arXiv:2304.00385v2 [cs.SE] 9 Dec 2024

让对话持续下去：使用 ChatGPT 修复 337 个缺陷中的 162 个，每个仅需 0.42 美元

Chunqiu Steven Xia 伊利诺伊大学厄巴纳-香槟分校
chunqiu2@illinois.edu

Lingming Zhang 伊利诺伊大学厄巴纳-香槟分校 li
ngming@illinois.edu

摘要

自动化程序修复（APR）旨在自动为有缺陷的程序生成补丁。传统的 APR 技术依赖人工编写或挖掘的缺陷修复模式，因此补丁种类有限，也难以推广到其他缺陷/修复类型。为解决这一局限，近期的 APR 研究致力于利用现代大语言模型（LLM）直接生成 APR 补丁。这类基于大语言模型的自动程序修复工具首先根据原始有缺陷的代码构造输入提示，然后查询大语言模型，让其在缺陷位置填入正确代码（填空式 APR），或生成全新的代码片段作为补丁。尽管基于大语言模型的自动程序修复工具能够取得最先进的结果，却仍遵循经典的生成与验证（G&V）修复范式：先从同一个初始提示中采样生成大量补丁，再逐一验证。这不仅会产生许多重复且不正确的补丁，还会忽略测试失败和合理补丁中至关重要、此前却未受重视的信息。

为了解决上述局限性，我们提出了 ChatRepair，这是首个完全自动化、由对话驱动的自动程序修复方法，通过交替进行补丁生成与即时反馈，以对话方式执行自动程序修复。ChatRepair首先向大语言模型提供相关测试失败信息，然后从同一缺陷先前修复尝试的失败与成功中学习，从而实现更强大的自动程序修复。对于未能通过所有测试的早期补丁，我们将错误补丁与其对应的相关测试失败信息结合起来，为大语言模型构造新的提示，以生成下一个补丁。这样可以避免重蹈覆辙。对于通过所有测试的早期补丁（即合理补丁），我们会进一步要求大语言模型生成原始合理补丁的替代变体。这样，我们就能在早期成功的基础上继续改进并从中学习，生成更多合理补丁，从而提高获得正确补丁的机会。虽然我们的方法具有通用性，但我们使用最先进的、基于对话的大语言模型 ChatGPT 实现了 ChatRepair。在广泛研究的 Defects4J数据集上的评估表明，ChatRepair 实现了新的最先进修复性能，在 Defects4j 1.2 和 2.0 上分别实现了 114 个和 48 个正确修复。根据访问 ChatGPT 的成本计算，我们可以将 337 个缺陷中的 162 个修复好，每个仅需 0.42 美元！

1 引言

自动化程序修复（APR） [16, 18] 是一种有前景的方法，可自动为软件缺陷生成补丁。传统的自动程序修复工具通常采用生成与验证（G&V） [37] 范式，先生成大量候选补丁，再根据原始测试套件逐一验证，以找出一组合理的补丁（即通过所有测试的补丁）。随后，开发人员会检查这些合理补丁，以找出能够正确修复底层缺陷的正确的补丁。传统自动程序修复技术可分为基于模板 [17, 20, 33, 34, 41], 、基于启发式 [28, 30, 55]和基于约束 [12, 27, 36, 42]的自动程序修复工具。在这些传统技术中，基于模板的自动程序修复工具通过使用手工编写或挖掘得到的修复模板来匹配并修复错误代码模式，一直被视为最先进的方法 [3, 17, 33]。然而，基于模板的工具难以生成多样化的补丁，因为它们无法轻易泛化到预定义模板列表之外的缺陷和模式。

为解决传统自动程序修复技术的局限，研究人员提出了利用深度学习进展的基于学习的自动程序修复方法。这类方法主要基于神经机器翻译（NMT）或大语言模型（LLM）。基于神经机器翻译的自动程序修复工具 [11, 22, 31, 39, 61, 62,65] 将修复视为一项翻译任务，通过使用历史缺陷修复数据集训练神经机器翻译模型 [53] ，将有缺陷的代码转换为正确代码。然而，这类工具严重依赖其训练数据，而这些数据是通过从开源代码库中抓取错误修复提交获得的。这意味着训练数据集可能含有噪声 [23] （即包含无关的提交或更改），而且这类基于神经机器翻译的方法可能无法泛化到有限训练数据中未见过的错误修复类型。

最近，研究人员开始直接利用先进的大语言模型进行自动程序修复 [25, 46, 57, 58]。现代大语言模型在数十亿个开源代码片段上进行训练，在许多代码相关任务中展现出令人瞩目的性能 [5, 10, 15, 60], ，并能根据周围上下文直接生成代码（这得益于代码的自然性 [19, 48]）。AlphaRepair [58] 提出了首个完形填空式（或填充式）自动程序修复方法：移除有缺陷的代码，由大语言模型根据前缀和后缀上下文直接预测正确代码。近期研究还将基于大语言模型的自动程序修复应用于自动补全一行正确代码 [25] 或生成完整的修复后函数 [46]。一项更全面的研究 [57] 探讨了将更大的大语言模型和不同的大语言模型架构（即生成式和填充式）应用于自动程序修复，并表明基于大语言模型的自动程序修复工具在许多自动程序修复任务上能够达到新的最先进性能。与此同时，现有基于大语言模型的自动程序修复流程仍存在以下局限：

1）缺少测试失败信息。当前基于大语言模型的工具并未考虑原始缺陷暴露测试中丰富的信息。这些信息不仅可以帮助大语言模型理解受测源代码的含义，还可以提供帮助和提示，给出具体的代码片段。图 1 展示了一个缺陷修复示例，以及原始测试失败信息。我们看到，修复方法是将追加的字符串替换为`"\\000"`。对于基于大语言模型的方法而言，这可能是一个极其困难的修复，因为这个独特的字符串并不是预训练过程中常见的字符串，而且

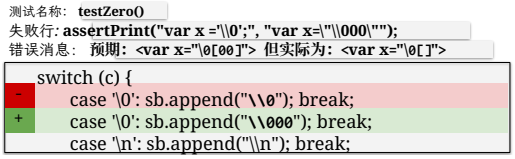


图1：包含原始测试用例信息的缺陷修复示例

当前函数上下文中也没有其他三重字符串的示例（"\\xxx"）。然而，从测试中的失败行及相应的错误消息可以看出，测试期望输出包含三个零，甚至还包含一个代码片段（"\\000"），而这个片段正是补丁中直接使用的内容！大语言模型在处理和利用测试失败日志这类非结构化、复杂信息方面表现出强大的能力。若忽略这些信息，基于大语言模型的工具可能会浪费大量时间生成无关补丁。

2）重复采样。当前基于大语言模型的方法首先使用原始有缺陷的代码构建输入提示，然后要求大语言模型填充正确代码（即完形填空式自动程序修复），或生成一个全新的修复后函数 [46, 57]。利用初始提示，基于大语言模型的技术会对大语言模型进行多次采样，以生成许多补丁，这类似于传统的G&V范式程序修复。然而，由于每次采样都是完全独立的，大语言模型并不知道任何先前生成的补丁。因此，基于大语言模型的工具可能会生成许多此前已经被判定为不正确的重复或相似补丁，从而浪费 API访问的美元成本或 GPU执行的时间。此外，这种重复采样过程也与人类开发人员修复缺陷的方式大相径庭：人类开发人员会基于先前失败尝试中的知识和经验不断迭代，提出下一个可能的补丁。

3）忽视有价值的合理补丁。除了未能利用过去的错误补丁外，当前基于大语言模型的自动程序修复工具也无法有效利用此前生成的合理补丁。研究表明，合理补丁很有价值，因为它们通常与实际正确补丁的位置相似 [17, 38]。此外，我们进一步假设，合理补丁还可能包含通过所有测试所需的关键代码要素，也可能帮助大语言模型更好地学习如何通过所有测试，从而生成更多合理补丁（进而提高生成正确补丁的可能性）。现有的基于大语言模型的自动程序修复方法忽视了这些有价值的合理补丁信息，并在生成合理补丁后从头开始，因此可能错失正确修复更多缺陷的机会。

我们的工作。 我们提出了 ChatRepair——一种完全自动化的对话驱动自动程序修复方法，通过交替进行补丁生成和即时反馈，以对话式风格生成补丁。虽然我们的构想具有通用性，但为了构建 ChatRepair，我们采用了近期开发的、当前最先进的基于对话的大语言模型——ChatGPT [50][1]，。该模型不仅基于数十亿个代码片段训练，还专为对话式交互设计，以更好地理解指令。ChatRepair 首先提取相关的测试失败信息，将其作为初始提示，为 ChatGPT 提供更多上下文信息以进行自动程序修复。此外，ChatRepair 还会利用针对同一缺陷的早期补丁尝试的失败和成功经验，从而实现更强大的自动程序修复。对于早期补丁未能通过所有测试的情况，我们会将错误补丁及其对应的测试失败信息结合起来，为大语言模型构建新的提示，以生成下一个补丁。这样可以避免重蹈覆辙。对于通过所有测试的早期补丁（即合理补丁），我们还会要求大语言模型生成原有合理补丁的替代变体。这样，我们就能在先前成功的基础上继续改进并吸取经验，生成更多合理补丁，从而提高获得正确补丁的机会。由于我们的方法使用 ChatGPT 模型，我们还会计算修复一个缺陷所需的 ChatGPT API 查询的美元成本。令人惊讶的是，我们发现借助 *ChatRepair*，每修复一个缺陷仅需 *0.42* 美元，共可修复 *337* 个缺陷中的 *162* 个。[2]

