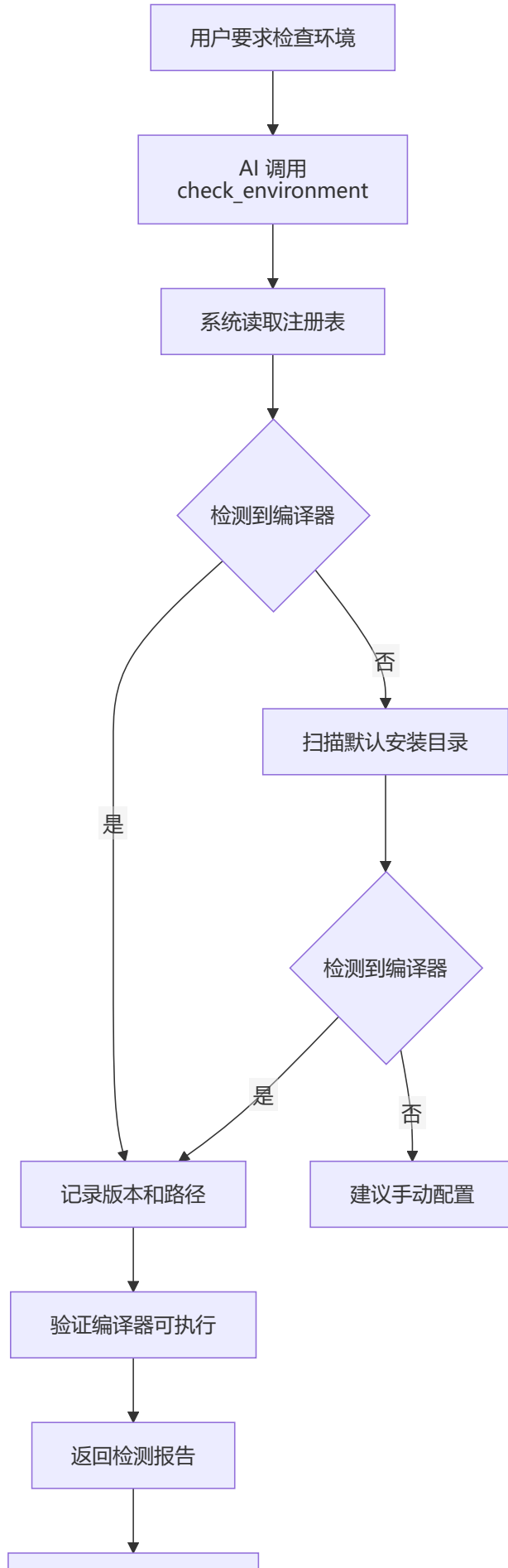
步骤 2: AI 调用 manage_component 添加组件 AI 调用 `manage_component(action="add", target_dfm="MainForm.dfm", target_pas="MainForm.pas", parent_name="Panel1", new_component_class="TEdit", new_component_name="EdSearch", properties={"Text": "", "Left": "10", "Top": "10", "Width": "200"})`。系统自动在 Panel1 下添加 TEdit 组件的 DFM 描述。

步骤 3: 自动同步 PAS 声明 系统自动在 MainForm.pas 的 TMainForm 类的 published 段添加 `EdSearch: TEdit;` 字段声明，确保编译通过。若 PAS 文件不存在或声明段无法定位，仅修改 DFM 并给出提示。

步骤 4: 返回结果 系统返回新组件已添加的确认信息和 PAS 同步结果。AI 呈现给用户：“已在 MainForm.Panel1 上添加 TEdit (EdSearch)，并已自动同步 PAS 声明。”。

示例 5: 环境诊断

操作描述：用户首次使用系统，说“检查一下 Delphi 环境”。



AI 呈现结果给用户

步骤 1: AI 调用 check_environment AI 调用 `check_environment(action="check")`。系统启动环境诊断流程, 首先尝试从 Windows 注册表 HKLM/HKCU 下 `SOFTWARE\Embarcadero\BDS` 键读取已安装的 Delphi 版本信息。

