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

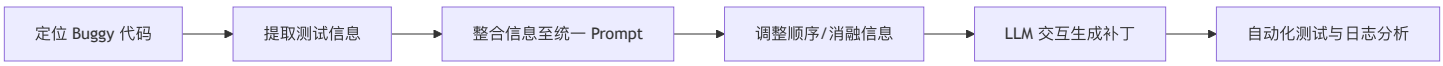
1. **信息输入顺序的影响**
   在信息充足时，何种输入顺序能使 LLM 达到最优修复效果?
2. **关键信息的决定性作用**
   哪些错误信息对修复成功率影响更大？是否存在决定性信息?
3. **框架扩展性验证**
   该提示框架能否适配不同数据集?

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。
  - 顺序：
    a. **初始**（`Buggy code → Failed test → Test line → Error → Error Code Block`）
    b. **人类启发**（`Failed test → Error → Test line → Error Code Block → Buggy code`）
    c. **自底向上**（`Buggy code → Error Code Block → Test line → Failed test → Error`）
    d. **Test-first 策略**
    （`Error Code Block → Test line → Failed test → Error → Buggy code`）
    e. **Buggy-code 优先**
    （`Buggy code → Test line → Failed test → Error Code Block → Error`）
    f. **先报错后源码**
    （`Error → Test line → Failed test → Error Code Block → Buggy code`）
    g. **LLM 默认策略**
    （`Buggy code → Failed test → Error → Error Code Block → Test line`）

- 消融组件
  a. **Error Code Block**
  b. **Failed test + Test line**
  c. **Failed test + Test line + Error Code Block**
  d. **Failed test + Test line + Error**

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

Defects4J数据集实验结果

调换顺序的实验结果

方法	通过数量	通过率
初始	25/60	41.7%
人类启发	20/60	33.3%
自底向上	19/60	31.7%
Test-first	22/60	36.7%
Buggy-code 优先	18/60	30.0%
先报错后源码	20/60	33.3%

方法	通过数量	通过率
LLM 默认策略	16/60	26.7%

从统计结果来看，信息顺序能够显著影响调试结果：不同策略的通过率差异达 15%（最佳策略 41.7% vs 最差 26.7%），证明信息组织方式对大模型定位和修复错误的能力有实质性影响

从正确率中可以看到“初始顺序”是最佳的顺序，显著优于其他策略，我们推测原因可能在于它首先呈现了待修复的**有缺陷源码**，让模型一开始就聚焦于代码细节，然后依次提供失败的测试及错误信息，形成由代码到错误的顺序。这种顺序类似于**先给出问题对象（代码），再给出症状（测试失败和错误）**，模型能够直接将错误症状与代码实现联系起来进行推理修复。

“LLM 默认策略”的顺序（源码→测试→错误→错误代码片段→测试行）成功率最低，该策略将**具体失败的测试行**信息放在最后提供，这可能导致模型在阅读前面内容时无法准确定位错误发生的上下文。等模型看到最后的测试行时，前面的信息可能已经在**模型注意力中衰减**，影响了综合理解。这与LLM的**注意力分散机制**有关

总的来说，高成功率策略体现出两个关键点：**及时提供症状信息**和**合理的上下文顺序**。要么在开头就呈现代码并紧随其后给出测试/错误线索（如“最开始”策略），要么在开头就呈现故障现象让模型理解问题再提供代码细节（如“Test-first”策略）。反之，低效策略往往把重要的测试线索或错误信息拖到最后，导致模型无法充分利用这些信息。

消融实验结果

消融内容	成功数	成功率	相对降幅
只去掉 **Error Code Block**	15/60	25.0 %	↓ 16.7 pp
去掉 **Failed test + Test line**	17/60	28.3 %	↓ 13.4 pp
去掉 **Failed test + Test line + Error Code Block**	14/60	23.3 %	↓ 18.4 pp
去掉 **Failed test + Test line + Error**	10/60	16.7 %	↓ 25.0 pp

消融实验以初始策略（41.7%通过率）为基线，通过逐步移除关键信息组件，量化了各要素对调试效果的影响：

1. Error Code Block 的核心作用：仅移除 Error Code Block 时，通过率骤降 16.7 个百分点（41.7% → 25.0%），降幅最大。
    - 结论：错误代码块是模型定位问题的核心锚点，直接暴露缺陷位置可显著降低模型认知负荷。
2. 测试上下文（Failed test + Test line）与Error Code Block在一定程度上互斥。