本文作出以下贡献：

- **维度**。 我们为全自动程序修复开创了一种新的对话驱动范式。我们的工作首次表明，我们能够以对话方式有效利用此前被忽视的测试失败信息和先前的补丁尝试，从而提示大语言模型生成更多正确补丁。此外，我们还展示了基于对话的大语言模型在自动程序修复领域的广阔前景。

- **技术**。 我们开发了 ChatRepair，这是一种全自动对话驱动型自动程序修复工具，使用最新的 ChatGPT 模型。具体而言，我们会自动提取有关初始测试失败的简洁且相关的信息，以及先前的补丁尝试，并将其用于提示 ChatGPT，从而实现有效的自动程序修复。

- **评估**。我们在广泛研究的 Defects4j 1.2、2.0 [24] 和 QuixBugs [32] 数据集上，将 ChatRepair 与当前最先进的基于学习和传统自动程序修复工具进行了比较。ChatRepair 在 Defects4j 1.2 和 2.0 上分别修复了 114 个和 48 个缺陷，取得了新的最先进修复结果；相比此前表现最佳的基线，分别多修复了 15 个和 17 个缺陷。此外，我们进行了广泛的消融研究，以展示利用丰富的语义测试失败信息以及 ChatRepair 的对话范式所带来的修复效果提升。

2 背景与相关工作

2.1 大语言模型

大语言模型（LLM） [5] 近年来，自然语言处理（NLP）的进展使模型规模得以扩展至数十亿参数，并使用数十亿个训练样本，推动了大语言模型性能的飞速提升及其应用的迅速普及。由于大语言模型经过通用化训练，能够掌握不同领域的知识，因此可通过微调 [47] 或提示 [35] 来执行下游任务。微调是指使用特定训练数据集更新模型参数，以适应某一特定下游任务。然而，微调不仅成本高昂，因为它需要额外训练模型；在缺乏足够训练数据集的情况下，也可能无法实施。另一方面，提示无需任何训练，只需向大语言模型提供下游任务的自然语言描述（例如，生成文档字符串），并可选择附上少量任务示例作为输入，即可直接使用大语言模型。

大语言模型基于 Transformer 架构 [54] 构建，并可根据所使用的组件进行分类。仅解码器模型

[1]虽然修复过程使用了 ChatGPT，但本文没有任何部分由 ChatGPT 撰写。

[2]这是对先前一项经典研究的引用，该研究针对自动程序修复（APR）开展， [29] 并未使用 ChatGPT。

（例如 Codex [10]和 CodeGen [44]）是广受欢迎的基于GPT的模型，通过以因果语言建模为目标进行训练，学习根据所有先前的左侧上下文预测下一个词元的概率。仅编码器模型（例如 CodeBERT [14]）和编码器-解码器模型（例如 CodeT5 [64]）分别以掩码语言建模（MLM）或掩码跨度预测（MSP）为目标进行训练：一小部分词元（例如 15%）会被替换为掩码词元或掩码跨度词元，然后训练大语言模型根据双向上下文恢复被掩码的词元。

除这些传统的大语言模型外，最近，研究人员还提出了使用强化学习训练的大语言模型，这些模型与人类偏好更加一致 [45, 50, 66]。其中包括 InstructGPT [45] 和 ChatGPT [50]，它们首先基于经过自回归生成预训练的预训练模型进行初始化，然后使用基于人类反馈的强化学习（RLHF） [66]进行微调。RLHF 首先使用一个由少量提示（输入）及期望输出（人工撰写）组成的数据集对基线模型进行微调。随后，通过从微调后的大语言模型中采样多个输出，并让人工标注员对每个输出分别进行排序，在更大规模的提示集上训练一个独立的奖励模型。最后，应用强化学习（例如近端策略优化 [49]），根据奖励模型计算生成输出的奖励，并相应地更新大语言模型权重。通过使用人类偏好进行微调得到的大语言模型，已展现出更好地理解复杂输入提示并遵循指令以执行各种任务的能力 [2, 45, 50]。具体而言，ChatGPT 因其对对话的关注而受到广泛关注：它专门通过对对话进行训练，并且能够跟踪和引用先前的对话。

在这项工作中，我们继续推进正在进行的研究 [59]：我们提出了一种更全面的方法，纳入了更稳健的反馈（包含多维度的相关测试信息），并旨在从失败的补丁和合理补丁中学习；我们还全面评估了基于基于人类反馈的强化学习的大语言模型修复真实世界系统（例如 Defects4j [24]）中缺陷的能力。这项工作首次证明，像 ChatGPT 这样强大的、基于人类反馈的强化学习大语言模型可以直接用于全自动化对话驱动程序修复，并且其表现显著优于所有现有的自动程序修复技术。

2.2 自动程序修复

自动化程序修复（APR）可以根据给定缺陷的潜在故障位置，为开发人员生成补丁。经典的自动程序修复技术主要分为启发式 [28, 30, 55], 基于约束 [12, 27, 36, 42]和基于模板 [17, 20, 33, 34, 41] 几类。由于修复的缺陷数量众多，基于模板的自动程序修复工具已被视为最先进的方法。同时，这类自动程序修复工具利用人工定义或自动挖掘的模板，先匹配潜在的错误代码模式，再应用相应的修复。不过，基于模板的工具只能修复其有限模式集合涵盖的缺陷，因此无法泛化到其他类型的缺陷或修复。为解决这一问题，研究人员利用深度学习的最新进展，提出了基于学习的自动程序修复技术。近年来，基于神经机器翻译的技术得到了广泛研究，例如 SelfAPR [61], RewardRepair [62], Recoder [65], CURE [22] 和 CoCoNuT [39]。这些方法有一个共同的认识，即自动程序修复可以视为神经机器翻译问题，旨在将有缺陷的代码转换为正确代码。通过在由有缺陷代码片段及其修复后代码片段组成的数据集上训练神经机器翻译模型，这些方法可以学习生成补丁。尽管这类基于神经机器翻译的技术效果显著，但它们严重依赖历史缺陷修复训练数据集，这些数据集通常通过抓取开源代码库中的缺陷修复提交来构建。因此，训练数据可能包含各种噪声，例如无关的更改或提交；此外，为了减少假阳性，这些数据集主要关注较小的提交，进一步限制了用于训练的缺陷和修复类型。因此，基于神经机器翻译的自动程序修复技术在可修复缺陷的类型和数量方面仍然受限。

为进一步克服基于神经机器翻译的工具的局限性，研究人员还探索了直接利用大语言模型合成正确补丁的可能性。大语言模型通过对大量开源代码片段进行预训练，无需将有缺陷的代码翻译为正确代码，即可根据周围上下文直接合成正确代码。AlphaRepair [58] 是首个用于完形填空式（或填充式）自动程序修复的工具：它先将有缺陷的代码行替换为掩码词元，再利用大语言模型根据上下文直接填入正确代码。AlphaRepair 首次表明，基于大语言模型的自动程序修复在真实世界系统上的表现可以超过受到广泛研究的基于神经机器翻译的自动程序修复技术。Prenner 等人 [46]和 Kolak 等人 [25] 也直接使用 Codex [10]，根据原始有缺陷的函数生成修复后的函数，或在一个小型数据集（QuixBugs [32]）上根据前缀代码自动补全单行代码。最近，Xia 等人 [57] 基于各种大语言模型（例如 Codex [10],、GPT-NeoX [4],、CodeT5 [64], 和 InCoder [15]），对基于大语言模型的自动程序修复技术进行了广泛研究，并进一步证明了这类技术的优越性。尽管基于大语言模型的自动程序修复成果令人鼓舞，但现有技术仅关注待修复的源代码，没有考虑测试失败信息中丰富的语义。此外，以往基于大语言模型的技术会持续从相同的初始提示中采样，无法利用先前失败的补丁或合理补丁中的知识。在 ChatRepair 中，我们基于正在进行的研究 [59]，通过引入一种基于对话的修复范式，将补丁生成历史与即时验证反馈结合起来用于自动程序修复，从而克服以往基于大语言模型的工具的局限性。

