

# 软件测试导论

## 大作业答辩

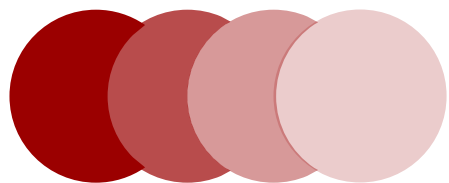
不同错误信息的结合下LLM的软件修复能力探究

信息科学技术学院 22级 元铭宇  
信息科学技术学院 22级 杨世航  
信息科学技术学院 22级 蒋康悦

# 目录

## Content

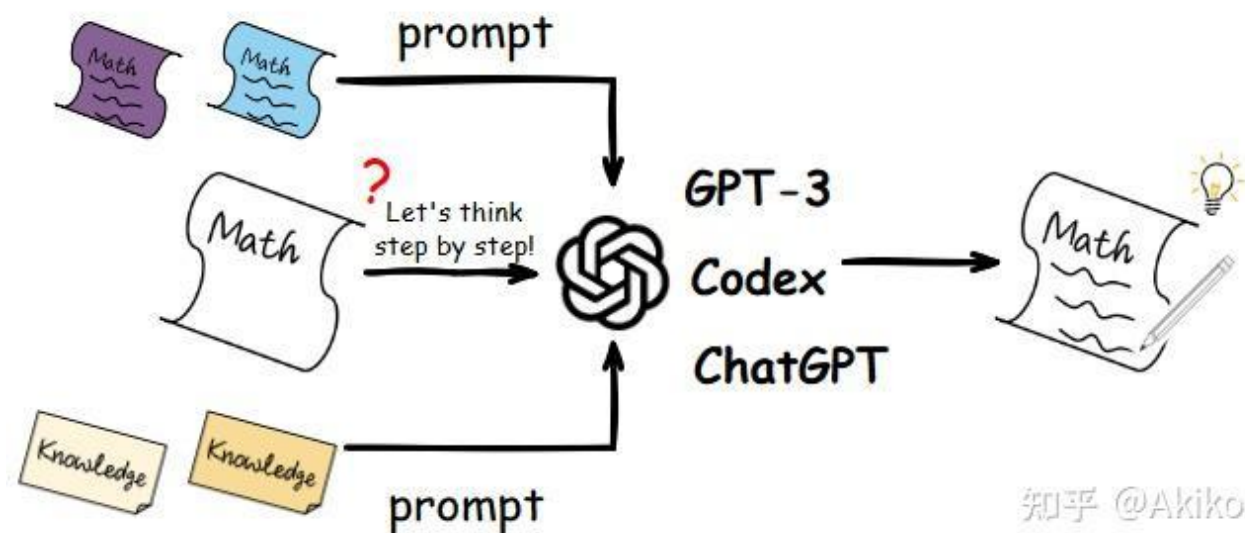
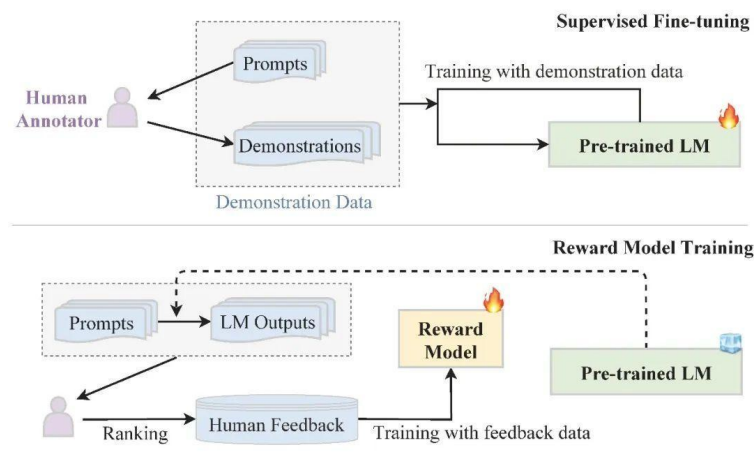
- 01 ● 研究背景与动机
- 02 ● 核心问题
- 03 ● 具体的研究过程与实现
- 04 ● 结果分析及结论
- 05 ● 反思和未来可能继续优化的工作