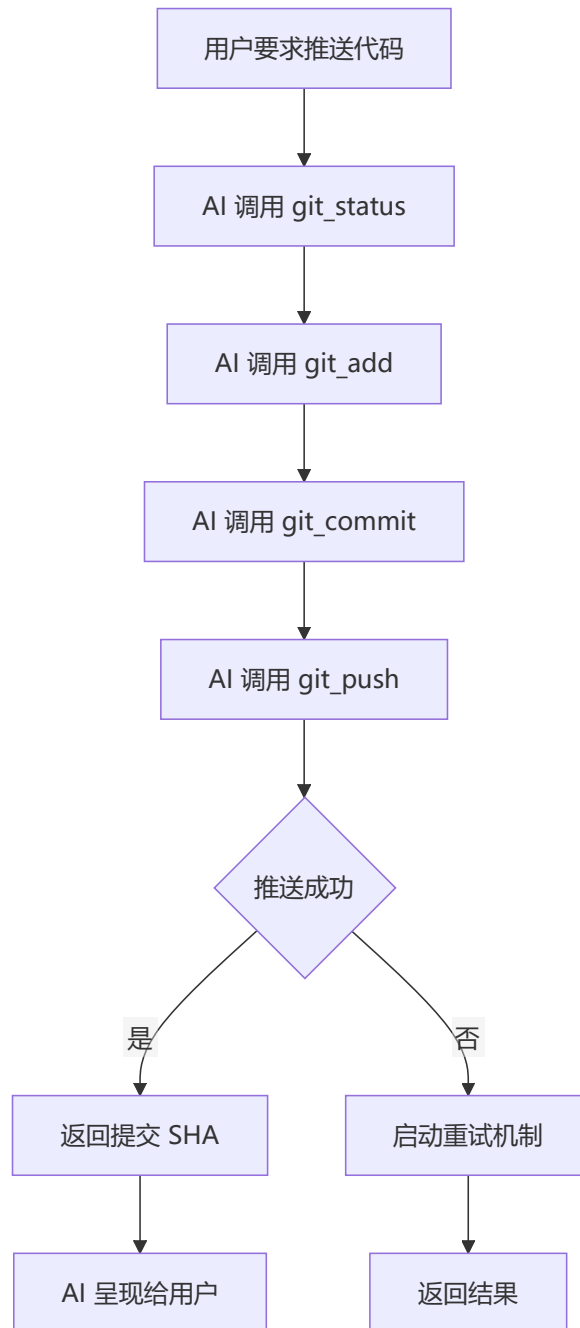
步骤 2: 注册表检测与回退 若注册表读取成功, 枚举所有版本号子键, 按版本号降序排列。对每个版本读取 `RootDir` 值, 拼接出 `dcc32.exe` 的完整路径。若注册表读取失败 (如权限不足或未安装 Delphi), 系统自动回退到默认安装目录列表 (`C:\Program Files (x86)\Embarcadero\Studio\`) 继续搜索。

步骤 3: 验证编译器可执行 对每个找到的编译器路径验证文件是否存在、是否为可执行文件、能否执行 `dcc32 --version` 或 `dcc32 -h` 命令。仅报告通过验证的编译器。

步骤 4: 返回诊断报告 系统返回检测到的编译器列表 (每个包含版本号、路径、平台), 以及 `pasfmt` 格式化工具的安装状态。若未检测到任何编译器, 返回提示信息: "未检测到 Delphi 编译器, 请确认 Delphi 已安装, 或使用 `check_environment(action='detect', search_path='...')` 手动指定搜索路径。" AI 将报告呈现给用户, 并视情况给出安装建议。

示例 6: 代码托管操作

操作描述: 用户在 AI 助手说"把我的项目提交并推送到 GitHub"。



步骤 1: 查看当前状态 AI 调用 `code_hosting(action="git_status", dir="D:\\Projects\\MyApp")` 查看工作区状态。系统返回当前分支名、未跟踪文件列表、已修改文件列表和暂存区状态。

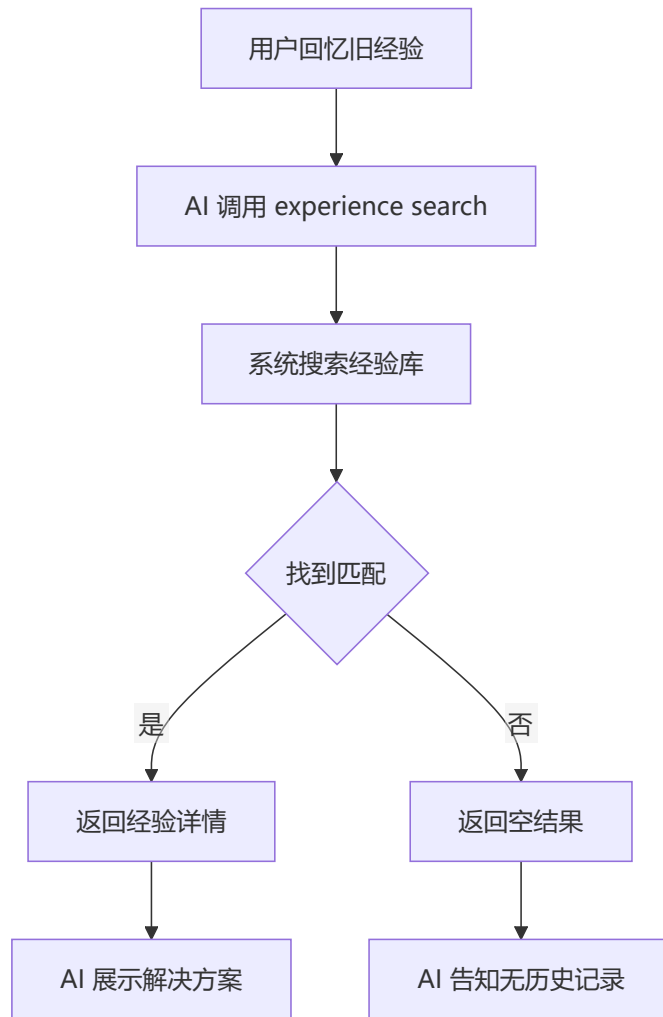
步骤 2: 暂存和提交 AI 根据状态选择合适的文件加入版本控制，调用 `code_hosting(action="git_add", dir="D:\\Projects\\MyApp", files=[".", "src/", "tests/"])` 暂存更改，然后调用 `code_hosting(action="git_commit", dir="D:\\Projects\\MyApp", message="feat: 添加用户登录功能")` 提交代码。系统返回提交的 SHA 值。

步骤 3: 推送远程 AI 调用 `code_hosting(action="git_push", dir="D:\\Projects\\MyApp", remote="origin", branch="main")` 推送到 GitHub。系统返回推送结果，包含推送的提交数量和远程分支信息。