以往的自动程序修复工具也曾利用简单的补丁执行或测试信息进行自动程序修复。GenProg [30] 是一种经典的自动程序修复工具，它使用进化算法将通过更多测试的候选补丁组合在一起。基于约束的自动程序修复工具 [12, 13, 36, 43]利用底层测试代码来提取并构建用于补丁合成的约束。近年来，RewardRepair [62] 提出使用基于训练集中补丁是否通过编译或测试执行的奖励函数来训练神经机器翻译模型。SelfAPR 是另一种基于神经机器翻译的自动程序修复工具，它将暴露缺陷的测试错误与原始有缺陷的代码一同编码为自动程序修复的输入。同时，据我们所知，ChatRepair 是首个针对对话式自动程序修复，为验证的每一个补丁利用详细反馈（例如相关测试代码和错误消息）的工作。此外，ChatRepair 直接利用大语言模型来处理测试反馈提取，该过程完全自动化、可泛化，并且能够理解深层语义信息。研究人员

也曾尝试将类似对话的方法用于程序修复/合成 [1]。然而，它们通常需要人工反馈，而 ChatRepair 则是完全自动化的。

3 方法

我们提出了 ChatRepair，这是一种完全自动化的对话驱动程序修复技术，它融合了多个维度的反馈信息，通过反复查询模型来生成补丁。与现有基于大语言模型的自动程序修复技术直接根据有缺陷的代码生成补丁不同，ChatRepair 还提供有价值的测试失败信息，以进一步协助大语言模型生成补丁。此外，与以往的基于大语言模型的自动程序修复技术持续从同一提示中采样不同，ChatRepair 会记录对话历史，并通过提示从针对同一缺陷的早期失败和成功的补丁尝试中继续学习。这样，ChatRepair 既能避免以往的失败，也能利用先前的成功（例如，合理补丁），从而更有效地进行自动程序修复。因此，ChatRepair 最大限度地提高了获得真正正确补丁的能力，以正确修复底层缺陷。虽然我们的方法具有通用性，可使用不同的大语言模型，并适用于各种不同的修复场景，但在本研究中，我们使用了专为对话交互设计的最先进的 ChatGPT 模型 [50] 。

图2通过一个示例性修复案例展示了 ChatRepair 的整体流程。 表示用于初始化模型、使其执行特定任务的系统消息；表示 ChatRepair 提供给大语言模型的提示和反馈； 表示 ChatGPT 给出的输出响应。首先，ChatRepair 使用系统消息“你是一个自动化程序修复工具”初始化 ChatGPT，为修复任务做好准备。然后，我们为 ChatGPT 构建初始提示，其中包含待修复的有缺陷的函数，以及用于修复缺陷的相关测试失败信息（第3.1节）。在使用初始提示请求 ChatGPT 生成潜在补丁后，我们进入对话阶段，首先从过去的失败中学习（第3.2节）。具体而言，我们使用原始测试套件评估生成的补丁，查看该补丁能否通过之前失败的测试。如果不能，ChatRepair 会立即提供反馈，生成一条包含相关失败信息（例如测试失败或编译错误消息）的响应，并重新向 ChatGPT 发出请求以生成新补丁，同时尽量避免重复类似的失败。此过程会重复进行，直到生成合理补丁或达到最大对话长度。获得合理补丁后，ChatRepair 会尝试从这些成功中学习，生成更多能够通过测试套件的合理补丁（第3.3节）。ChatRepair 会向 ChatGPT 提供先前生成的合理补丁，提示其生成更多备选合理补丁。通过这一过程，ChatRepair 可以获得多个合理补丁，从而提高获得正确补丁的机会。接下来，我们将更详细地介绍各个步骤。

3.1 初始输入

首先，我们使用原始的有缺陷项目和缺陷构建发送给 ChatGPT 的初始提示，以启动修复过程。们遵循先前基于学习的自动程序修复工具 [22, 58, 62] ，并主要关注行级修复（具体而言，采用填充或完形填空式自动程序修复，因为这已被证明是最先进的方法 [58]）。

与此同时，ChatRepair 具有通用性，也可用于各种不同的修复场景（参见图2右上角的其他提示），我们将在后续章节中对此进行更详细的评估。

图2展示了一个初始提示示例。在将待修复的目标缺陷加入提示之前，我们先加入同一有缺陷项目中的若干历史缺陷修复示例。这样做可以引导模型执行修复任务，并使其学习该任务所需的输出格式（即补丁）。在少样本示例之后，我们将待修复的原始目标有缺陷函数及缺陷位置作为输入。我们用填充位置指示符 (≫ [填充] ≪) 替换函数中的有缺陷代码，并在提示的后文中引用该指示符，指示模型填入正确代码。随后，我们将被填充位置指示符替换掉的原始有缺陷代码行提供给模型，因为该代码行也能为候选补丁的形式提供有用信息。接下来，我们提供其他相关信息，帮助 ChatRepair 修复缺陷。我们的方法使用从暴露原始缺陷的失败测试中提取的信息。这些缺陷暴露测试包含丰富的语义信息和提示，可帮助生成用于修复缺陷的正确补丁 [38]。

ChatRepair 会使用失败测试中的各种信息，包括：1）测试名称；2）触发测试失败的相关代码行；3）产生的错误消息。失败测试的名称可以作为被测函数的简短摘要 。在图 2 的示例中，失败测试为`testGreatestSubtypeUnionTypes5()`，这表明我们正在测试一项与从联合类型中确定最大子类型相关的功能。相关测试代码和错误消息提供了测试失败原因的具体信息。在该示例中，相关测试代码和错误消息告诉模型，我们正在比较一个`NO_OBJECT_TYPE`，但源代码函数错误地返回了一个`None` 对象。此类失败测试信息不仅能让模型更深入地了解源代码的功能，还能提供有关预期输出和函数用法的具体信息，帮助模型生成正确的修复方案。请注意，如果有多个失败测试，ChatRepair 只会提供其中一个测试的信息，以使初始提示保持简洁。最后，我们会在初始提示末尾指示模型生成正确的代码行，以替换填充位置处的有缺陷代码。设 C 为 ChatGPT，它输出生成序列的概率， pre 和 suf 分别是移除有缺陷代码行后的前缀和后缀， $infill$ 是替换有缺陷代码行的填充词元， f_0 是构造的失败测试信息， I_{fill} 是填充指令提示。生成的补丁 p 可形式化为以下条件概率：$C(p|pre,\ infill,\ suf,\ f_0,\ I_{fill})$

据我们所知，ChatRepair 是首个通过纯提示方法应用这些测试失败和错误消息的工作：它将失败信息的自然语言描述（例如，`代码在此测试中失败：{failure_test}`）作为强大的 ChatGPT模型的输入。与以往利用测试执行信息进行修复的方法不同，后者依赖自定义编码或手工设计的启发式方法；ChatRepair 通过提示使用 ChatGPT，不仅适用于不同的编程语言，也不受测试信息类型的限制。