# 01 【研究背景与动机】

---

# 一，研究背景与动机



知乎 @Akiko

## 【研究背景】

- 随着大语言模型（LLM）能力的提升，它们在代码生成与**自动修复任务**中表现出巨大潜力
- 软件开发实践中，自动化错误修复能显著降低调试成本，提高维护效率。
- 当前研究多集中于直接利用错误源码或错误信息进行修复，但实际开发场景中，开发者常常同时依赖多种错误相关信息（如错误提示、失败测试、错误堆栈、代码上下文等）。

# 一，研究背景与动机

## 【研究背景——当前的研究工作】

### 一、Bug 定位 + 修复生成（最基础范式）

逻辑：LLM 读取缺陷代码 → 定位问题 → 生成修复代码

代表方法：InferFix（先检索示例，再由 LLM 生成修复）

适用场景：无需外部交互或工具，单轮提示式修复任务

### 二、多代理协同修复（模块分工 + 协作）

逻辑：多个 LLM Agent 各司其职 → 定位 + 修复 + 验证 → 共享上下文

代表方法：FixAgent（定位、修复、变量分析等由独立代理完成）

优势：可模块化扩展，适合构建端到端自动调试系统

### 三、对话式半自动修复（人类交互指导）

逻辑：人类通过交互引导 LLM → 明确任务意图 → 优化修复输出

代表方法：CREF（Conversational Repair Framework）

亮点：兼顾人类经验与 LLM 生成能力，提高可靠性和可控性

### 四、自我调试与反馈机制（Self-Debugging）

逻辑：模型先生成代码 → 主动进行推理检查 → 发现问题后自动修正

代表方法：Self-Debug (OpenReview)

核心机制：让 LLM 扮演自己的“橡皮鸭”，进行反思与自我修复

### 五、基于运行时信息的调试（动态执行辅助）

逻辑：将程序划分为基本块 → 利用变量值、路径等执行信息逐块验证

代表方法：LDB (Debug Like a Human)

亮点：结合运行时上下文，强化调试与修复的因果推理能力



但是现实中谁管那么多，  
我们肯定直接输入啊！

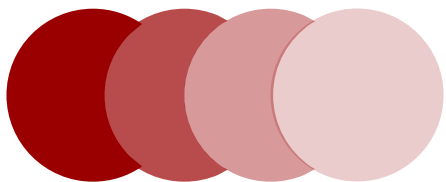
# 一，研究背景与动机

## 【研究动机】

为提高大语言模型（LLM）在不同软件项目中的修复效率、准确性，我们希望提出构建一种高效的提示（prompt）输入框架，通过尽可能精简和组织错误相关信息（如报错语句、堆栈追踪等），在控制 token 数量的同时保持信息完整性，实现高效地LLM辅助debug

	Claude 3.5 Sonnet (new)	Claude 3.5 Haiku	Claude 3.5 Sonnet	GPT-4o*	GPT-4o mini*	Gemini 1.5 Pro	Gemini 1.5 Flash
Graduate level reasoning <i>GPQA (Diamond)</i>	65.0% 0-shot CoT	41.6% 0-shot CoT	59.4% 0-shot CoT	53.6% 0-shot CoT	40.2% 0-shot CoT	59.1% 0-shot CoT	51.0% 0-shot CoT
Undergraduate level knowledge <i>MMLU Pro</i>	78.0% 0-shot CoT	65.0% 0-shot CoT	75.1% 0-shot CoT	—	—	75.8% 0-shot CoT	67.3% 0-shot CoT
Code <i>HumanEval</i>	93.7% 0-shot	88.1% 0-shot	92.0% 0-shot	90.2% 0-shot	87.2% 0-shot	—	—
Math problem-solving <i>MATH</i>	78.3% 0-shot CoT	69.2% 0-shot CoT	71.1% 0-shot CoT	76.6% 0-shot CoT	70.2% 0-shot CoT	86.5% 4-shot CoT	77.9% 4-shot CoT
High school math competition <i>AIME 2024</i>	16.0% 0-shot CoT	5.3% 0-shot CoT	9.6% 0-shot CoT	9.3% 0-shot CoT	—	—	—
Visual Q/A <i>MMMU</i>	70.4% 0-shot CoT	—	68.3% 0-shot CoT	69.1% 0-shot CoT	59.4% 0-shot CoT	65.9% 0-shot CoT	62.3% 0-shot CoT
Agentic coding <i>SWE-bench Verified</i>	49.0%	40.6%	33.4%	—	—	—	—
Agentic tool use <i>TAU-bench</i>	Retail 69.2%	Retail 51.0%	Retail 62.6%	—	—	—	—
	Airline 46.0%	Airline 22.8%	Airline 36.0%				

\* Our evaluation tables exclude OpenAI's o1 model family as they depend on extensive pre-response computation time, unlike typical models. This fundamental difference makes performance comparisons difficult.



