AI 驱动的代码编辑器 - Rust 核心开发计划 (优化版)

🎯 项目目标

构建一个平台无关的 Rust 核心库，为 AI 驱动的代码编辑器提供强大的后端支持。该库将被集成到各种平台中（Flutter、Web、原生应用等）。

🏗️ 核心设计原则

1. **平台无关性**：核心逻辑不依赖特定平台
2. **模块化设计**：每个功能都是独立模块
3. **清晰接口**：通过 trait 定义标准接口
4. **可配置性**：支持不同平台的配置需求
5. **渐进式开发**：每个模块可以独立开发和测试
6. **性能优先**：对象池、缓存、异步处理
7. **可扩展性**：支持插件化架构

📁 模块化架构设计 (优化版)

```
rust_core/
├── core/                    # 核心抽象层
│   ├── traits/             # 接口定义
│   ├── types/              # 通用数据类型 (Span, Range, TextDocument,
FileId)
│   ├── errors/             # 错误处理体系
│   └── common/             # 通用工具 (对象池、缓存等)
├── parsers/                # 代码解析模块
│   ├── tree_sitter/        # Tree-sitter 解析器
│   ├── common/             # AST 抽象、Visitor 模式、Query 系统
│   └── incremental/        # 增量解析支持
├── analysis/               # 代码分析模块
│   ├── semantic/           # 语义分析 (SymbolTable, 作用域链)
│   ├── diagnostics/        # 诊断系统 (Severity, Fixable, Suggestion)
│   ├── context/            # 上下文管理 (AI 专用)
│   └── symbols/            # 符号管理 (SymbolKind, 引用查找)
├── ai/                     # AI 交互模块
│   ├── providers/          # 不同 AI 服务提供商
│   ├── prompts/            # 提示词模板引擎 (YAML 定义)
│   ├── responses/          # 响应解析和验证
│   ├── trace/              # 请求/响应追踪
│   └── streaming/          # 流式响应支持
├── lsp/                    # LSP 支持模块
│   ├── client/             # LSP 客户端 (lsp-types 集成)
│   ├── providers/          # 不同语言的 LSP (Pyright subprocess)
│   ├── integration/        # 与核心分析集成
│   └── cache/              # LSP 结果缓存
└── bridge/                 # 平台桥接层
    ├── flutter/            # Flutter 特定接口 (async + frb)
```

```
    ├── wasm/                # WebAssembly 接口
    ├── native/              # 原生库接口
    └── schema/              # JSON 接口 schema 定义
```

🔧 核心 Trait 设计 (优化版)

1. AST 抽象接口

```rust
pub trait Ast {
    type Node;
    type Error;

    fn root_node(&self) -> &Self::Node;
    fn children(&self) -> Vec<&Self::Node>;
    fn node_text(&self, node: &Self::Node) -> &str;
    fn node_kind(&self, node: &Self::Node) -> &str;
    fn node_span(&self, node: &Self::Node) -> Span;
}

pub trait AstVisitor {
    type Ast: Ast;
    type Result;

    fn visit_node(&mut self, node: &<Self::Ast as Ast>::Node) ->
Self::Result;
    fn visit_children(&mut self, node: &<Self::Ast as Ast>::Node) ->
Self::Result;
}
```

2. 代码解析器接口

```rust
pub trait CodeParser {
    type Ast: Ast;
    type Error;

    fn parse(&self, source: &str, language: Language) ->
Result<Self::Ast, Self::Error>;
    fn parse_incremental(&self, source: &str, old_ast: &Self::Ast) ->
Result<Self::Ast, Self::Error>;
    fn get_syntax_errors(&self, ast: &Self::Ast) -> Vec<SyntaxError>;
}

pub trait IncrementalParser: CodeParser {
    fn compute_diff(&self, old_source: &str, new_source: &str) -> Diff;
    fn apply_diff(&self, ast: &Self::Ast, diff: &Diff) ->
Result<Self::Ast, Self::Error>;
}
```