步骤 4: 处理推送结果 若推送成功，AI 呈现提交哈希和推送摘要给用户。若推送失败（如网络超时），系统自动返回详细的错误信息，AI 可选择调用 `git_push_retry` 启动异步重试机制（默认 300 秒间隔），并告知用户重试状态。

示例 7: 经验记忆管理

操作描述: 用户指出"上次解决过 StringGrid 排序的问题，再查一下怎么处理的"。



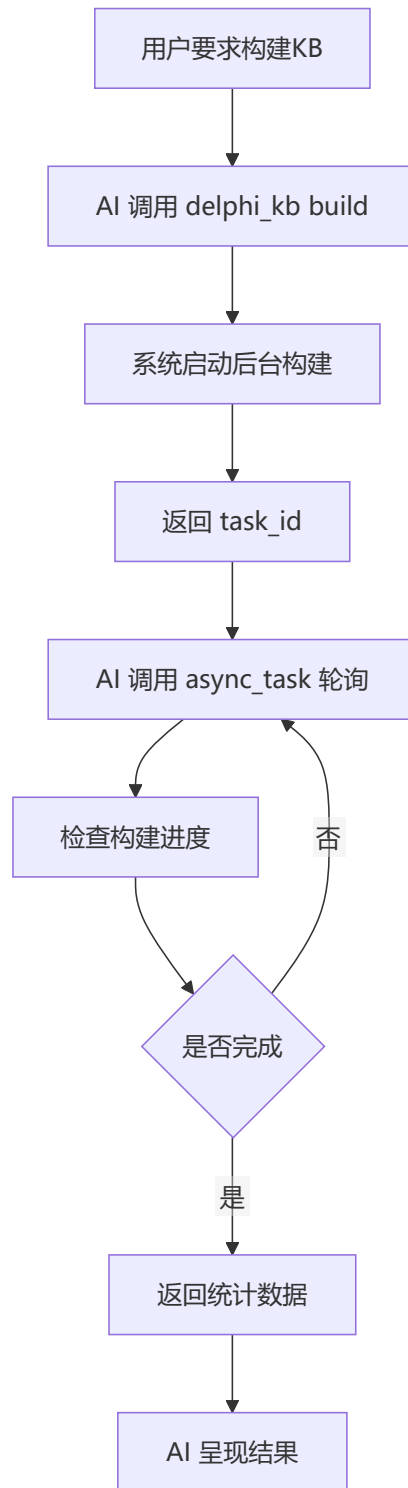
步骤 1: AI 搜索经验库 AI 调用 `experience(action="search", query="StringGrid 排序", top_k=3)`。系统在经验知识库中执行语义搜索，基于向量 `embedding` 计算相似度，返回匹配度最高的经验记录列表。

步骤 2: 经验匹配与展示 若找到匹配记录（相似度 > 0.7 ），系统返回经验详情，包含问题描述、解决步骤和用到的工具列表。AI 将方案呈现给用户："找到您之前处理 StringGrid 排序的经验：在 OnCompare 事件中实现自定义排序逻辑，调用 `exchange` 方法交换行。"

步骤 3: 经验保存（可选延伸） 若用户本次解决了新问题，AI 可调用 `experience(action="save", problem="...", solution="...", tags=["StringGrid", "排序"])` 将当前方案存入经验库。系统自动检测与已有经验的相似度，若 > 0.85 则合并到旧记录而非新增。

示例 8: 异步知识库构建

操作描述：用户说"帮我构建项目的知识库"。



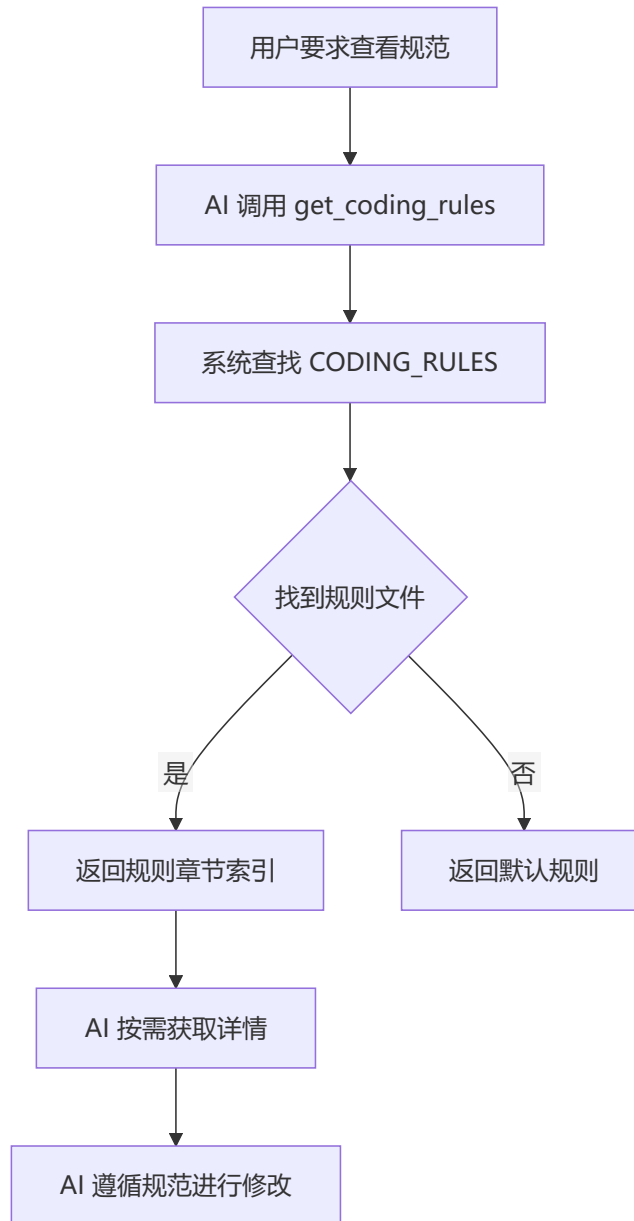
步骤 1: AI 提交构建任务 AI 调用 `delphi_kb(action="build", kb_type="project", project_path="D:\\Projects\\App1\\App1.dproj", async_mode=true)`。系统启动后台知识库构建任务，立即返回 `task_id` 和初始状态 "pending"。