## 02 【核心问题】

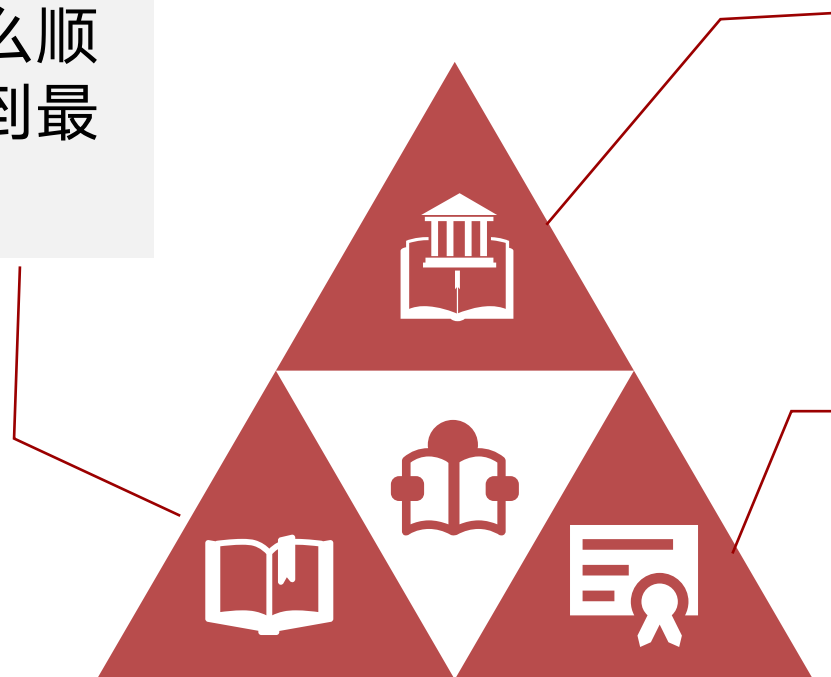
---

## 二，核心问题

---

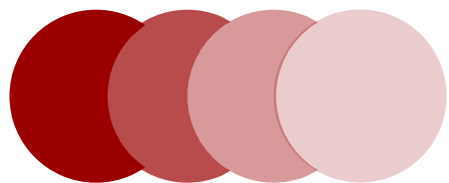
1. 在信息充足，不需要考虑tokens数的情况下，以什么顺序将信息输入给LLM会得到最好的debug效果

2. 哪些信息对LLM的软件修复成功率的影响更大，或者说是否有决定性的debug信息



3. 这套提示（prompt）输入框架是否具有良好的扩展性，是否可以运用到不同的数据集上（非主要的）





## 03 【具体的研究过程与实现】

---

### 三，具体的研究过程与实现

---

gitbug-java?  
GHPK? BugsInPy

## 前期工作——数据集准备

**Defects4J:** Defects4J 是一个被广泛使用的、专为软件缺陷研究设计的 真实 Java 项目缺陷数据集。该数据集包含多个 工业级 开源 Java 项目，每个项目中都提供了大量经过人工确认的 真实缺陷（buggy）与修复版本（fixed）对，并配有可复现的回归测试用例。在本项目中，我们选择了里面的Lang项目共63份代码，涵盖了字符串处理、数字处理、对象工具类等。

**Bears Benchmark:** 简称Bears，是Java 8中自动程序修复研究的bug基准。通过分析 GitHub 上开源项目在 Travis CI 持续集成平台上的构建记录，来自动收集程序错误及其修复补丁，选择了其中的24份项目代码。项目中的每个commit包括：1、错误版本的源代码；2、引发测试失败的测试用例；3、修复后的源代码；4、包含错误元数据的 bears.json 文件

**QuixBugs** 基准测试包含 40 个来自 Quixey Challenge 的程序，这些程序均已翻译成 Python 和 Java 语言。每个程序都包含一行代码的缺陷，以及通过（如果可能）和失败的测试用例，缺陷分为 14 个缺陷类别，并且拥有多种测试框架。此外，程序的正确版本也可以通过--correct调用。在实验中我们选取了本数据集中31份Python代码和pytest测试框架。