保持对话持续进行：使用 ChatGPT 修复 337 个缺陷中的 162 个，每个仅需 0.42 美元

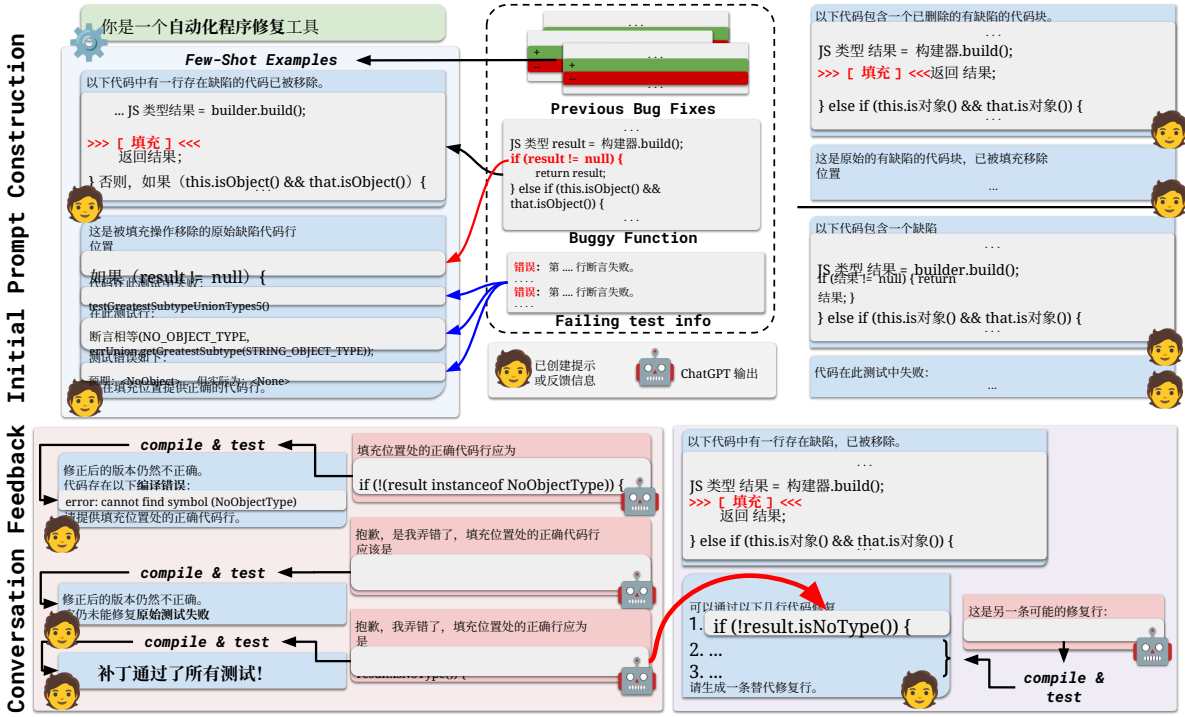


图2：ChatRepair 概览

算法1： 对话式修复

```
函数对话式修复：
    输入 ：初始提示（初始提示），原始失败（原始失败测试信息），测试套件（测试套件），ChatGPT，
           最大对话长度（最大对话长度），最大尝试次数（最大尝试次数），AltInstruct（合理补丁提示）
    输出：补丁列表（合理补丁），成本（总成本）
    补丁列表、当前尝试次数、成本 ← NONE、0、$0
    当 当前尝试次数 < 最大尝试次数且 补丁列表 为 NONE 时
        当前长度 ← 0
        输入 ←初始提示
        当当前长度 < 最大对话长度 时
            补丁、成本 ← ChatGPT（输入）
            测试结果 ← 验证（补丁，测试套件）
            如果测试结果是通过 那么
                补丁列表 ← [ 补丁 ]
                中断
            否则如果测试结果 是原始失败 那么
                反馈 ← "仍未修复原始失败"
            else
                反馈 ← 构造提示 (测试结果)
            输入 ← { 输入, 补丁, 反馈 }
            当前尝试次数 ←当前尝试次数 +1
            当前长度 ← 当前长度 +1
    如果 补丁列表 不为 NONE 则
        当 当前尝试次数 < 最大尝试次数执行
            输入 ← { 初始提示, 补丁列表, 替代指令 }
            补丁，费用 ← ChatGPT（输入）
            测试结果 ← 验证（补丁，测试套件）
            如果测试结果 是通过 且补丁不在补丁列表 则
                补丁列表 ← 补丁列表 | [补丁 ]
            当前尝试次数 ←当前尝试次数 +1
    返回补丁列表，成本
```

3.2 对话式修复

我们首先使用第3.1节中设计的初始提示查询 ChatGPT，以获取模型输出并提取候选补丁。然后进入该方法的对话环节，在此过程中交替进行补丁生成和测试验证反馈，以对后续生成提供提示，形成对话式流程。模型生成的每个补丁之后都会立即进行补丁验证：编译该补丁，并在测试套件上运行。如果补丁未能通过测试，我们会将错误补丁和失败测试一并纳入提示，构造详细的反馈信息，用于下一轮补丁生成。与初始提示类似，测试失败信息有助于模型理解失败原因，并为生成正确的修复提供指导。在对话步骤中，我们进一步将测试失败信息与先前的错误补丁结合起来，不仅避免生成更多类似的错误补丁，也从先前生成过程中的错误中学习。我们重复这一过程，直到生成能够通过整个测试套件的合理补丁。

更确切地说，我们将一次**对话轮次**定义为一次补丁生成及对该候选补丁的验证反馈（即 { , }）。在一次修复对话中，ChatGPT生成的下一个补丁会以初始提示与所有先前对话轮次的串接内容作为提示。例如，第3个补丁$_3$ 的输入为 { o,1, 1, 2, $_2$}。设 C 为输出序列生成概率的 ChatGPT模型，I为初始提示，$(_n = \{p_1, \ldots, p_{n-1}, f_{n-1}\})$为同一对话中先前生成的补丁$p$及反馈信息 f。生成下一个补丁的过程可形式化为条件概率：$C(p_i|I, Q_{<i})$

由于ChatGPT（以及其他大语言模型）的上下文窗口大小有限 [7],，也就是说它无法接收任意长度的输入，因此我们将**对话长度**（即单次连续对话中的轮次数）作为另一项停止标准：达到最大对话长度后，便从初始提示重新开始修复过程。最大对话长度为1表示反复从初始提示进行采样这一基本情形；随着最大对话长度增加，提供给模型的历史信息（先前的补丁和反馈）也会增多。

算法1详细说明了我们的对话修复过程。输入包括初始提示、原始测试失败信息、测试套件、ChatGPT模型、合理补丁生成提示（用于第3.3节），以及最大对话长度和最大尝试次数这两个超参数。最终输出是一组合理补丁，以及使用 ChatGPT API 访问服务的总成本。最大尝试次数是一个停止条件：当用于修复某个缺陷的尝试次数（即向 ChatGPT 发出的查询次数）达到上限时，修复过程便会停止（第3行）。最大对话长度则进一步限制了生成后续补丁时可使用的历史记录长度（第6行）。按照图2中的示例，我们首先将初始提示设为向 ChatGPT 提供的初始输入（第5行）。ChatGPT 首先生成了一个补丁，用于检查结果 是否为`NoObjectType`的实例。这个补丁可能是受到初始提示中原始测试失败信息的启发，其中测试断言行使用了一个类似的全局常量（`No_对象_类型`）。然而，这个补丁并不正确，因为它包含编译错误。ChatRepair 通过直接尝试编译并运行测试套件来识别这一问题，并构造反馈提示，指出生成的补丁存在编译错误（找不到符号(NoObjectType)）。为了生成第二个候选补丁，ChatRepair 将初始提示、第一个生成的补丁以及验证反馈（指出编译错误）拼接在一起，作为输入提供给 ChatGPT。

第二个补丁确实修复了编译错误，并检查 `result` 是否为 `UnionType` 的实例。ChatRepair 采用了算法 1 中所述的**动态** 反馈方法。ChatRepair 首先会编译并运行测试套件（第 8 行），以查看补丁能否成功通过测试用例（第 9 行）。在这个例子中，我们看到补丁仍未通过原始的缺陷暴露测试（第 12 行）。我们没有重复原始测试的错误消息，而是简单地回指初始提示，并说明它仍未解决原始测试失败（第 13 行）。由于我们始终会在输入中包含初始提示，因此可以向模型提供简洁的消息，说明测试失败的原因。另一方面，如果补丁通过了初始提示中使用的缺陷暴露测试，却在另一项测试（另一项原始失败测试或回归测试）中失败，我们就会像构造初始提示一样提供反馈，其中包括测试名称、相关测试代码和错误消息（第 15 行）。请注意，如果存在多项失败测试，为了使回复简洁，我们会像处理初始提示那样，只针对其中一项提供反馈。

第三个补丁的生成方式与之前类似：将初始提示与所有先前对话交换拼接起来。我们发现，在这种情况下，直接调用成员函数 `isNoType()` 的补丁能够成功通过测试套件。利用测试反馈信息（例如错误消息和相关测试代码），ChatGPT 识别出这个缺陷涉及与 None 对象或类型相关的边界情况，并生成了一个能够修复该缺陷的合理补丁。

3.3 合理补丁生成

在上一步之后，ChatRepair 应获得一个能够通过整个测试套件的合理补丁。然而，由于测试套件可能不完整，因而无法覆盖底层代码的所有可能预期用法，合理补丁并不总是能够正确修复底层缺陷 [52]。因此，开发人员必须手动检查合理补丁，以确定其中哪些是正确补丁。合理补丁和最终的正确补丁具有相似特征：它们都能够通过整个测试套件。因此，ChatRepair 不再从头开始（再次使用有缺陷的代码），而是直接利用现有的合理补丁来创建更多合理补丁。简而言之，为了提高生成正确补丁的概率，ChatRepair 会使用之前生成的合理补丁，并要求模型生成替代变体，从而产生额外的候选补丁。

图2展示了我们的合理补丁生成过程。首先，我们采用所使用的初始提示（第3.1节），其中包含原始的含缺陷的代码函数以及有用的测试失败信息。然后，我们将已生成的合理补丁列表附加到提示中（算法1第21行）。开始时，该列表只包含上一步得到的单个合理补丁；但是，随着我们继续生成更多合理补丁，列表也会不断增长。接下来，我们在提示中（算法1中的 Al- tInstruct）指出希望解决的任务——请生成一行替代修复行。 然后，我们将该提示作为输入提供给 ChatGPT，并获得一个候选补丁；我们会再次对其进行编译并运行测试套件，以检查它是否确实是另一个合理补丁（第24行）。我们会持续查询 ChatGPT，并更新提示以纳入新生成的合理补丁，从而避免反复生成相同的合理补丁，同时进一步基于较早的合理补丁进行构建（第25行）。再次设 C 为 ChatGPT模型，它输出生成一个序列的概率，I为初始提示，I_{pl}为任务指令，$PL_{<n} = \{pl_1, ..., pl_{n-1}\}$为之前生成的合理补丁。生成的下一个合理补丁可以形式化为条件概率：$C(pl_i|I, PL_{<i}, I_{pl})$

最终，我们得到一组合理补丁，可供开发人员手动检查。与以往仅在原始有缺陷的代码上操作以生成补丁的自动程序修复工具不同，ChatRepair 会利用每个合理补丁中的额外有用信息，生成更多合理补丁。合理补丁通常包含有用的要素或模式，这些要素或模式使其能够通过原始测试套件。因此，不必从头开始（即重新修复缺陷），而是在现有合理补丁的基础上，ChatGPT 凭借其强大的指令理解能力，可以生成更多合理补丁，从而提高最终补丁列表中包含能够修复缺陷的正确补丁的可能性。

4 实验设计

我们围绕以下研究问题评估 ChatRepair：

• **研究问题 1：** ChatRepair 的表现与自动程序修复领域最先进技术相比如何?
• **研究问题 2：** ChatRepair 在不同修复场景中的表现如何?
• **研究问题3：** ChatRepair 的不同组件对提升修复有效性有何贡献?