步骤 2: AI 轮询任务进度 AI 调用 `async_task(action="status", task_id="task_kb_001", long_poll_seconds=10)`。系统以 10 秒长轮询等待构建完成，超时后降级为短轮询，每次返回当前进度。AI 多次调用直到状态变为 "completed"。

步骤 3: 获取结果并验证 构建完成后，AI 调用 `async_task(action="result", task_id="task_kb_001")` 获取统计数据，再调用 `delphi_kb(action="stats")` 验证知识库状态，确认搜索可用后告知用户。

示例 9：编码规则查询

操作描述：AI 准备修改 Delphi 代码前，用户说"先查一下项目的编码规范"。

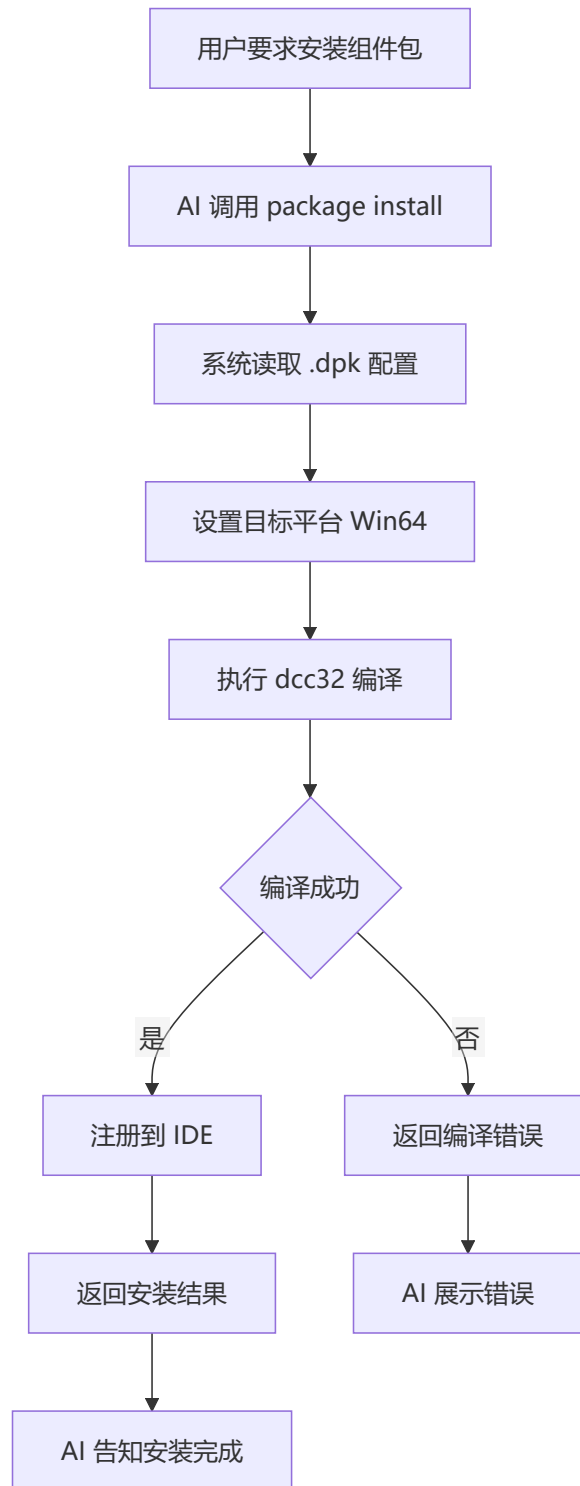


步骤 1：获取规则索引 AI 调用 `get_coding_rules(project_path="D:\\Projects\\App1")`。系统在项目目录下查找 `CODING_RULES.md` 文件，若找到则返回规则文档的章节索引；若未找到则返回内置默认规则。

步骤 2：按需获取详细规则 AI 根据索引选择需要的章节，调用 `get_coding_rules(project_path="D:\\Projects\\App1", section="writing")` 获取书写规范详细内容。AI 严格按规范进行后续代码修改。