## 三，具体的研究过程与实现

---

### 中期工作——数据集处理

步骤一：定位buggy代码在源代码中的位置

步骤二：提取相关测试信息，如出错的test代码，具体的assert error，错误与正确的输入输出等等

步骤三：通过自动化脚本整合信息到统一格式的prompt

步骤四：按照事先调研确定好的顺序调整/消去部分信息后，通过自动化对话脚本喂给LLM

步骤五：获得LLM反馈后，整合到buggy代码后进行test

步骤六：从log日志中收集信息并且整合分析

### 三，具体的研究过程与实现--Defects4J数据集

## 第一板块：generate\_prompt

```
root@ca3270bdf51:/ymy/test# defects4j export -p tests.trigger -w .
Running ant (export.tests.trigger).....
..... OK
org.apache.commons.lang3.time.FastDateFormat_ParserTest::testLANG_831
org.apache.commons.lang3.time.FastDateParserTest::testLANG_831
root@ca3270bdf51:/ymy/test#
```

### 3. 再运行defects4j export -p tests.trigger -w . 命令 获取test相关信息

#### 4. 调用开源项目ChatRepair中的函数提取未通过测试的method

## 5. 将收集到的信息合并到类型为markdown的prompt文件中

6. 对于打乱顺序的和消融实验，会额外进行一次处理

```

Excluding broken/flaky tests..... OK
Excluding broken/flaky tests..... OK
Backing up build file: /ymy/test/default.properties..... OK
Initialize fixed program version..... OK
Apply patch..... OK
Initialize buggy program version..... OK
Diff d1a45e97:396afc3e..... OK
Apply patch..... OK
Tag pre-fix revision..... OK
Check out program version: Lang-1b..... OK

```

## 三，具体的研究过程与实现

### Buggy code

```
public int translate(final CharSequence input, final int index, final Writer out) throw
    int max = longest;
    if (index + longest > input.length()) {
        max = input.length() - index;
    }
    // descend so as to get a greedy algorithm
    for (int i = max; i >= shortest; i--) {
        final CharSequence subSeq = input.subSequence(index, index + i);
        final CharSequence result = lookupMap.get(subSeq);
        if (result != null) {
            out.write(result.toString());
            return i;
        }
    }
    return 0;
}
```

### Failed test

org.apache.commons.lang3.text.translate.LookupTranslatorTest::testLang882

### Test line

```
assertEquals("Incorrect codepoint consumption", 3, result);
```

### Error

junit.framework.AssertionFailedError: Incorrect codepoint consumption expected:<3> but was:<0>

### 第一板块：generate\_prompt

### Error Code Block

```
public void testLang882() throws IOException {
    final LookupTranslator lt = new LookupTranslator(new CharSequence[][] { { new S
    final StringWriter out = new StringWriter();
    final int result = lt.translate(new StringBuffer("one"), 0, out);
    assertEquals("Incorrect codepoint consumption", 3, result);
    assertEquals("Incorrect value", "two", out.toString());
}
```

## 三，具体的研究过程与实现--Defects4J数据集

### 第二板块：debug\_with\_llm

1. 在communicate.py中写了与deepseek交互的脚本，通过读取prompt.md的内容，并且加了一些限制语句来规范ds的回复。

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Communicate with LLM")
    parser.add_argument("--filepath", help="File path to the prompt")
    parser.add_argument("--folder", help="Folder path to the prompt")
    parser.add_argument("--prefix", help="Prefix for the prompt")
    parser.add_argument("--model", help="Model name")
```

根据测试用例和错误信息，问题出在`translate`方法没有正确处理`StringBuffer`类型的输入。修复方法是修改`lookupMap`的键类型为`String`，以便正确匹配输入。以下是修复后的代码：