我们首先将 ChatRepair 与当前最先进的自动程序修复工具在常用的 Defects4j [24] 和 QuixBugs [32]修复数据集上的表现进行比较，以展示 ChatRepair 的性能。随后，我们详细考察每种修复场景（单行、单代码块和单函数），并与采用相同评估方式的基线工具进行比较，同时评估合理补丁生成步骤如何帮助提高正确修复的数量。最后，我们对 ChatRepair 的不同配置开展全面的消融研究。具体而言，我们不仅考察对话特性，还研究如何提供反馈，以及随着对话最大长度的变化，修复性能会受到怎样的影响。

4.1 实现

修复场景。 在 ChatRepair 中，我们研究了以往工作中使用的 3 种不同修复场景 [57]: **单行** —— 通过替换或添加单行代码进行修复；**单代码块**——通过替换或添加连续代码块进行修复；以及**单函数**——通过生成一个新函数来替换原有的缺陷版本。根据修复场景的不同，我们的初始提示略有差异，并在图2中给出了这三种场景的示例。请注意，以往基于学习的自动程序修复工具已对单代码块修复设置进行了广泛研究 [58, 61, 62]。

实现。 我们通过调用 ChatGPT API 端点 [6]，使用 Python 实现了 ChatRepair 的主要逻辑。我们使用 ChatGPT 系列中的 `gpt-3.5-turbo-0301` 模型；这是我们目前可用的最新模型。对于每个选定的提示，作者遵循最佳实践指南 [51]，并使用 ChatGPT 网页版 [8]，针对选定的若干缺陷手动考察了几种替代方案。我们将采样温度设为 1，以获得多样化的候选补丁。默认情况下，允许的最大修复尝试次数（包括初始修复和合理补丁生成步骤）为：单行和单代码块自动程序修复各 200 次，单函数场景为 100 次。我们使用 1 个少样本示例，并将最大对话长度设为 3。我们在一台 8 核工作站上评估所有生成的补丁，该工作站配备 Intel i7 10700KF Comet Lake CPU @3.80GHz 和 64GB RAM，运行 Ubuntu 20.04.3 LTS 和 OpenJDK Java 64 位服务器版 1.8。0_312遵循以往自动程序修复研究 [31, 58, 65], 的做法，我们将修复一个缺陷的默认端到端超时设为 5 小时。实际上，由于每个缺陷采样的补丁数量较少<500，我们的耗时远低于 5 小时。

4.2 被测系统

在评估中，我们使用了广泛研究的 Defects4j [24] 和 QuixBugs [32]程序修复基准。Defects4j 是一个 Java 基准，收集了来自开源项目的缺陷及其对应修复。与先前的自动程序修复工具 [17, 57, 58, 61], 类似，我们将 Defects4j 分为 1.2 和 2.0 两个版本。Defects4j 1.2 包含 6 个不同 Java 项目中的 391 个缺陷（移除 4 个已弃用缺陷后）。在这项工作中，我们遵循先前研究 [57]，将 Defects4j 1.2 中的缺陷分为单函数（255 个缺陷）、单代码块（154 个缺陷）和单行（80 个缺陷）三类。请注意，单代码块是单函数缺陷的子集，而单行缺陷是单代码块缺陷的子集。随后，我们将提出的 3 种修复场景分别应用于这 3 个数据集。请注意，在 RQ1 中，我们与先前工作 [57] 类似，报告结合全部三种修复场景后修复的缺陷总数，并在后续研究问题中分别考察每种修复场景。Defects4j 2.0 包含来自另外 9 个项目的 438 个新缺陷。为便于与先前自动程序修复工具进行比较 [58]，我们仅选取 Defects4j 2.0 中的 82 个单行缺陷，这是先前自动程序修复工具主要采用的设置。此外，我们还在 QuixBugs [32] 数据集上进行评估，该数据集由经典编程问题的 40 组有缺陷和已修复版本组成，涵盖 Python 和 Java。QuixBugs-Python 中的 40 个缺陷均为单函数、单代码块和单行缺陷；而 QuixBugs-Java 中分别有 40、37 和 36 个单函数、单代码块和单行缺陷。

4.3 对比技术

基线技术。我们将 ChatRepair 与最先进的传统自动程序修复、基于神经机器翻译的学习方法以及基于大语言模型的自动程序修复基线进行比较。我们选取了 8 种近期的基于学习和基于大语言模型的自动程序修复基线：SelfAPR [61], AlphaRepair [58],RewardRepair [62], Recoder [65], CURE [22], CoCoNuT [39], DLFix [31] 和 SequenceR [11]

具体而言，AlphaRepair 是一种最先进的基于大语言模型的修复工具，它通过预训练的 CodeBERT 模型 [10] 结合完形填空式自动程序修复来进行修复。此外，我们还纳入了一种近期研究中使用 Codex 模型 [14] 构建的基于大语言模型的自动程序修复工具（我们称之为 CodexRepair）；该研究中，研究人员直接将大语言模型用于自动程序修复，未进行任何微调 [57]。CodexRepair 也针对我们工作中使用的三种修复设置进行了研究，因此可以进行更直接的比较。对于传统自动程序修复工具，我们选取了 12 种具有代表性的技术进行比较：TBar [33], PraPR [17], AVATAR [34], SimFix [21],FixMiner [26], CapGen [55], JAID [9], SketchFix [20], NOPOL [12], jGenProg [40], jMutRepair [41], 和 jKali [41]。总共，我们与 21 种已有的自动程序修复工具进行了比较。此外，我们还评估了一个基线：直接使用 ChatGPT 模型进行采样来修复缺陷，不进行任何对话或提供任何反馈信息。我们将此基线称为 BaseChatGPT。由于我们的 3 种修复场景都依赖于已知缺陷位置，因此我们采用先前工具在完美故障定位条件下（即已知缺陷的真实位置）取得的结果。这是更合适的评估设置，因为它消除了运行故障定位工具所造成的性能差异 [56–58, 61, 62, 65]。按照自动程序修复研究中的惯例 [58, 65], ，我们直接报告先前研究中得到的修复结果 [17, 57, 61]

指标。 为了评估 ChatRepair，我们采用自动程序修复领域的标准指标：合理补丁 ——通过整个测试套件；以及正确补丁——与参考开发者补丁在语义或语法上等价。我们遵循自动程序修复领域的惯例，通过人工判断语义等价性来计算正确补丁的数量。此外，我们还使用尝试次数 这一指标，它表示查询 ChatGPT 时，为获得合理补丁或正确补丁所使用的样本数。尝试次数越少越好，因为这能缩短修复缺陷所需的时间。最后，我们还计算修复一个缺陷的美元成本 。本文撰写时，ChatGPT 每处理或生成 1000 个

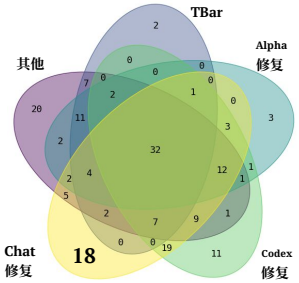


图 3：Defects4j 1.2 上的缺陷修复维恩图

测试名称: testCreateNumber()
失败行: 0xFADE == NumberUtils.createNumber("0Xfade").intValue()
错误消息: 0Xfade 不是有效数字。

```
    }
-   if (str.startsWith("0x") || str.startsWith("-0x")) {
+   if (str.startsWith("0x") || str.startsWith("0X") ||
+       str.startsWith("-0x") || str.startsWith("-0X")) {
        case '\n': sb.append("\\n"); break;
```