示例 10：组件包编译安装

操作描述：用户说"帮我编译并安装这个组件包 MyPackage.dpk, 64位 Release"。

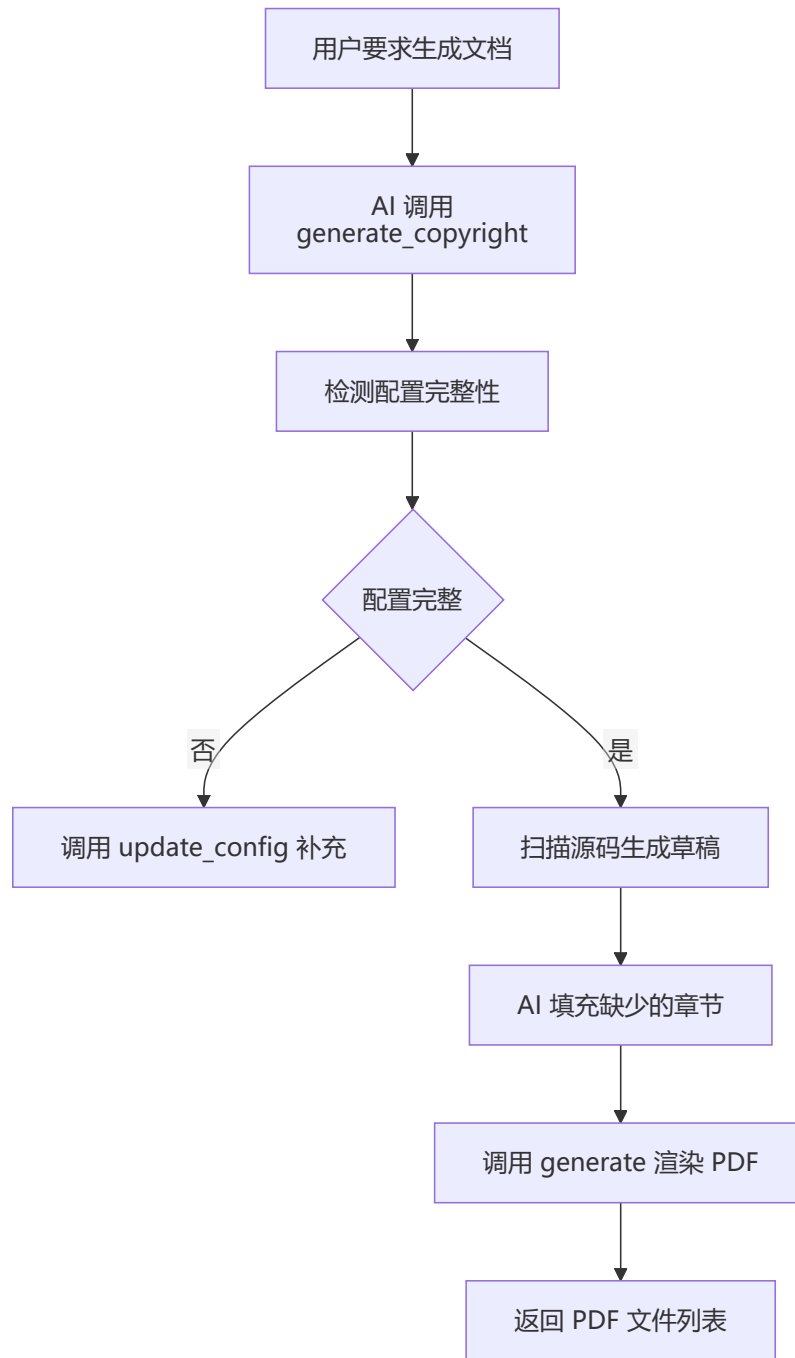


步骤 1: AI 调用 package 工具 AI 调用 `package(action="install", package_path="D:\\Components\\MyPackage.dpk", target_platform="win64", build_configuration="Release", install=true)`。系统读取 .dpk 文件配置，解析源文件列表和依赖关系。

步骤 2: 编译与注册 系统调用 dcc32 编译 .dpk 文件。若编译成功且 `install=true`，自动将 .bpl 注册到 RAD Studio IDE。若编译失败，返回详细的编译器错误信息列表供用户修正。