```
```java
public int translate(final CharSequence input, final int index, final Writer out) throws IOException {
    int max = longest;
    if (index + longest > input.length()) {
        max = input.length() - index;
    }
    // descend so as to get a greedy algorithm
    for (int i = max; i >= shortest; i--) {
        final CharSequence subSeq = input.subSequence(index, index + i);
        final CharSequence result = lookupMap.get(subSeq.toString());
        if (result != null) {
            out.write(result.toString());
            return i;
        }
    }
    return 0;
}
```
```

主要修改是将`lookupMap.get(subSeq)`改为`lookupMap.get(subSeq.toString())`，确保使用String类型作为键来查找匹配项。这样就能正确处理StringBuffer类型的输入了。

my\_code/  
└─ \_\_pycache\_\_/  
└─ debug\_with\_llm/

ch.py

py

py



## 三，具体的研究过程与实现--Defects4J数据集

### 第二板块：debug\_with\_llm

2. （最dirty work的一集）将ds返回的补丁运用到源代码中并且测试。过程涉及截取代码，定位，替换等等操作，看似简单实则一点都不容易x

重点：要避免编译错误导致误判（）  
且该过程不能受到打乱顺序/消去信息的影响。

### 第三板块：统计数据

从log中统计数据，  
查看通过的测试数后  
制成图表进行对比分析



```
my_code/
├── __pycache__/
├── debug_with_llm/
│   ├── __pycache__/
│   ├── all_test_patch.py
│   ├── communicate.py
│   ├── constants.py
│   ├── test_patch.py
│   └── utils.py
└── generate_prompt/
```

## 三，具体的研究过程与实现--Bears Benchmark数据集

### 定位 buggy 代码

使用 `git diff --name-only <commit 2> <commit 3>` 来定位修改的代码文件，再具体比较文件来分析其中具体的修改位置，可能是class、function或者import.

比如：

@@ -141,0 +142,2 @@ public class BeanDeserializerFactory

@@ -95 +95 @@ public class BeanPropertyMap

@@ -194 +197,0 @@ public class CreatorCollector

### 提取相关测试信息

读取 <commit 4> 中的 bears.json 来获取测试用例信息，比如：

"testClass": "com.fasterxml.jackson.databind.creators.Creator1476Test"

"detail": "Could not find creator property with name 'intField' ....."

