ChatRepair

中心思想：首次将对话式交互引入自动程序修复。ChatRepair 使用 ChatGPT 作为对话代理，流程：首先将失败测试的信息（包括断言失败消息、堆栈等）与buggy代码一起作为初始提示，生成一个候选补丁；若补丁未通过测试，则将该补丁及对应的测试失败信息反馈给ChatGPT，让其“学习”先前补丁的错误，再生成新补丁

实验设计：比较ChatRepair与传统生成验证式方法在修复数量和代价上的差异，统计ChatRepair成功修复的Bug数量和调用LLM的次数/成本。同时，与以往APR工具（如基于Codex的修复）对比修复率，并分析多轮对话对修复质量的贡献

关键结论：通过将测试反馈融入对话迭代，ChatRepair 显著提升了修复效果和效率。在Defects4J上，ChatRepair 成功修复了162个/337个缺陷

具体的流程：ChatRepair通过初始化系统消息和构建初始提示来准备修复任务，提示内容包括有bug的函数和测试失败信息。ChatGPT生成候选补丁后，进入对话阶段，通过验证补丁是否通过测试来学习。如果补丁未通过，ChatRepair会提供反馈并请求ChatGPT生成新补丁，避免之前的错误。这个过程会重复，直到生成合理补丁或达到最大对话轮数。获得合理补丁后，ChatRepair会从中学习，生成更多可通过测试的补丁变体，以增加生成正确补丁的机会。

要点

1. 使用“行级修复”，特别是填空式或“完形填空”风格的

> 在这种方法中，系统将有缺陷的代码行视为需要填补的空白，并利用上下文信息（即该行前后的代码）来预测并生成正确的代码片段以填补这些空白

2. 有三种修复场景
    - 单行修复
    - 单块修复
    - 单函数修复
3. 该论文中，将bug添加到提示中包括了：
    - **历史bug修复示例**——让模型更好理解修复任务，学习期望的输出格式。
    - **用特殊标记替换掉有bug的那行**——引导填入。
    - 失败测试信息
        - 测试名称
        - 导致失败的相关代码行
        - 产生的错误信息
        - 正确情况下的预期输出和函数行为信息

    - $C(p|pre, infill, suf, f_0, I_{fill})$ pre 和 suf 为buggy代码中填空前后的上下文，infill 表示用于替代buggy行的填空标记，f_0 是构造的测试失败信息，I_fill 是具体的填空指令
      json示例
4. 生成合理补丁之后，甚至还有**替代补丁**，产生额外补丁
5. 设置了最大修复尝试次数
6. 数据集
    - Defects4j， 分为了1.2和2.0，其中1.2又划分为了单函数修复、单块修复、单行修复；而2.0则选取了新增9个项目的bug
    - QuixBUgs，分为python和java两种，也分别分为了单函数、单块和单行
7. baseline，可以暂时不关注，等我们能跑通我们自己的一套时可以去对比。
8. **消融实验**（重点关注）
    - BasePrompt：仅提供 buggy 代码，提示模型“这里有 bug，请修复”；
    - TestName+ErrMsg：在提示中加入失败测试的名字（如 testGetCategoryIndex）与错误信息（如 NullPointerException）；
    - TestName+ErrMsg+FailLine：进一步加入了测试中触发 bug 的具体代码行（如 assertEquals(-1, empty.getCategoryIndex())）；
    - TestName+ErrMsg+TestBody：提供整个失败测试函数体，而非仅失败代码行。
9. 反馈方式：
    - BaseFeedback（基础反馈）：仅告诉模型当前补丁不正确，不提供其他信息；
    - TestName+ErrMsg：提供失败测试名称（如 testGetCategoryIndex）和测试错误信息（如 NullPointerException）；
    - TestName+ErrMsg+FailLine：进一步包括测试中发生错误的确切代码行；
    - Dynamic（动态反馈）：这是默认方案。只有当生成的新补丁导致与之前不同的测试失败时，才提供新的 test name、error message 和 fail line。这种方式可以明确告知模型：它是否取得了某种修复进展（如不再空指针崩溃，但在另一个测试中失败）。

example.txt

Toggle

将LLM用于精细粒度的缺陷定位，然后再用于修复，分别优化两步并集成。
三个模块

- Bug定位模型：基于 CodeT5 Encoder，精确定位 Token 级别的 Bug 起止位置；
- Bug修复模型：使用多个 LLM（如 CodeParrot、CodeGen 等）生成补丁；
- 修正模块（Adjustment Model）：解决定位与修复模型之间 Tokenizer 不一致问题。

1. Prompt 1：完整函数替换（不使用任何位置偏置信息）
    - 描述：给出完整的 buggy 函数，要求模型生成完整的修复函数。

- 优点：结构简单，直观。
- 缺点：模型需要从头生成整个函数，包括很多无需修改的部分，容易引入额外错误。

2. Prompt 2：提供 shared prefix，预测 truncated 修复函数
   - 描述：在输入中提供共享前缀（如函数定义等），buggy 函数中保留前缀部分，模型只需生成从 bug 起始处的修复代码。
   - 优点：利用了前缀信息作为上下文，减少模型生成内容。
   - 缺点：buggy 函数中和修复函数中会重复出现前缀，有冗余。
3. Prompt 3：去掉 shared prefix，引导模型从 bug 开始生成修复代码
   - 描述：输入中省略前缀，buggy 函数从 bug 开始；模型预测修复代码后再拼接前缀得到完整函数。
   - 优点：通过结构引导模型专注 bug 位置，提高修复效率。
   - 缺点：只需要预测 bug 的起始位置，适合定位不完全精准的情况。
4. Prompt 4：去掉 shared prefix 和 suffix，模型仅需预测 bug 的替换部分
   - 描述：输入中不包含前缀和后缀，只包含 buggy 部分；修复时将替换段生成后再拼接前后缀。
   - 优点：生成内容最少、最精准，具有最强的位置引导。
   - 缺点：需要准确预测 bug 的起始和结束位置；如果预测不准，修复效果反而下降。

一些附加的

关于baseline如何设置，即如何把**加入了LLM的软件修复**与**未加入LLM的软件修复**进行对比