示例 11: 软著文档自动生成

操作描述: 用户说"我要申请软件著作权, 帮我生成申请文档"。

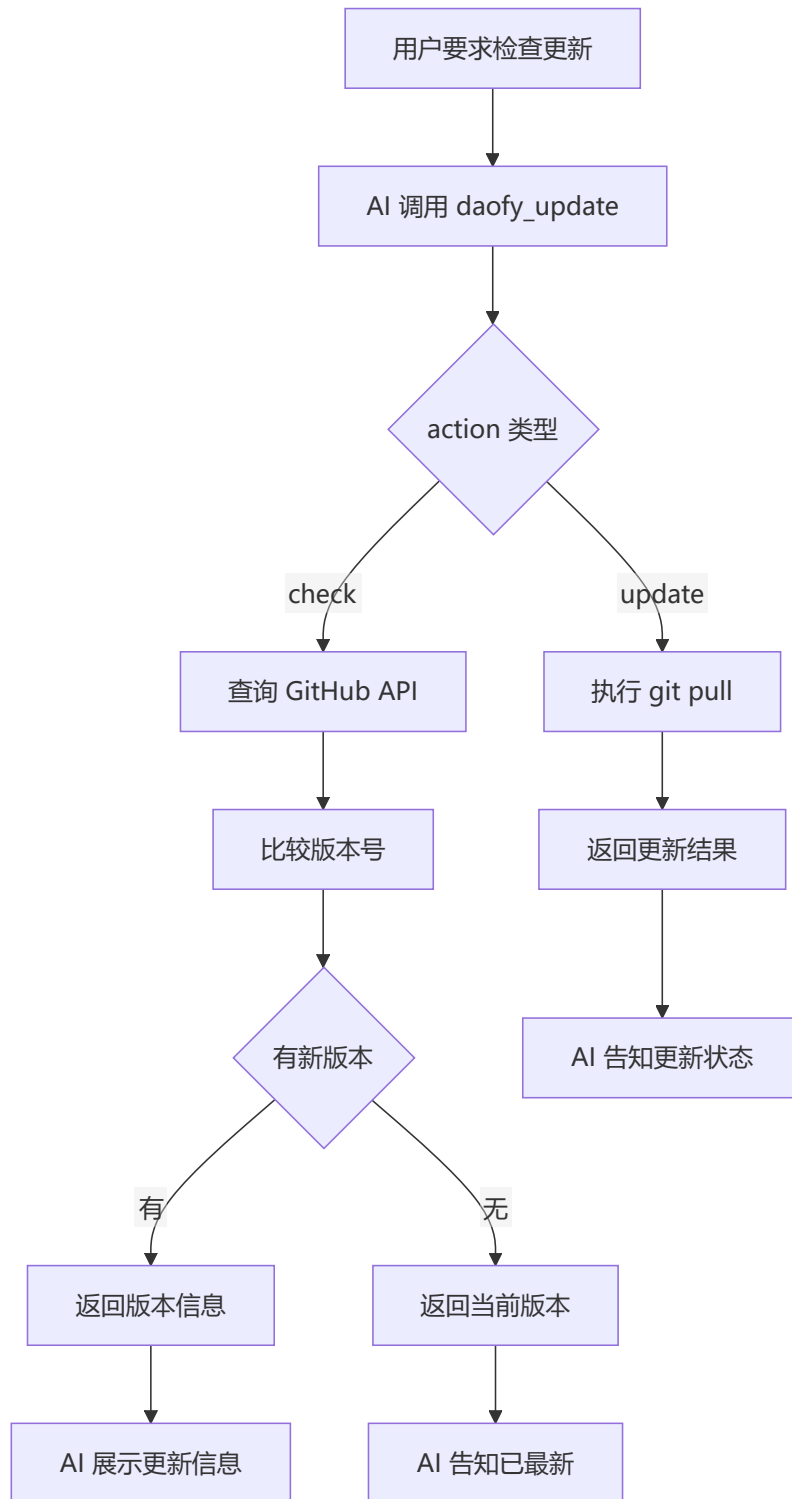


步骤 1：检查配置并生成草稿 AI 调用 `generate_copyright(action="validate")` 检查著作权配置完整性。若缺少字段则调用 `generate_copyright(action="update_config", config={...})` 补充。配置完整后调用 `generate_copyright(action="generate_content")` 扫描源码生成 markdown 草稿。

步骤 2：AI 填充内容并渲染 AI 读取生成的草稿文件，按每章 `<!-- 生成要求 -->` 填充正文。填充完成后调用 `generate_copyright(action="generate")` 由 Edge 浏览器渲染为三份 PDF：源代码文档、软件说明书和申请信息汇总表，输出到 `docs/copyright/` 目录。

示例 12：版本更新检查

操作描述：用户说“检查一下道飞有没有新版本”。

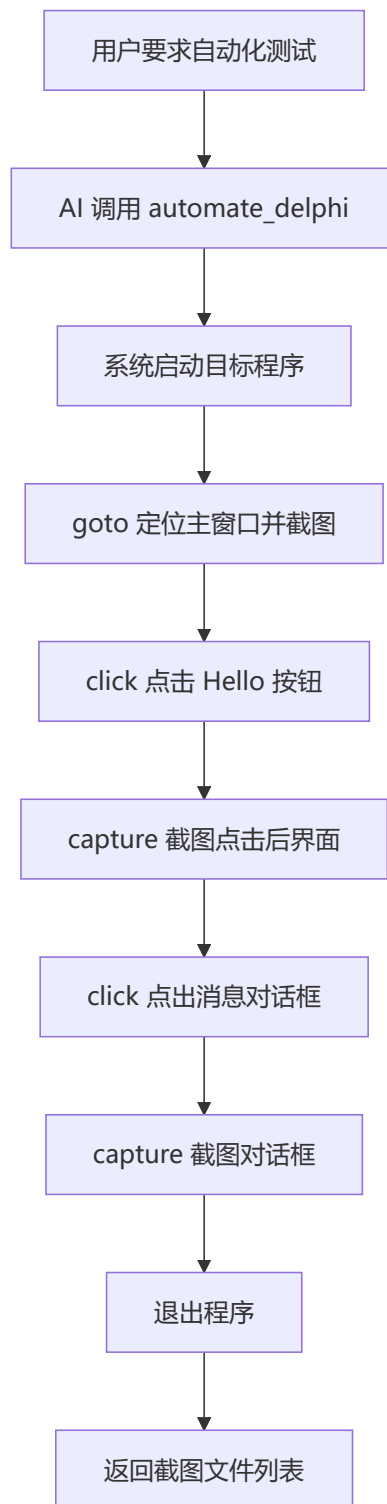


步骤 1：检查新版本 AI 调用 `daofy_update(action="check")`。系统通过 GitHub API 查询远程仓库最新版本标签，与本地版本号比较。若有新版本，返回版本号和发布时间。