```
"tests": {  
  "overallMetrics": {  
    "numberPassing": 1569,  
    "numberRunning": 1570,  
    "numberFailing": 0,  
    "numberErroring": 1,  
    "failures": [  
      {  
        "failureName": "com.fasterxml.jackson.databind.JsonMappingException",  
        "isError": true,  
        "occurrences": 1  
      }  
    ],  
    "numberSkipping": 0  
  },  
}
```



## 三，具体的研究过程与实现--Bears Benchmark数据集

---

自动化脚本整合信息 流程：

- 1、从 bugs.json 文件中加载所有错误的元数据信息，查找与指定 bugId 对应的分支名称；
- 2、git checkout 包含指定错误的分支；
- 3、git log --reverse --oneline 列出相关的所有提交；
- 4、git checkout 包含错误源代码的提交以及修正后源代码的提交；
- 5、git diff 获取其中所需要的相关信息；

统一格式的prompt 格式：

"Buggy code"（原始源代码），分析 git diff 的内容，从 git checkout <commit 2>来获取；

"Failed test"（触发该 Bug 的测试用例），从 <commit 4> 的 bears.json 中获取；

"Error"（测试运行时抛出的异常类型及其简要信息），从 <commit 4> 的 bears.json 中获取；

"Error Code Block"（整个测试方法的代码），分析 bears.json 的内容，从 git checkout <commit 2> 中获取；

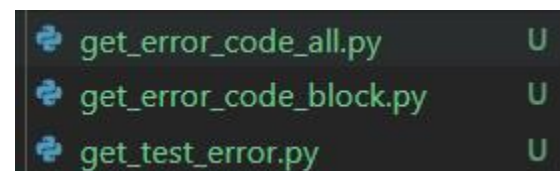
"Test line"（导致测试失败的具体断言或语句行）=> 最终整合到 prompt\_<bugId>.md 文件里

### 三，具体的研究过程与实现--QuixBugs数据集

---

#### generate\_prompt

1. 提取Buggy代码，运行自动化测试框架
2. 开发自动化脚本使用正则匹配提取错误信息，包括触发该 Bug 的测试用例、导致测试失败的具体断言或语句行、测试运行时抛出的异常类型及其简要信息。
- 3.在错误代码块方面，开发脚本使得对于提供了错误行信息的多函数复杂代码，可以利用ast方法解析并进一步提取错误代码块信息。
4. 将收集到的信息合并到类型为markdown的prompt文件中，同一代码的多个测试的结果进行合并
- 5.用交换脚本对于上述五种类型的信息进行重排



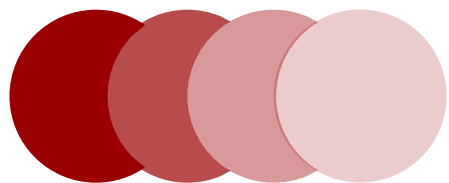
```
get_error_code_all.py  U
get_error_code_block.py  U
get_test_error.py  U
```

## 三，具体的研究过程与实现--QuixBugs数据集

---

### 交互与测试

1. 自动化提取reply中的正确代码信息并替换到源代码中
2. 将经过修复的代码整合进数据集文件夹并利用pytest框架进行自动化测试
3. 利用日志分析测试结果

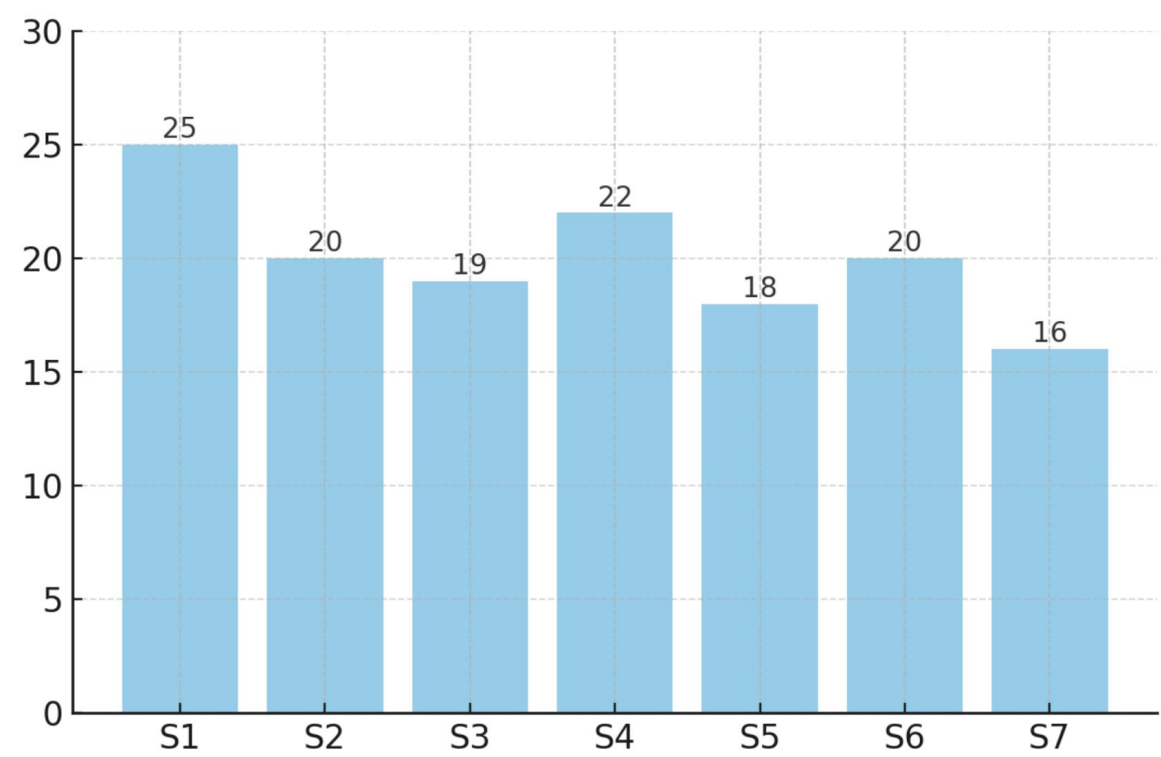


## 04 【结果分析及结论】

---

## 四，结果分析及结论——Defects4J数据集

- . 初始 ( Buggy code → Failed test → Test line → Error → Error Code Block )
- . 人类启发 ( Failed test → Error → Test line → Error Code Block → Buggy code )
- . 自底向上 ( Buggy code → Error Code Block → Test line → Failed test → Error )
- . **Test-first 策略** ( Error Code Block → Test line → Failed test → Error → Buggy code )
- . **Buggy-code 优先** ( Buggy code → Test line → Failed test → Error Code Block → Error )
- . 先报错后源码 ( Error → Test line → Failed test → Error Code Block → Buggy code )
- . **LLM 默认策略** ( Buggy code → Failed test → Error → Error Code Block → Test line )



| 提示顺序策略        | 调试成功数 (N=60) | 成功率   |
|---------------|--------------|-------|
| 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% |
| LLM 默认策略      | 16/60        | 26.7% |

## 四，结果分析及结论——Defects4J数据集

| 提示顺序策略        | 调试成功数 (N=60) | 成功率   |
|---------------|--------------|-------|