3. 语义分析器接口

```rust
pub trait SemanticAnalyzer {
    type Context;
    type Error;

    fn analyze(&self, ast: &dyn Ast) -> Result<Self::Context,
Self::Error>;
    fn get_symbols(&self, context: &Self::Context) -> Vec<Symbol>;
    fn get_references(&self, context: &Self::Context, symbol: &Symbol) -
> Vec<Reference>;
    fn get_symbol_table(&self, context: &Self::Context) -> &SymbolTable;
}

pub struct SymbolTable {
    pub symbols: HashMap<SymbolId, Symbol>,
    pub scopes: HashMap<ScopeId, Scope>,
    pub scope_chain: Vec<ScopeId>,
}
```

4. AI 服务接口 (优化版)

```rust
pub trait AiProvider {
    type Request;
    type Response;
    type Error;

    async fn generate_code(&self, request: Self::Request) ->
Result<Self::Response, Self::Error>;
    async fn explain_code(&self, code: &str, context: &AiContext) ->
Result<String, Self::Error>;
    async fn suggest_improvements(&self, code: &str, context:
&AiContext) -> Result<Vec<Suggestion>, Self::Error>;
    async fn stream_response(&self, request: Self::Request) ->
Result<Stream<Self::Response>, Self::Error>;
}

pub struct AiContext {
    pub source_code: SourceCode,
    pub symbols: Vec<Symbol>,
    pub diagnostics: Vec<Diagnostic>,
    pub file_context: FileContext,
    pub trace_id: String,
}

pub struct AiResponse {
    pub content: String,
    pub trace_id: String,
```

```
    pub metadata: HashMap<String, Value>,
}
```

5. 诊断系统接口 (优化版)

```
pub trait DiagnosticProvider {
    type Diagnostic;
    type Error;

    fn analyze(&self, ast: &dyn Ast, context: &Context) ->
Result<Vec<Self::Diagnostic>, Self::Error>;
    fn get_quick_fixes(&self, diagnostic: &Self::Diagnostic) ->
Vec<QuickFix>;
}

#[derive(Debug, Clone)]
pub struct Diagnostic {
    pub severity: Severity,
    pub message: String,
    pub span: Span,
    pub code: Option<String>,
    pub fixable: bool,
    pub suggestions: Vec<Suggestion>,
}

#[derive(Debug, Clone)]
pub struct QuickFix {
    pub title: String,
    pub command: FixCommand,
    pub kind: FixKind,
}
```

17 渐进式开发计划 (优化版)

阶段 1: 核心抽象层 (1周)

目标: 建立基础接口和类型系统

任务清单:

- [ ] 定义核心 trait (Ast, CodeParser, SemanticAnalyzer, AiProvider, DiagnosticProvider)
- [ ] 实现通用数据类型 (Span, TextRange, FileId, TextDocument)
- [ ] 建立错误处理体系 (Error types, Result wrappers)
- [ ] 实现对象池和缓存基础设施
- [ ] 编写基础测试框架
- [ ] 创建示例和文档

技术要点：

- **Span 和 Range 设计**：支持多文件语义关联
- **FileId 概念**：为后期 AI 和引用查找做准备
- **对象池缓存**：提升 Tree-sitter 节点性能

交付物：

- 完整的核心抽象层代码
- 单元测试覆盖 (> 90%)
- API 文档
- 性能基准测试

阶段 2: Tree-sitter 解析器 (1-2周)

目标： 实现多语言代码解析

任务清单：

- ☐ Tree-sitter 依赖集成
- ☐ 实现 Ast trait for Tree-sitter
- ☐ 实现 CodeParser trait for Tree-sitter
- ☐ 支持 Python, JSON, YAML, Markdown 语言解析
- ☐ 实现 Visitor 模式和 Query 系统
- ☐ 语法错误检测
- ☐ 增量解析支持
- ☐ 语法高亮测试用例