- 结论：存在一定的信息冗余性，两者都提供了错误定位的关键信息，但侧重点不同。Error Code Block直接指向问题代码，而测试上下文则提供了错误发生的场景和触发条件。当信息不足或者需要节约tokens数时，可以选择保留其一

Bears Benchmark数据集实验结果

方法	通过数量	通过率
初始	8	33.3%
人类启发	5	20.8%
自底而上	5	20.8%
Test-first	6	25%
Code优先	6	25%
报错优先	6	25%
LLM默认	5	20.8%

与Defects4J相比，Bears Benchmark数据集的难度较高：一方面数据集的信息提取操作较为复杂，另一方面自动构建记录收集的bug种类繁杂，修复较困难；这样导致了整体的通过率比较低，但第一项顺序的突出表现仍然可以辅助Defects4J证明“初始”prompt具有不错的优势。

由于full information的通过数量都不够高，考虑到在消融实验后可能会进一步扩大某些特定样本的影响导致实验偏差，于是没有对Bears Benchmark做进一步的消融实验。

可以根据前两个实验的结果来分析“初始”策略的优势在于——信息上下文衔接紧密。先给出“Buggy code”，为模型提供完整的代码上下文；随后展示“Failed test”与“Test line”，迅速引导模型关注失败的场景和位置；最后补充“Error”与“Error Code Block”，让模型在已有代码与测试信息的基础上，再去理解错误细节；这样先整体后局部、由广及深的信息呈现显著地减少了模型理解的“跳跃”成本。

QuixBugs数据集实验结果

调换顺序的实验结果

方法	通过数量	通过率
初始	27/31	87.10%
人类启发	26/31	83.87%
自底向上	26/31	83.87%

方法	通过数量	通过率
Test-first	28/31	90.32%
Buggy-code 优先	29/31	93.55%
先报错后源码	27/31	87.10%
LLM 默认策略	27/31	87.10%

从统计结果来看，整体通过率比较高，信息顺序对于实验结果的影响不是特别大，可能是因为代码相对简单的缘故，但是依然有一些策略表现出较高的潜力。初始策略在本数据集中表现中规中矩，表现不是特别有优势，但也没有表现得很糟糕。

初始策略的平庸可能和“U-形利用率”的特点有一定联系。因为本数据集代码较为短小，第一部分buggy code和最后一部分error code block之间重复率大，而大模型对这两个部分注意力较为集中，可能导致信息冗余从而使得修复效率下降。

由于整体正确率较高且每组之间正确率差异并不十分显著，考虑进行消融实验进一步分析各类信息对于修复的作用。

消融实验结果

消融内容	成功数	成功率
只去掉 **Error Code Block**	24/31	77.42 %
去掉 **Failed test + Test line**	29/31	93.55 %
去掉 **Failed test + Test line + Error Code Block**	25/31	80.65 %
去掉 **Failed test + Test line + Error**	26/31	83.87 %

消融实验以初始策略（27/31）为基线，通过逐步移除关键信息组件，量化了各要素对调试效果的影响：

1. Error Code Block 的核心作用：仅移除 Error Code Block 时，通过率骤降 9.48 个百分点（87.10% → 77.42%），降幅最大。
   - 结论：虽然代码短小，但是错误代码块信息依然重要，可以显著降低模型的负担
2. 去除测试上下文（Failed test + Test line）反而导致正确率上升。
   - 结论：存在一定的信息冗余性，由于本数据集中单个代码测试样例较多，测试上下文内容较多且不同测试之间相似性比较高，产生了一定的冗余。

跨数据集的实验分析与结论

1. **信息输入顺序显著影响修复成功率**
   通过三个数据集的实验验证，我们发现信息输入顺序对LLM的修复效果具有显著影响。实验结果表明，最优的信息输入顺序为：Buggy Code → Failed Test → Test Line → Error → Error Code Block。这种顺序的优势主要体现在以下几个方面：
   - Transformer架构的"位置偏置"特性使得序列开头和末尾的信息权重更高，体现了**首因效应**和**近因效应**
   - 将关键信息（如测试行）前置可以有效避免"Lost-in-the-Middle"问题
   - 信息上下文衔接紧密，形成了由代码到错误的**适合LLM的自然推理路径**，有助于模型更好地理解问题
2. **数据集特性对提示框架的影响**
   通过对比三个不同特性的数据集，我们发现提示框架的效果与数据集特性密切相关：
   - 轻量级代码（QuixBugs）：
     - 代码结构简单，信息冗余度较高，这种情况下可以考虑使用更少的tokens来输入信息
     - 信息输入顺序对修复效果的影响相对较弱，在某些情况下可能会出现消去一些信息后获得更好效果的情况。这与LLM的注意力分散机制有关。
   - 中度复杂系统（Defects4J）：
     - 提示设计对修复效果的影响最为显著
     - 不同信息顺序的通过率差异可达15个百分点（最佳41.7% vs 最差26.7%）
     - 需要更完整的信息组合才能达到理想的修复效果
   - 复杂系统（Bears Benchmark）：
     - 整体通过率较低（最高33.3%），反映了其较高的修复难度
     - 信息提取过程复杂，bug种类繁多
     - 实验结果验证了"初始"prompt策略的普适性优势