| 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% |
| LLM 默认策略      | 16/60        | 26.7% |

- 现象一**：“最开始”策略和“Test-first”策略表现突出，成功率明显高于平均水平，说明**将关键信息放在提示开头可能对LLM理解问题尤为有利**。
- 原因：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。
  
- 现象二**：“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，提示了**默认顺序可能并未优化关键信息的传递**
- 原因：将测试定位信息（Test line）放在提示的末尾是很低效的，模型往往难以及时利用这条关键信息，因而修复成功率大幅下降。

## 四，结果分析及结论——Defects4J数据集

| 提示顺序策略        | 调试成功数 (N=60) | 成功率   |
|---------------|--------------|-------|
| 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% |
| LLM 默认策略      | 16/60        | 26.7% |

- 现象三：人类启发策略的成功率为33.3%，并不如预期中突出。这可能暗示：人类直觉的调试顺序不一定最优于LLM
- 原因：LLM在预训练中习得的模式可能不同。例如，LLM可能经常见到的问题描述是先给代码片段再给错误（如论坛问答中，提问者常先贴出代码然后给出错误信息）
- 底层逻辑：预训练语料中的常见叙事顺序形成了概率先验；因此，LLM对输入顺序天然敏感

### 结论：

LLM 的“位置偏置 + 上下文竞争”机制决定了：关键信息越靠近序列两端越容易被注意并参与推理；若把错误日志或失败测试行放在末尾，中途的注意力已被后续 token 冲淡，从而错过修复线索，导致成功率下降。

## 四，结果分析及结论——Defects4J数据集

### 消融实验

结论一： error code block对于LLM的软件修复能力具有极大的贡献值，单独去掉后下降了16.7%

结论二： error code block与failed test+test line作用近似，二者可以保留其一。说明某些信息是相对的“冗余”的

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

(pp = 百分点; N = 60 个缺陷)

- 建议：
- 1. 尽量保留堆栈错误信息error和测试方法的代码，二者极大影响了LLM的软件修复能力
  - 2. 为了节省tokens数，我们可以在error code block与failed test+test line中保留一个



## 四，结果分析及结论——Bears Benchmark数据集

| 方法         | 信息顺序  | 通过数量 | 通过率   |
|------------|---|------|-------|
| 初始         | Buggy code → Failed test → Test line → Error → Error Code Block | 8    | 33.3% |
| 人类启发       | Failed test → Error → Test line → Error Code Block → Buggy code | 5    | 20.8% |
| 自底而上       | Buggy code → Error Code Block → Test line → Failed test → Error | 5    | 20.8% |
| Test-first | Error Code Block → Test line → Failed test → Error → Buggy code | 6    | 25%   |
| Code优先     | Buggy code → Test line → Failed test → Error Code Block → Error | 6    | 25%   |
| 报错优先       | Error → Test line → Failed test → Error Code Block → Buggy code | 6    | 25%   |
| LLM默认      | Buggy code → Failed test → Error → Error Code Block → Test line | 5    | 20.8% |

通过率比较低，可能是数据集的难度导致的，但仍然可以辅助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，于是没有对Bears Benchmark做进一步的消融实验；

## 四，结果分析及结论——Bears Benchmark数据集

| 方法 | 信息顺序  | 通过数量 | 通过率   |
|----|---|------|-------|
| 初始 | Buggy code → Failed test → Test line → Error → Error Code Block | 8    | 33.3% |

分析”初始“策略的优势：**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；

**先整体后局部、由广及深的信息呈现 => 减少模型理解的“跳跃”成本**

## 四，结果分析及结论（QuixBugs）

- . 初始（ Buggy code → Failed test → Test line → Error → Error Code Block ）
