# 不同错误信息的结合下LLM的软件修复能力探究

信息科学技术学院 22级
元铭宇、杨世航、蒋康悦
代码仓库：https://github.com/mkbk-with-circle/software-testing

## 一、研究背景与动机

### 研究背景

随着大语言模型（LLM）在自然语言处理和计算机程序生成领域的突破，它们在代码生成与自动修复任务中展现出巨大潜力。自动化错误修复系统能够极大降低开发者在调试过程中的时间成本和精力投入，从而提高软件开发和维护的效率。尤其在复杂系统中，传统的人工调试和修复方法面临着越来越大的挑战，迫切需要能够自动识别和修复代码缺陷的工具。

然而，现有的研究和应用大多依赖**单一的错误信息源**，例如直接使用错误源码或者某一类错误提示进行修复。这些方法虽然在一些简单场景下表现不错，但并未充分考虑到实际开发中错误信息的多样性和复杂性。在实际的软件开发过程中，开发者往往需要综合考虑多个错误信息来源，如错误提示、堆栈追踪、失败的测试用例、代码上下文等，这些信息共同构成了调试和修复过程中不可或缺的一部分。现有方法尚未能够有效利用这些多样化的信息源，因此在多变和复杂的错误情境下，修复效果往往不尽如人意

### 研究动机

本研究的核心动机是提出一种**高效的 Prompt 输入框架**，旨在提升大语言模型（LLM）在跨项目调试任务中的修复效率与准确性。通过精简和合理组织多种来源的错误信息，我们期望在**控制 Token 数量**的同时，尽可能保留关键信息的完整性。具体而言，该框架的设计目标包括：

信息精简与组织：通过结构化的方式将多源错误信息（如报错信息、堆栈追踪、测试失败等）整合进统一的输入框架，以确保在尽可能少的 Token 数量下，仍能有效传递所有关键修复线索。

跨项目适应性：不仅限于特定的项目或代码库，框架设计需具备良好的扩展性和通用性，能够适用于不同类型的代码和错误模式，提高 LLM 在各类软件项目中的调试和修复能力。

提高修复效率与准确性：通过精心设计的输入框架，使 LLM 能够在更短的时间内高效定位错误、生成修复代码，从而提升整体的修复准确率，减少开发者的干预和后续维护成本。

# 二、核心问题

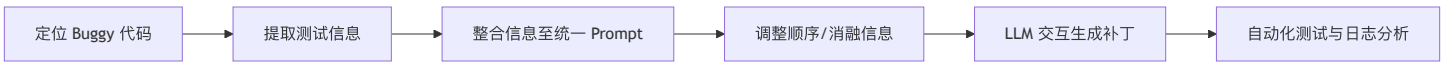
1. **信息输入顺序的影响**
   在信息充足时，何种输入顺序能使 LLM 达到最优修复效果?
2. **关键信息的决定性作用**
   哪些错误信息对修复成功率影响更大？是否存在决定性信息?
3. **框架扩展性验证**
   该提示框架能否适配不同数据集?

# 三、研究过程与实现

## 数据集准备

| 数据集 | 语言 | 样本量 | 特点 |
| --- | --- | --- | --- |
| Defects4J | Java | 63 | 工业级项目，真实缺陷-修复对 |
| Bears Benchmark | Java | 24 | 基于 Travis CI 构建记录收集 |
| QuixBugs | Python | 31 | 来自挑战赛，多测试框架支持 |

## 技术流程




## 关键步骤详解

1. **信息提取**

> 在信息提取阶段，主要通过以下几个步骤收集与错误相关的数据：

- **定位代码变更位置（Defects4J）**
  通过 `git diff` 命令，我们能够精确地定位到源代码中被修改的部分，具体来说，使用

`git diff --name-only` 比较两个提交之间的差异，确定哪些文件发生了变化，进一步通过比较具体的文件内容找出修改的代码行。这一过程是定位缺陷代码的基础。

- **解析错误信息（Bears）**
  在 Bears Benchmark 数据集中，每个错误记录都包含了 `bears.json` 文件，其中详细列出了与错误相关的元数据，例如触发错误的测试用例、错误消息、异常类型等。通过解析 `bears.json` 文件，我们可以提取出错误的具体信息，如失败的测试类、错误的输入输出数据、抛出的异常等。这些信息是构建 Prompt 输入框架的关键。
- **解析测试日志（QuixBugs）**
  在 QuixBugs 数据集中，错误信息通常会通过测试日志记录下来。我们从这些日志中提取出失败的测试用例信息和具体的断言错误。这些测试信息有助于定位修复过程中的关键问题，能够为 LLM 提供更丰富的上下文。