图 4：Defects4j 1.2 中修复的独有缺陷

词元的费用为 [6] 美元。与尝试次数不同，成本不仅取决于我们查询 ChatGPT 的次数，还会因每次查询的输入大小而异。

5 评估

5.1 RQ1：最先进方法比较

我们首先将 ChatRepair 与最先进的自动程序修复工具进行比较。表 1 展示了顶尖基线工具以及 BaseChatGPT 在 Defects4j 1.2 和 2.0 上修复的缺陷数量；BaseChatGPT 仅使用 ChatGPT，不提供任何测试失败信息和对话。首先可以观察到，与仅使用 ChatGPT模型的基线相比，ChatRepair 在 Defects4j 1.2 和 2.0 上分别多修复了 34 个和 23 个缺陷。这一改进得益于成功利用 ChatGPT模型的对话特性，通过结合先前生成的错误补丁或合理补丁以及测试失败信息来提供即时反馈。有趣的是，我们还观察到，诸如 AlphaRepair 这样的早期工具虽然使用规模小得多的大语言模型（CodeBERT），但在 Defects4j 2.0 上的单行修复等场景中，其表现优于 CodexRepair，这是因为它使用了修复专用模板，而不是纯代码填充。事实上，ChatRepair 首次证明了不使用任何修复模板的基于大语言模型的自动程序修复也能在 Defects4j 上达到顶尖性能。总体而言，*ChatRepair* 能够在 *Defects4j 1.2* 和 *2.0* 上分别实现 *114* 个和 *48* 个正确的缺陷修复，比当前最先进的自动程序修复工具分别多 *15* 个和 *17* 个。计算查询 ChatGPT 的总成本后，我们可以以每个缺陷 *0.42* 美元的成本修复 *337* 个缺陷中的 *162* 个！虽然这 114 个和 48 个修复结果是通过结合三种修复设置获得的，但与以往基于学习的工具相比，ChatRepair 生成的补丁仍然少得多（每个缺陷总共 <500 个）；以往工具每个缺陷最多可生成 10,000 个补丁 [22, 39]。同样，在表 2 中，ChatRepair 能够正确修复 QuixBugs-Java 和 QuixBugs-Python 数据集中的所有缺陷，击败所有表现最好的技术。

图 3 展示了所有受研究的基线方法和 ChatRepair 在 Defects4j 1.2 上修复缺陷的维恩图。我们选取修复缺陷数量最多的 3 种基线方法，并将所有其他受研究的自动程序修复工具（不仅限于表 1 中表现最好的工具）归为“其他”。可以看到，*ChatRepair* 能够为 *18* 个

测试名称: testNonFiniteDoublesWhenLenient()
失败行: jsonWriter.value(Double.NaN);
错误消息: 数值必须是有限值，但实际为 NaN

```
    writeDeferredName();
-   if (Double.isNaN(value) || Double.isInfinite(value)) {
+   if (!isLenient() && (Double.isNaN(value) ||
+       Double.isInfinite(value))) {
```

图 5：在 Defects4j 2.0 中修复的独特缺陷

在 *Defects4j 1.2* 上迄今为止所有先前方法都无法修复的独特缺陷。为展示 ChatRepair 的能力，图 4 展示了一个在 Defects4j 1.2 中仅由 ChatRepair 修复的缺陷示例（Lang-16）。修复方法是追加两个条件，使字符串以 “-0X” 或 “0X” 开头，分别对应一个数字的十六进制表示。这个缺陷很难修复，因为这些字符串在基于神经机器翻译的缺陷修复训练数据或基于大语言模型的预训练数据中都并不常见。为了生成这些条件，自动程序修复工具需要理解预期行为，以及其他可能的输入形式。事实上，其中一个条件（“0X”）直接出现在失败测试中，该测试尝试创建一个值为 “0Xfade” 的数字。ChatRepair 能够利用这条相关的测试代码信息，并生成测试中使用的字符串作为一个条件。此外，ChatRepair 也能轻松生成新的负数变体 “-0X”，因为 ChatGPT 能够从原始缺陷代码行中学习，其中也包含成对的负数和正数条件。将这两个条件结合起来，ChatRepair 得以获得修复该缺陷的正确补丁。

图 5 展示了 Defects4j 2.0 中另一个只有 ChatRepair 才能修复的缺陷（Gson-15）。修复需要满足另一个独特条件：`!isLenient()`。更棘手的是，这种函数用法在原始有缺陷的函数上下文中并未出现。因此，对于以往基于学习的自动程序修复工具而言，修复该缺陷可能极其困难，因为上下文中没有该条件的用法示例。然而，我们注意到，失败测试的名称是 `testNonFiniteDoublesWhenLenient`，其中直接出现了 lenient 一词。ChatGPT 通过查看失败测试名称，可以理解测试的语义信息；在本例中，测试旨在检验 宽松模式 = 开启 时的特定设置，并生成用于检查这一独特设置的正确修复代码行。这个例子进一步展现了 ChatRepair 如何利用以往被忽略的失败测试中的语义信息，直接引导修复过程。

5.2 研究问题 2：修复场景

接下来，我们将更详细地考察三种修复设置（单行、单代码块和单函数）。本节中，我们重点分析 BaseChatGPT（使用 ChatGPT，不提供任何测试失败信息或对话）以及 CodexRepair；后者是表现最佳的基于大语言模型的自动程序修复方法，也已在我们采用的三种修复设置下进行评估。

表 3 展示了 ChatRepair 在 Defects4j 1.2 和两个 QuixBugs 数据集上与两种基线的对比结果。有趣的是，我们首先发现，在 Defects4j 1.2 这一真实世界基准上，基础版 ChatGPT 模型的表现甚至略逊于 CodexRepair。我们推测，这是因为 ChatGPT 并非专为代码生成而设计，也没有像 Codex 那样针对代码生成进行直接微调。因此，像以往基于大语言模型的自动程序修复工具那样，仅从相同的初始提示中采样而不提供额外

表 1：Defects4j上的正确修复

数据集 ChatRepair		BaseChatGPT	CodexRepair	AlphaRepair SelfAPR		RewardRepair Recoder TBar CURE CoCoNuT				
图表	15	9	9	9	7	5	10	11	10	7
闭包	37	23	30	23	19	15	21	16	14	9
Lang	21	15	22	13	10	7	11	13	9	7
Math	32	25	29	21	22	19	18	22	19	16
Mockito	6	6	6	5	3	3	2	3	4	4
Time	3	2	3	3	3	1	3	3	1	1
D4J 1.2	**114**	80	99	74	64	50	65	68	57	44
D4J 2.0	**48**	25	31	36	31	25	11	8	-	-

表 2：QuixBugs 的正确修复

QuixBugs	图表修复	Base ChatGPT	Codex 修复	Alpha 修复	CoCoNuT
Python	**40**	40	40	27	19
Java	**40**	40	38	28	13

表 3：使用三种修复设置得到的正确修复

工具	D4J 1.2			Quixbugs-Py			Quixbugs-J		
	SL	SH	SF	SL	SH	SF	SL	SH	SF
ChatRepair	**57**	**79**	**76**	**39**	**40**	**40**	**36**	**37**	**39**
BaseChatGPT	41	55	45	38	37	35	33	36	39
CodexRepair	39	62	63	39	39	37	34	34	32

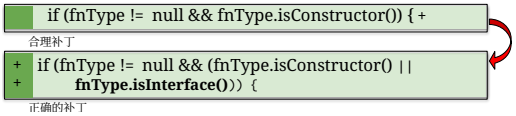


图 6：合理补丁生成示例

信息，直接使用 ChatGPT 并不能带来显著提升 [63]。另一方面，ChatRepair 将 ChatGPT 强大的对话与指令理解能力和动态反馈相结合，能够更好地利用先前被忽略的测试失败信息和早期补丁尝试，从而更有效地进行自动程序修复。相反，Codex 主要针对代码补全而设计，缺乏以对话方式使用的能力。总而言之，在每种单独的修复设置中，与最先进的 CodexRepair 和直接运行基础版 ChatGPT 相比，ChatRepair 修复的缺陷数量均为最多。

此外，正确修复数量的增加不仅源于对话和验证反馈方面，也得益于我们的合理补丁生成步骤。请回想一下，一旦生成了一个合理补丁，我们就会直接利用该补丁，通过请求 ChatGPT 提供其他变体来生成更多合理补丁。总之，在 Defects4j 1.2 上，合理补丁生成平均可额外增加 9.4、16.6 和 5.5 个合理补丁，并分别使单行、单代码块和单函数修复场景中正确修复的缺陷数量增加 4、7 和 2 个。这一提升表明，我们提出的方法能够有效利用合理补丁中的重要信息来生成更多补丁，从而实现正确修复。

图 6 展示了 Cha- tRepair 对 `Closure-125` 给出的一个正确修复示例。该修复起初仅是合理补丁，随后通过引导 ChatGPT 从早先的合理补丁中学习，变得正确。我们看到，ChatRepair 生成的初始合理补丁确实能够通过开发者测试，因为它会检查 `fnType` 是否为构造函数。然而，测试套件并未覆盖所有边界情况，实际的正确修复还需要检查接口的一个附加条件。通过合理补丁生成，ChatRe- pair 无需从头开始（仅使用有缺陷的代码），而是可以利用第一个合理补丁中已获得的知识。在这个缺陷修复中，ChatRepair 通过从原始合理补丁中学习，添加了正确修复所需的附加条件。