工程启示：

1. 必要组件：错误代码块（Error Code Block）必须保留，缺失时将损失较大。
2. 优化建议：在输入长度受限时，可以去除Failed test 和Test line信息，该信息冗余度比较大

五、反思与未来工作

局限性

1. 虽然从实验结果上看，在不同的数据集上较优的信息顺序基本上相同，但由于数据集总量还是太少，仍然需要扩大数据量来进一步验证结论的可靠性。

2. 由于目前仅使用了 DeepSeek 作为修复的 LLM，且未能调用 ChatGPT 等主流模型的 API，后续应选择多个广泛使用的 LLM，探究同一套信息组织规则是否适用于大部分模型，因为不同模型的预训练和训练方式存在差异。
3. 当前评估仅考察了代码是否通过测试，未对代码风格、代码复杂度、可维护性等方面进行更全面深入的评价，后续需建立更完善的多维评估体系。
4. 尚未进行迭代代码测试，未来可以尝试多轮交互或迭代修复，或许能获得更好的修复效果，也可能覆盖或弥补部分信息缺陷。

优化方向

1. **数据集扩充：**
   持续收集和整理更多样本，覆盖更多项目和错误类型，提升实验结论的统计显著性和泛化能力。
2. **跨模型验证：**
   探究提示框架在 ChatGPT、Claude、Gemini 等主流 LLM 上的适用性和表现，分析不同模型对信息组织方式的敏感性。
3. **动态迭代机制：**
   引入多轮交互反馈，优化修复结果，尝试自动化的迭代修复流程，提升修复的成功率和代码质量。
4. **多维评估体系：**
   除了测试通过率，还应引入代码复杂度、风格一致性、可维护性等多维度指标，全面评价修复效果。

六、分工

元铭宇：进行前期调研以及确定实验方案、路径，及分配任务；在Defects4j数据集上进行实验
杨世航：前期进行论文阅读；在Bears Benchmark数据集上进行实验
蒋康悦：前期进行论文阅读；在QuixBugs数据集上进行实验
实验报告及PPT答辩由三人协作完成

七、参考文献

1. **Integrating Various Software Artifacts for Better LLM-based Bug Localization and Program Repair** ,Feng et al., TOSEM 2024
2. **Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT**, Chunqiu Steven Xia, Lingming Zhang
3. **SWE-bench: Can Language Models Resolve Real-World GitHub Issues?**, Jimenez et al., ICLR 2024

4. **Demystifying faulty code: Step-by-step reasoning for explainable fault localization**, Ratnadira WIDYASARI et al.
5. **AutoCodeRover: Autonomous Program Improvement**,Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, Abhik Roychoudhury
6. **A Deep Dive into Large Language Models for Automated Bug Localization and Repair**,SONEYA BINTA HOSSAIN et al.
7. **Evaluating Fault Localization and Program Repair Capabilities of Existing Closed-Source General-Purpose LLMs**, Shengbei Jiang Beijing Jiaotong University, Beijing, China; Jiabao Zhang; Wei Chen; Bo Wang; Jianyi Zhou; Jie M. Zhang
8. **Less is More: Adaptive Program Repair with Bug Localization and Preference Learning**, Zhenlong Dai et al.
9. **The Impact of Input Order Bias on Large Language Models for Software Fault Localization**,Md Nakhla Rafi, Dong Jae Kim, Tse-Hsun Chen, Shaowei Wang
10. **The Order Effect: Investigating Prompt Sensitivity in Closed-Source LLMs**,Bryan Guan et al.
11. **Order Matters: Exploring Order Sensitivity in Multimodal Large Language Models**,Zhijie Tan et al.