2. **Prompt 构建**

> 在信息提取后，下一步是将提取到的错误信息按照一定的格式整合成一个标准化的 Prompt 输入框架：

- **统一 Markdown 格式**
  所有的错误信息都被整理成统一的 Markdown 格式，包含以下几个主要部分：
  - **Buggy Code**：出现问题的源代码段，用于提供错误的上下文。
  - **Failed Test**：触发错误的测试用例信息，帮助 LLM 定位到具体的测试失败情况。
  - **Test Line**：导致测试失败的断言或语句行，标明出错的具体位置。
  - **Error Message**：运行时抛出的错误信息或异常类型，指出错误的性质。
  - **Error Code Block**：出错的代码块或与错误相关的代码段，帮助 LLM 更好地理解问题。
- **顺序调整与要素消融**
  利用 `custom_sort.py` 脚本进行顺序调整和要素消融实验。我们通过改变信息在 Prompt 中的顺序，测试不同顺序对 LLM 修复效果的影响。此外，消融实验将逐步去除某些要素（如错误代码块或测试行），观察这些信息的缺失如何影响修复成功率。

3. **LLM 交互与测试**

> 在 Prompt 构建完毕后，接下来是与大语言模型（LLM）的交互和测试验证过程：

- **通过 DeepSeek API 提交 Prompt**
  我们通过 DeepSeek API 将构建好的 Prompt 提交给 LLM。DeepSeek 是一个自动化的 LLM 交互平台，能够通过 API 接口与 LLM 进行对话，生成修复建议。通过这种方式，LLM 会读取 Prompt 中的信息，分析错误并生成相应的修复代码。我们会通过在prompt信息前后加一些提示词，来引导LLM生成更符合我们需求的代码。
- **自动化脚本替换源码并执行测试**
  一旦 LLM 返回了修复建议，自动化脚本会将生成的修复代码替换到原始代码中。替换完成后，脚

本会自动运行测试框架（如 Defects4J 中的测试用例、Bears Benchmark 中的测试集等）验证修复结果。通过对比修复前后的测试结果，判断 LLM 是否成功修复了错误。

通过这一整套流程，我们能够评估不同错误信息顺序对 LLM 修复效果的影响，并探索如何在控制信息量的同时，最大化提升 LLM 在代码修复中的性能。

# 四、结果分析与结论

## 关键发现

1. **信息输入顺序显著影响修复成功率**
   最优顺序：Buggy Code → Failed Test → Test Line → Error → Error Code Block

原因：
Transformer 存在 "位置偏置"：序列开头/末尾信息权重更高（首因效应+近因效应）
关键信息前置（如测试行）避免 "Lost-in-the-Middle" 问题

2. **关键信息的边际价值分析**
   消融信息 Defects4J成功率变化 QuixBugs成功率变化
   移除 Error Code Block ↓16.7% 成功率略有降低
   移除 Failed Test+Test Line 影响较小 成功率显著上升
   核心结论：
   Error Code Block 和 Error Message 为 决定性信息
   Failed Test + Test Line 存在冗余性
3. **数据集特性对提示框架的影响**
   轻量级代码（QuixBugs）：
   信息冗余度高（Buggy Code ≈ Error Code Block）
   顺序影响较弱

复杂系统（Defects4J）：
提示设计显著影响修复效果

# 五、反思与未来工作

## 局限性

数据集规模有限（总计 118 个样本），需扩大验证范围
仅使用 DeepSeek 单一模型，未对比 ChatGPT/Gemini 等主流 LLM

评估维度单一（仅测试通过率），未考察代码质量/可维护性

## 优化方向

1. **跨模型验证：**
   探究提示框架在 ChatGPT、Claude 等模型上的泛化能力
2. **动态迭代机制：**
   引入多轮交互反馈优化修复结果
3. **信息压缩技术：**
   设计 Token 高效的错误信息表示方法
4. **多维评估体系：**
   加入代码复杂度、风格一致性等指标

# 六、个人分工

## 信息收集阶段

分头读论文，具体论文笔记在仓库内

## 数据集选择和实验

defects4j 元铭宇
Bears Benchmark 杨世航
QuixBugs 蒋康悦

## 答辩PPT和实验报告

协作完成