5.3 研究问题3：ChatRepair 的配置

我们研究了 ChatRepair 的不同配置。具体而言，我们考察了以下重要参数：（1）所用的初始提示；（2）提供的反馈响应；（3）最大对话长度。由于在消融研究的每个维度上多次调用 ChatGPT API 的成本较高，我们聚焦于 Defects4j 1.2 中的 80 个单行错误。此外，鉴于检查补丁需要投入大量人工，我们在本研究问题中分析的是生成的合理修复数量，而非正确修复数量。除默认采用零样本（即不提供任何先前的错误修复示例）外，我们的每项消融实验均使用第4.1节所述的默认设置；采用零样本是因为它最能体现各个组成部分的单独作用，也便于研究少样本示例的影响。

表4：初始提示的不同变体

初始提示	#P	平均尝试次数	平均费用（美元）
基础提示	55	22.53	$0.069
测试名称+错误消息	59	22.47	$0.072
测试名称+错误消息+失败行	**64**	**21.86**	**$0.061**
测试名称+错误消息+测试正文	61	23.42	$0.083
你是一个有帮助的助手	**64**	24.17	$0.074
你是一个自动程序修复工具	**64**	**21.86**	**$0.061**
零样本	64	21.86	**$0.061**
单样本	**65**	9.91	$0.072
双样本	**65**	**9.87**	$0.085

5.3.1 初始提示。 除了提供给 ChatGPT 的默认初始提示外，我们还评估了几种替代变体。每种变体都试图说明一些信息的关键方面，这些信息在修复过程中可能对 ChatGPT 有所帮助。表4展示了不同初始提示的结果。行 **基础提示**指的是仅指出代码包含缺陷并要求模型提供修复方案的提示，**测试名称+错误消息** 同时包含失败测试名称（例如 `testGetCategoryIndex`）和测试失败错误信息（例如 空指针异常），**测试-名称+错误消息+失败行** 还额外包含失败在测试中发生的确切行，

失败发生在测试中的位置（例如，断言相等(-1, empty.getCategoryI以及 **测试名称+错误消息+测试主体** 此外还使用整个失败测试函数体，而不仅仅是失败代码行。

首先，我们观察到，仅提供有缺陷的代码并要求模型生成补丁的基线初始提示，在修复缺陷数量方面表现最差。我们发现，添加失败测试名称和错误消息等辅助信息，可以进一步提升修复性能。命名清晰的测试能够传达测试的语义。错误消息也能为测试失败的性质提供独特线索（例如，空指针异常、数组越界检查），并可直接启发一种可能正确的补丁。此外，我们还发现，提供测试中发生失败的确切行，也能提升修复性能。这类代码行可能包含展示预期结果的断言（例如，数值比较），也可能包含触发异常或崩溃的语句（例如，字段解引用）。这还能为 ChatGPT 提供具体线索，帮助其修复特定缺陷。此外，我们观察到，包含整个测试函数代码的提示在修复缺陷数量方面也表现良好。然而，与其他使用的初始提示相比，它的平均成本最高。这是因为模型在每次修复尝试中都需要额外处理整个测试函数代码，成本很大程度上取决于测试函数的长度，而且其中可能包含与缺陷无关的测试代码。因此，只包含失败测试代码行的简洁提示，就能实现有效的修复性能，同时更加经济。

表4还展示了初始输入的其他几个参数，而不仅仅是其中包含的测试失败信息。行 `You are ahelpful assistant` 使用 ChatGPT 的默认系统消息（第3节），而 `You are an APR tool`（此处使用 APR 的全称）则是我们修改后的系统消息。虽然这两种系统消息对应的修复数量相近，但我们发现，将系统消息与我们要解决的任务——程序修复——相匹配后，模型能够更快（尝试次数更少）地生成合理补丁，因为它能更快理解自己要解决的任务。因此，我们可以通过设计特定的系统消息来降低 ChatRepair 的成本。此外，我们还在表4中评估了在目标有缺陷的代码输入之前提供错误修复少样本示例的效果。我们发现，向 ChatGPT 提供一些以往错误修复的示例后，合理补丁的数量略有增加，同时修复缺陷所需的尝试次数大幅减少。与系统消息类似，少样本示例可以帮助模型理解任务和输入/输出格式，从而熟悉错误修复。

表5：反馈响应变体

反馈响应	#P	平均尝试次数	平均费用
基础反馈	58	23.12	$0.071
测试名称+错误消息	61	22.48	$0.073
测试名称+错误消息+失败行	62	24.71	$0.074
动态	**64**	**21.86**	**$0.061**

5.3.2 反馈响应。 ChatRepair 设计的另一个重要方面是我们向模型提供的反馈响应。与初始提示设计类似，我们也考虑了多种向模型提供反馈的不同方式。表5展示了

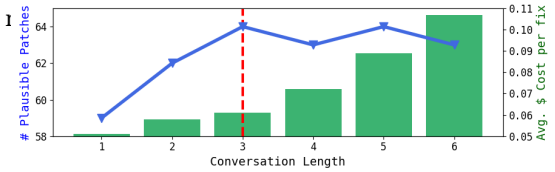


图7：最大对话长度的影响

不同反馈响应变体的结果。行**基线-反馈** 表示我们只告诉模型生成的补丁不正确，不提供任何额外反馈。与初始提示构造类似，**测试名称+错误消息**包含失败测试名称和测试失败错误消息，**测试名称+错误消息+失败行**还包含测试中发生失败的确切行。与初始提示构造不同，**动态** 是我们的默认方法：只有当新生成的补丁出现了不同于原始补丁的失败时，我们才提供测试名称、错误消息和行号（第3.2节）。这样，我们就能更具体地告知 ChatGPT 是否在修复缺陷方面取得了进展（例如，补丁不再因空指针异常而崩溃，但在另一个测试中失败）。

首先，我们看到，基线响应消息在修复缺陷数量方面的结果最差。与初始提示的表现类似，通过添加失败测试名称、错误消息以及失败测试中的确切失败行，可以提升性能。此外，采用动态反馈响应还能进一步提升性能。由于我们已在初始提示中向 ChatGPT 提供了失败测试名称、错误消息和失败行，因此在动态反馈响应中，只有当生成的补丁包含不同的失败信息时，我们才会提供新数据。这样，我们便能通过引用先前的消息，更充分地利用对话特性。此外，由于补丁未取得进一步进展时，我们只需生成一条简短精炼的消息，因此还能降低成本。

5.3.3 对话长度。 图7展示了不同最大对话长度下，合理修复的数量以及修复一个缺陷的平均美元成本。回顾第3.2节，最大对话长度决定了每次修复对话中包含的历史记录/反馈量，其中长度 = 1相当于使用初始提示进行采样，且不提供任何反馈。我们观察到，直接从 ChatGPT 采样而不进行任何对话时，得到的合理修复数量最少。随着我们在 ChatRepair 中加入对话/反馈机制，合理补丁的数量也有所增加。与反复使用同一提示进行采样相比，通过以对话方式使用 ChatRepair，模型可以从之前的错误中学习，并结合简洁的测试失败反馈信息，生成更多合理补丁。我们还注意到，即使将对话长度增加到较高水平（即 5 和 6），模型仍能保持其性能。然而，与较短的对话长度（3）相比，较长的对话长度会大幅增加修复缺陷的成本。原因是，随着长度增加，模型处理的历史记录/上下文（词元）也会增多，从而提高每个已修复缺陷的成本。默认对话长度 3 在成本与修复缺陷数量之间实现了良好的平衡。

6 有效性威胁

内部。 首个内部威胁来自用于判定合理补丁是否正确的人工验证，即将其与参考开发者补丁进行比较。为应对这一问题，我们按照以往研究的做法 [22, 57, 58, 61, 62, 65], 仔细检查并讨论了每个补丁。

另一个有效性威胁来自参考开发者补丁的数据泄漏问题：这些补丁可能属于 ChatGPT 的原始训练数据。由于 ChatGPT 是专有模型，只能通过 API 访问，我们无法获知其所用训练数据的确切内容。为了解决这一问题，我们沿用先前工作 [58] 的方法，首先统计 ChatRepair 在 Defects4j 1.2 上生成的正确补丁中，有多少与参考开发者补丁相同。我们发现，在 212 个正确补丁（汇总三个修复场景中所有正确补丁）中，有 77 个与开发者参考修复相同（36%）。此外，即使移除所有与参考开发者补丁相同的正确补丁（77 个），ChatRepair 仍能为 12 个以往方法均无法修复的不同缺陷生成正确补丁。进一步地，与使用相同底层模型的 ChatGPT 基础修复基线相比，ChatRepair 显著提升了性能（多生成 34 个正确修复），这表明 ChatRepair 获得的结果并非仅仅源于对训练数据的记忆。要彻底消除此项威胁，我们需要从头开始重新训练 ChatGPT，而这对学术项目而言并不可行。

外部有效性。 我们的评估数据集带来的主要外部有效性威胁是，ChatRepair 获得的改进未必能推广到其他修复数据集。为此，我们不仅在广受欢迎的 Defects4j 1.2 数据集上进行评估，还在 Defects4j 2.0 和两个 QuixBugs 数据集上进行评估，以证明其泛化能力。