技术要点：

- **Query 系统**：使用 Tree-sitter query 提取结构，避免手动遍历
- **Visitor 模式**：封装为统一的访问模式
- **增量解析**：实现 IncrementalParser trait

交付物：

- Tree-sitter 解析器实现
- 多语言支持
- AST 操作工具
- 语法错误检测系统
- 性能基准 (< 100ms for 1000 lines)

阶段 3: 基础语义分析 (2周)

目标： 实现代码理解和分析

任务清单：

- ☐ SymbolTable 实现
- ☐ 作用域链管理

- ☐ SymbolKind 定义 (Function, Class, Variable 等)
- ☐ 变量和函数识别
- ☐ 类型推断 (基础)
- ☐ 代码结构提取
- ☐ 依赖关系分析
- ☐ 上下文管理实现
- ☐ 引用查找准备

技术要点：

- **SymbolKind 枚举**：Function, Class, Variable, Module 等
- **scope_id 支持**：为引用查找做准备
- **SymbolTable 设计**：支持作用域链和快速查找

交付物：

- 语义分析器实现
- 符号表系统
- 上下文管理器
- 代码结构分析工具

阶段 4: 诊断系统 (1-2周)

目标： 代码质量分析

任务清单：

- ☐ 实现 DiagnosticProvider trait
- ☐ Severity, Fixable, Suggestion 属性
- ☐ 语法错误检测
- ☐ 代码风格检查
- ☐ 基础建议系统
- ☐ QuickFix 与 Diagnostic 关联
- ☐ FixCommand 模式实现
- ☐ 诊断分级 (Error, Warning, Info, Hint)
- ☐ 自定义规则支持

技术要点：

- **FixCommand 模式**：为 AI 与 UI 提供自动修复建议
- **结构化诊断**：支持 Severity, Fixable, Suggestion 属性

交付物：

- 诊断系统实现
- 代码质量检查工具
- 修复建议系统
- 可配置的规则引擎

阶段 5: AI 集成 (2-3周)

目标： AI 驱动的代码生成和分析

任务清单：

- ☐ AI 服务提供商抽象
- ☐ 提示词模板引擎 (YAML 定义)
- ☐ OpenAI API 集成
- ☐ Claude API 集成
- ☐ 上下文管理优化 (AiContext)
- ☐ 结构化请求类型 (CodeGen, Explain, Fix, CommentSuggest)
- ☐ 代码生成功能
- ☐ 代码修改功能
- ☐ 代码解释功能
- ☐ 流式响应支持
- ☐ 请求/响应追踪 (trace_id)
- ☐ 错误处理和重试机制

技术要点：

- **提示词模板引擎**：支持 YAML 定义 prompt
- **trace 机制**：记录请求/响应用于调试和 UI 展示
- **结构化请求**：CodeGen, Explain, Fix, CommentSuggest 多种类型

交付物：

- AI 服务集成
- 智能代码生成
- 代码修改工具
- 上下文管理系统
- 流式响应支持

阶段 6: LSP 支持 (2周)

目标： 增强语义分析能力

任务清单：

- ☐ LSP 客户端实现 (lsp-types 集成)
- ☐ 标准化协议 (initialize, textDocument/didOpen 等)
- ☐ Pyright 集成 (subprocess + stdio)
- ☐ JSON Schema 验证
- ☐ 智能补全数据
- ☐ 高级诊断信息
- ☐ 与现有分析集成
- ☐ LSP 结果缓存 (带文档 hash key)
- ☐ 性能优化

- [ ] 错误处理和重连

技术要点：

- **lsp-types crate**：标准化 LSP 协议
- **Pyright subprocess**：本地部署，避免 WebSocket 复杂性
- **LSP 缓存**：带文档 hash key 的结果缓存

交付物：

- LSP 客户端
- 语言特定支持
- 增强的语义分析
- 智能补全系统

阶段 7: 平台桥接 (1-2周)