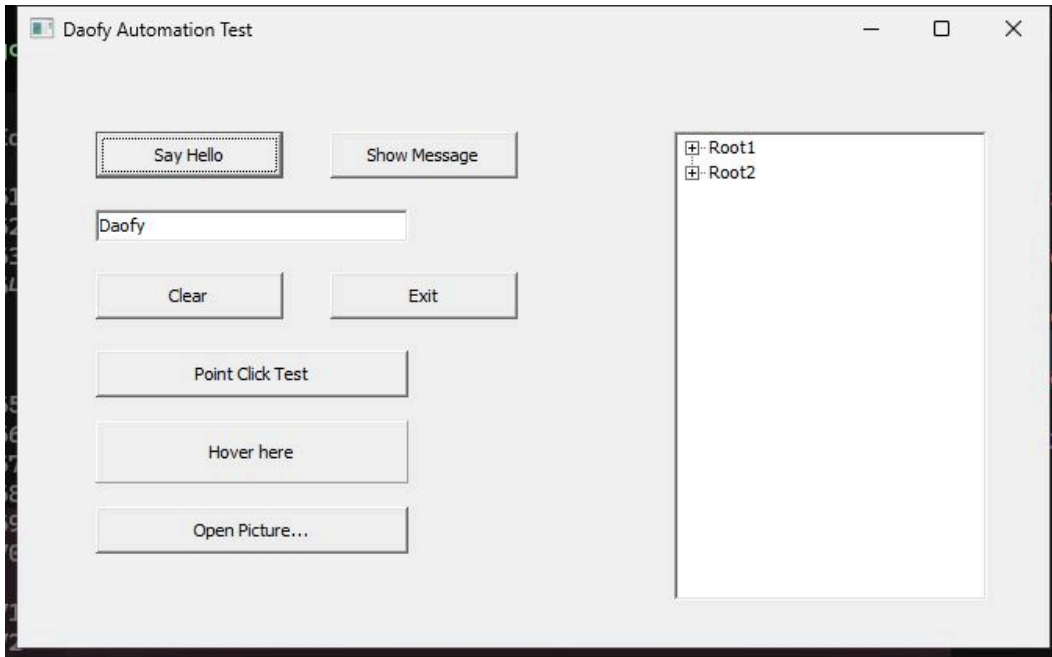
步骤 2：执行更新 用户同意后，AI 调用 `daofy_update(action="update")` 执行 git pull。系统返回更新结果并建议重启 MCP Server。

示例 13：Delphi 自动化测试

操作描述：用户说“自动化测试这个 Delphi VCL 程序，启动后截图主界面，点击 Hello 按钮后截图，弹出对话框也截图”。

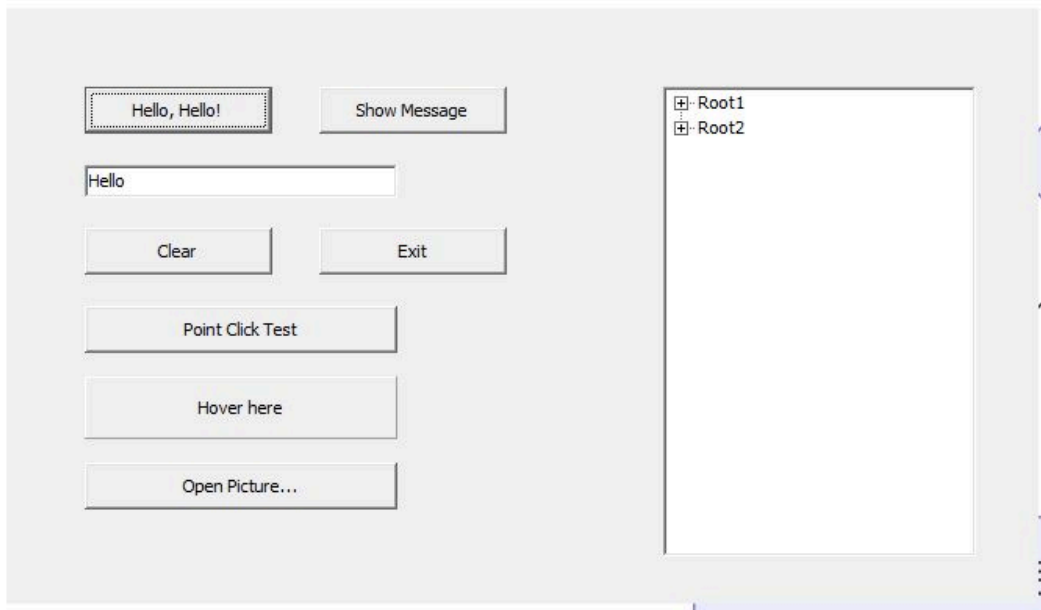


步骤 1：启动程序并捕获主界面 AI 调用 `automate_delphi(app_path="AutomateTestApp.exe", script=[{"cmd":"goto","target":"TMainForm","capture":"VCL_main"}])`。系统启动 Delphi 测试程序，通过命名管道等待主窗口就绪，定位到 TMainForm 窗口后截图。



VCL 测试程序主界面，包含 Hello 按钮、编辑框和菜单栏

步骤 2：点击按钮并截图 AI 追加脚本指令 `{"cmd": "click", "target": "TMainForm.btnHello"}`, `{"cmd": "capture", "capture": "after_hello"}`。系统定位到 btnHello 按钮控件，执行鼠标点击操作。按钮点击后界面发生变化（如按钮文本改变或标签更新），截图保存点击后的界面状态。