- . 人类启发（ Failed test → Error → Test line → Error Code Block → Buggy code ）
- . 自底向上（ Buggy code → Error Code Block → Test line → Failed test → Error ）
- . **Test-first 策略**（ Error Code Block → Test line → Failed test → Error → Buggy code ）
- . **Buggy-code 优先**（ Buggy code → Test line → Failed test → Error Code Block → Error ）
- . 先报错后源码（ Error → Test line → Failed test → Error Code Block → Buggy code ）
- . **LLM 默认策略**（ Buggy code → Failed test → Error → Error Code Block → Test line ）

发现一：“首因 + 近因”效应，Transformer 的自注意力在长上下文里呈 U-形利用率：开头和结尾的信息权重最高，中段信息易被忽略（“Lost-in-the-Middle”问题）在本数据集中也得到了体现。QuixBugs和前述数据集的主要区别在于代码属于轻量级，导致了Buggy code 和error code block的信息高度重合，没有能够很好地起到定位作用，反而可能造成信息冗余，这或许可以解释初始策略效果不如其他两个数据集明显

发现二：默认策略和人类启发策略同样没有很好的效果。

|               |       |        |
|---------------|-------|--------|
| 最开始           | 27/31 | 87.10% |
| 人类启发          | 26/31 | 83.87% |
| 自底向上          | 26/31 | 83.87% |
| Test-first 策略 | 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 style  | 27/31 | 87.10% |

## 四，结果分析及结论（QuixBugs）

消融实验 以 “最开始:Buggy code → Failed test→Test line→Error→Error Code Block” 的顺序为基础

- 1.消融['Error Code Block']
- 2.消融['Failed test','Test line']
- 3.消融['Failed test','Test line','Error Code Block']
- 4.消融['Failed test','Test line','Error']

发现三：当除去Failed test 和Test line信息时，LLM表现明显优于其他策略。结合实际的prompt内容，猜测或许这两个信息是冗余度较大的信息，但是本数据库测试主要集中在assert test，或许造成了‘Failed test 和Test line单一，多几种测试预言或许情况就不同了

|     |       |        |
|-----|-------|--------|
| 最开始 | 26/31 | 83.87% |
| 1   | 24/31 | 77.42% |
| 2   | 29/31 | 93.55% |
| 3   | 25/31 | 80.65% |
| 4   | 26/31 | 83.87% |

## 四，结果分析及结论（三个数据集结果对比）

### 提示顺序对 LLM 修复表现的影响具有一致性趋势

策略（Buggy Code → Failed Test → Test Line → Error → Error Code Block）

在三个数据集中均表现良好：

Defects4J中为最高（41.7%）

Bears中为相对最优（尽管整体成功率低）

QuixBugs中表现接近最优（83.87%），但被“Buggy-code 优先”略微超过

### 不同数据集下的“提示偏好差异”反映了模型与数据的结构互动

不同数据集的结构特性（如代码体量、错误模式、测试多样性）会影响提示中哪类信息更关键

轻量级程序中提示顺序差异影响较小，复杂系统中提示设计影响显著。

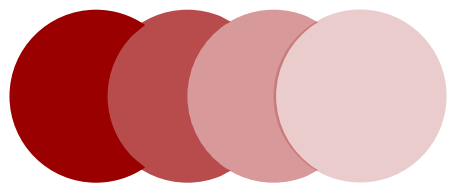
### 不同信息对模型修复能力的边际价值可区分

在 Defects4J 和 QuixBugs 的消融实验中均发现：

去除 Error 或 Error Code Block → 成功率大幅下降

去除 Failed test + Test line → 影响较小，部分情况下表现反而更好（如 QuixBugs）





## 05 【反思和未来可能继续优化的工作】

---

## 五，反思和未来可能继续优化的工作

---

1. 虽然从实验结果上看，在不同的数据集上较优的顺序基本上相同，但是由于数据集总量还是太少，仍然需要扩大数据量来证明可信度。
2. 由于没法搞定chatgpt的api所以只暂时使用了ds作为修复的LLM，以后应该选择多个广泛使用的LLM，探究是否同一套信息组织的规则适用于大部分的LLM，因为不同模型的预训练、训练方式不同。
3. 当前考察了代码是否通过测试，在代码风格、代码复杂度、可维护性等方面需要涉及更全面深入的评估体系
4. 未进行迭代测试，或许迭代会获得更好的效果，可能可以覆盖确实某些信息的缺陷

# 感谢观看

信息科学技术学院 22级 元铭宇  
信息科学技术学院 22级 杨世航  
信息科学技术学院 22级 蒋康悦