目标： 支持不同平台集成

任务清单：

- [ ] Flutter 桥接 (flutter_rust_bridge, async + frb)
- [ ] WebAssembly 支持
- [ ] 原生库接口
- [ ] JSON 接口 schema 定义 (diagnostic.json, tokens.json)
- [ ] 平台特定优化
- [ ] 集成测试
- [ ] 部署文档

技术要点：

- **async 接口设计**：AI 与 LSP 调用需要异步
- **frb_mirror 注解**：结构体序列化支持
- **JSON schema**：前后端协作清晰

交付物：

- 多平台支持
- 集成指南
- 示例项目
- 部署文档

🔄 开发策略 (优化版)

1. 测试驱动开发 (TDD)

- 先写测试，再实现功能
- 每个模块都有完整的测试覆盖
- 持续集成和自动化测试

- 性能基准测试

2. 模块独立开发

- 每个模块可以独立开发和测试
- 清晰的模块边界和接口
- 最小化模块间依赖
- 插件化架构支持

3. 接口优先设计

- 先定义接口，再实现具体功能
- 接口设计考虑扩展性
- 保持向后兼容
- 支持多态和泛型

4. 性能优化策略

- **对象池缓存**：Tree-sitter 节点缓存
- **LSP 结果缓存**：带文档 hash key
- **异步处理**：AI 调用、LSP 通信异步化
- **内存管理**：及时释放不需要的上下文

5. 错误处理策略

- **分级错误**：致命错误、可恢复错误、警告
- **降级机制**：LSP 失败时回退到 Tree-sitter
- **用户反馈**：清晰的错误信息和解决建议
- **重试机制**：网络请求自动重试

📊 质量保证 (优化版)

代码质量

- **测试覆盖率**：> 90%
- **代码审查**：所有代码必须经过审查
- **静态分析**：使用 clippy 等工具
- **文档完整性**：所有公共 API 都有文档
- **性能基准**：每个模块都有性能测试

性能要求

- **解析速度**：< 100ms for 1000 lines
- **内存使用**：合理的内存占用，对象池优化
- **响应时间**：AI 调用 < 5s，LSP 调用 < 2s
- **并发支持**：支持多线程操作
- **缓存效率**：LSP 结果缓存命中率 > 80%

兼容性

- **Rust 版本**：支持最新的稳定版本
- **平台支持**：Windows, macOS, Linux
- **架构支持**：x86_64, ARM64
- **语言支持**：Python, JSON, YAML, Markdown

📝 文档要求 (优化版)

每个阶段都需要：

1. **API 文档**：完整的接口文档，包含示例
2. **使用示例**：实际的使用案例和最佳实践
3. **集成指南**：如何集成到不同平台
4. **性能指南**：性能优化建议和基准
5. **故障排除**：常见问题和解决方案
6. **架构文档**：模块间关系和设计决策

🎯 成功标准 (优化版)

项目成功的标准：

1. **功能完整**：所有计划功能都实现
2. **性能达标**：满足性能要求，通过基准测试
3. **质量保证**：通过所有测试，代码质量高
4. **文档完整**：文档清晰易懂，示例丰富
5. **易于集成**：可以轻松集成到不同平台
6. **用户满意**：满足最终用户需求
7. **可扩展性**：支持插件化扩展
8. **可维护性**：代码结构清晰，易于维护

🧠 技术亮点

AI 接口设计

- **统一结构**：AiContext + SourceCode + RequestType → AiResponse
- **追踪机制**：所有响应支持 trace_id，便于链式追踪
- **流式响应**：支持 chunked 响应，为前端提示式体验准备

性能保障

- **对象池缓存**：Tree-sitter 节点使用对象池缓存
- **LSP 结果缓存**：带文档 hash key 的智能缓存
- **异步处理**：所有 I/O 操作异步化
- **内存优化**：及时释放不需要的上下文

扩展性设计

- **插件化架构**：支持新的语言和 AI 服务
- **配置驱动**：用户可配置的规则和行为
- **标准化接口**：清晰的模块接口和协议

最后更新：2024年