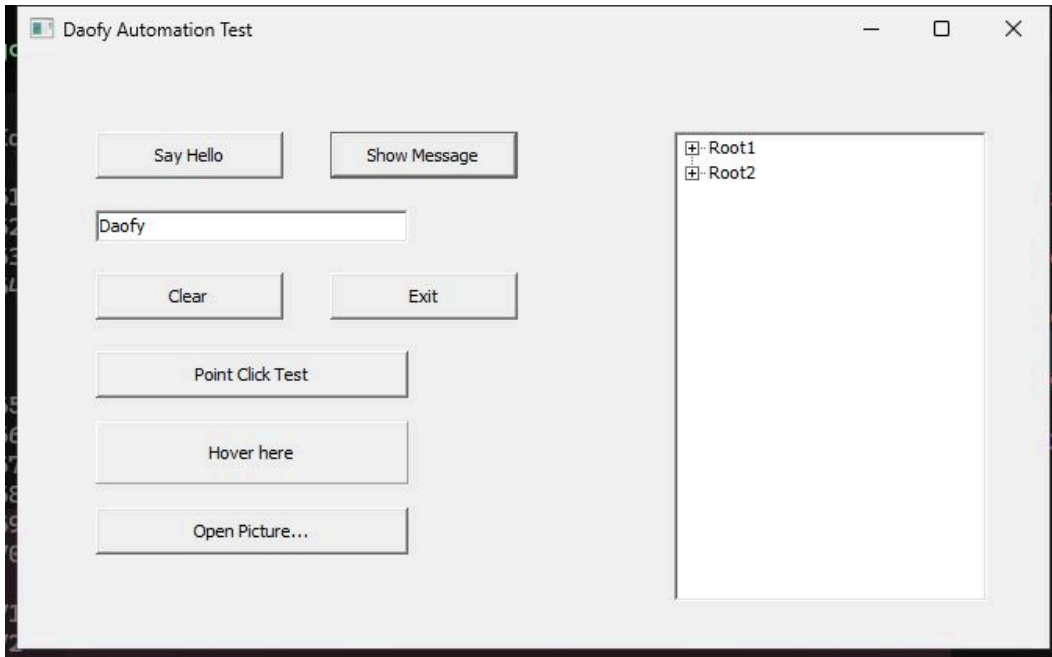
点击 Hello 按钮后，按钮文本变为“Hello Daofy!”，编辑框显示问候语

步骤 3：弹出消息对话框并截图 AI 追加脚本 `{"cmd": "click", "target": "TMainForm.btnMsg"}`, `{"cmd": "wait", "ms": 500}`, `{"cmd": "capture", "capture": "VCL_msgbox"}`。系统点击消息按钮触发 ShowMessage 对话框，等待 500 毫秒确保对话框完全弹出后截图。



弹出的消息对话框，显示提示信息

步骤 4：关闭弹窗并退出 AI 追加脚本 `{"cmd": "dlgclick", "caption": "确定", "capture": "after_msgbox"}, {"cmd": "exit"}`。系统自动识别并点击对话框中的“确定”按钮关闭弹窗，截图关闭后的界面状态，最后退出程序。所有截图的文件路径返回给 AI，AI 将截图嵌入到操作说明中供用户查看。



关闭消息对话框后，界面恢复初始状态

第七章 测试与验收

7.1 测试方案

系统使用 `pytest` 作为测试框架，采用 Python 3.10+ 标准编写测试用例。测试范围覆盖单元测试（各服务的核心函数逻辑校验，如编译器参数生成、知识库搜索路由、文件编码检测）、集成测试（多模块协作场景，如编译→文件读取→知识库查询的完整链路）和接口兼容性测试（模拟 MCP 协议调用验证工具响应格式）。共 45 个测试文件，覆盖配置管理、编译器服务、知识库构建与搜索、文件操作、DPROJ 解析、工具帮助、经验记忆和异步任务管理等功能模块。

7.2 验收标准

编号	验收项	验收标准
1	编译器自动检测	系统启动后调用 <code>check_environment(action="check")</code> 能正确返回已安装 Delphi 编译器的版本号和路径
2	项目编译功能	调用 <code>project(action="compile")</code> 编译 <code>.dproj</code> 文件, 返回码为 0, 生成的可执行文件存在于输出目录
3	编码规则查询	调用 <code>get_coding_rules()</code> 返回包含 <code>worklow/review/safety</code> 等至少 5 个章节索引的完整规则文档
4	知识库搜索	调用 <code>delphi_kb(action="search", query="TStringList")</code> 返回至少 3 条包含 <code>TStringList</code> 定义的源代码位置记录
5	文件读写与备份	调用 <code>delphi_file(action="write")</code> 写入后, 对应 <code>__history/</code> 目录下生成带版本号后缀的备份文件
6	批量文件编辑	调用 <code>delphi_file(action="batch_write", edits=[...])</code> 同时修改两个不连续行区间, 结果文件内容与预期一致
7	DFM 二进制转换	读取二进制 DFM 文件后自动转换为文本模式, 编辑并回写后文件仍能被 Delphi IDE 正常打开
8	组件管理	调用 <code>manage_component(action="add")</code> 向 DFM 添加组件后, 对应 PAS 文件的 <code>published</code> 段自动生成新组件字段声明
9	代码托管推送	调用 <code>code_hosting(action="git_push")</code> 将本地提交推送到远程仓库, 返回结果包含推送的提交计数
10	异步任务管理	提交 <code>async_task(action="start", task_type="build_knowledge_base")</code> 后返回有效的 <code>task_id</code> , 轮询后得到 <code>completed</code> 状态
11	经验记忆去重	两次保存相似度 > 0.85 的经验记录时, 第二次保存自动合并到第一次记录的更新中而非新增
12	MCP 工具覆盖	<code>list_tools()</code> 返回至少 14 个 Tool 对象, 所有工具在 <code>call_tool()</code> 中均有对应的 handler 处理