7 结论

我们提出了 ChatRepair，这是首个由对话驱动的全自动自动程序修复工具，它利用新开发的 ChatGPT模型执行修复。ChatRepair 从先前不正确和合理的补丁中学习，并利用测试失败信息向模型提供即时、动态的反馈，以生成新补丁。借助对话式修复范式，ChatRepair 在 Defects4j 1.2 和 2.0 上分别修复了 114 个和 48 个缺陷（比表现最佳的基线分别多 15 个和 17 个），达到最先进的性能。

参考文献

[1] Jacob Austin、Augustus Odena、Maxwell Nye、Maarten Bosma、Henryk Michalewski、David Dohan、Ellen Jiang、Carrie Cai、Michael Terry、Quoc Le 和 Charles Sutton。2021。大型语言模型的程序合成。arXiv:2108.07732。[2] Yejin Bang、Samuel Cahyawijaya、Nayeon Lee、Wenliang Dai、Dan Su、Bryan Wilie、Holy Lovenia、Ziwei Ji、Tiezheng Yu、Willy Chung 等。2023。对 ChatGPT 在推理、幻觉和交互性方面进行多任务、多语言、多模态评估。*arXiv* 预印本 *arXiv:2302.04023* (2023)。[3] Samuel Benton 、 Xia Li 、 Yiling Lou 和 Lingming Zhang。2020。统一调试的有效性：对 16 个程序修复系统的广泛研究。发表于 *ASE*。907–918。[4] Sid Black、Stella Biderman、Eric Hallahan、Quentin Anthony、Leo Gao、Laurence Golding、Horace He、Connor Leahy、Kyle McDonell、Jason Phang、Michael Pieler、USVSN Sai Prashanth、Shivanshu Purohit、Laria Reynolds、Jonathan Tow、Ben Wang 和 Samuel Weinbach。2022。GPT-NeoX-20B：一种开源自回归语言模型。发表于*ACL* 大型语言模型创建的挑战与展望研讨会论文集。arXiv:2204.06745。[5] Tom Brown、Benjamin Mann、Nick Ryder、Melanie Subbiah、Jared D Kaplan、Prafulla Dhariwal、Arvind Neelakantan、Pranav Shyam、Girish Sastry、Amanda Askell 等。2020。语言模型是小样本学习器。《神经信息处理系统进展》33（2020），1877–1901。[6] chatgptendpoint 2023。推出 ChatGPT 和 Whisper API。https://openai.com/blog/introducing-chatgpt-and-whisper-apis。[7] chatgptguide 2023。ChatGPT 指南。https://platform.openai.com/docs/guides/ chat。[8] chatgptweb 2023。ChatGPT 网页版。https://chat.openai.com/chat。[9] Liushan Chen、Yu Pei 和 Carlo A. Furia。2017。无需契约的基于契约的程序修复。载于《2017年第32届IEEE/ACM国际自动化软件工程会议（ASE）》。637–647。[10] Mark Chen、Jerry Tworek、Heewoo Jun、Qiming Yuan、Henrique Ponde de Oliveira Pinto、Jared Kaplan、Harri Edwards、Yuri Burda、Nicholas Joseph、Greg Brockman 等。2021。评估基于代码训练的大语言模型。arXiv预印本 arXiv:2107.03374（2021）。[11] Zimin Chen、Steve Kommrusch、Michele Tufano、Louis-Noël Pouchet、Denys Poshyvanyk 和 Martin Monperrus。2019。SequenceR：用于端到端程序修复的序列到序列学习。《IEEE软件工程汇刊》（2019）。[12] Favio DeMarco、Jifeng Xuan、Daniel Le Berre 和 Martin Monperrus。2014。使用 SMT 自动修复有缺陷的 If 条件和缺失的前置条件。载于《第六届软件测试、验证与分析中的约束国际研讨会论文集》（印度海得拉巴）（CSTVA 2014）。30–39。[13] Thomas Durieux 和 Martin Monperrus。2016。Dynamoth：用于自动程序修复的动态代码合成。载于《第11届软件测试自动化国际研讨会论文集》。85–91。[14] Zhangyin Feng、Daya Guo、Duyu Tang、Nan Duan、Xiaocheng Feng、Ming Gong、Linjun Shou、Bing Qin、Ting Liu、Daxin Jiang 和 Ming Zhou。2020。CodeBERT：用于编程语言和自然语言的预训练模型。arXiv:2002.08155。[15] Daniel Fried、Armen Aghajanyan、Jessy Lin、Sida Wang、Eric Wallace、Freda Shi、Ruiqi Zhong、Wen-tau Yih、Luke Zettlemoyer 和 Mike Lewis。2022。InCoder：用于代码填充与合成的生成模型。arXiv:2204.05999。[16] Luca Gazzola、Daniela Micucci 和 Leonardo Mariani。2019。自动化软件修复：综述。《IEEE软件工程汇刊》45，1（2019），34–67。[17] Ali Ghanbari、Samuel Benton 和 Lingming Zhang。2019。通过字节码变异实现实用程序修复。载于《第28届ACM SIGSOFT国际软件测试与分析研讨会论文集》（中国北京）（ISSTA 2019）。ACM，19–30。[18] Claire Le Goues、Michael Pradel 和 Abhik Roychoudhury。2019。自动程序修复。《ACM通讯》62，12（2019），56–65。[19] Abram Hindle、Earl T. Barr、Zhendong Su、Mark Gabel 和 Premkumar Devanbu。2012。论软件的自然性。载于《第34届国际软件工程会议论文集》（ICSE '12）。837–847。[20] Jinru Hua、Mengshi Zhang、Kaiyuan Wang 和 Sarfraz Khurshid。2018。SketchFix：一种采用惰性候选生成的自动程序修复工具。载于《2018年第26届ACM欧洲软件工程会议与软件工程基础研讨会联合会议论文集》（美国佛罗里达州布埃纳维斯塔湖）（ESEC/FSE 2018）。ACM，888–891。[21] Jiajun Jiang、Yingfei Xiong、Hongyu Zhang、Qing Gao 和 Xiangqun Chen。2018。利用现有补丁和相似代码塑造程序修复空间。载于《第27届ACM SIGSOFT国际软件测试与分析研讨会论文集，ISSTA 2018，荷兰阿姆斯特丹，2018年7月16日至21日》，Frank Tip 和 Eric Bodden（编）。ACM，298–309。[22] Nan Jiang、Thibaud Lutellier 和 Lin Tan。2021。CURE：用于自动程序修复的代码感知神经机器翻译。2021年IEEE/ACM第43届国际软件工程会议（ICSE）（2021年5月）。[23] Yanjie Jiang、Hui Liu、Nan Niu、Lu Zhang 和 Yamin Hu。2021。从版本控制系统中的人工编写补丁提取精简的缺陷修复补丁。载于《2021年IEEE/ACM第43届国际软件工程会议（ICSE）》。IEEE，686–698。[24] René Just、Darioush Jalali 和 Michael D. Ernst。2014。Defects4J：用于对 Java 程序开展受控测试研究的现有故障数据库（ISSTA 2014）。计算机协会，美国纽约州纽约，437–440。[25] Sophia D Kolak、Ruben Martins、Claire Le Goues 和 Vincent Josua Hellendoorn。2022。使用语言模型生成补丁：可行性与扩展行为。载于《代码深度学习研讨会》。[26] Anil Koyuncu、Kui Liu、Tegawendé F. Bissyandé、Dongsun Kim、Jacques Klein、Martin Monperrus 和 Yves Le Traon。2020。FixMiner：挖掘与自动程序修复相关的修复模式。《实证软件工程》25，3（2020），1980–2024。[27] Xuan-Bach D Le、Duc-Hiep Chu、David Lo、Claire Le Goues 和 Willem Visser。2017。S3：通过示例编程实现语法和语义引导的修复合成。载于《2017年第11届软件工程基础联合会议论文集》。593–604。[28] Xuan Bach D. Le、David Lo 和 Claire Le Goues。2016。历史驱动的程序修复。载于《2016年IEEE第23届软件分析、演化与再工程国际会议（SANER）》，第1卷。213–224。

[29] Claire Le Goues、Michael Dewey-Vogt、Stephanie Forrest 和 Westley Weimer。2012。自动程序修复的系统研究：每个缺陷花费8美元，修复105个缺陷中的55个。载于《2012年第34届国际软件工程会议（ICSE）》。IEEE，3–13。[30] Claire Le Goues、ThanhVu Nguyen、Stephanie Forrest 和 Westley Weimer。2012。GenProg：一种通用的自动软件修复方法。《IEEE软件工程汇刊》38，1（2012），54–72。[31] Yi Li、Shaohua Wang 和 Tien N. Nguyen。2020。DLFix：用于自动程序修复的基于上下文的代码转换学习。载于《ACM/IEEE第42届国际软件工程会议论文集》（韩国首尔）（ICSE '20）。计算机协会，美国纽约州纽约，602–614。[32] Derrick Lin、James Koppel、Angela Chen 和 Armando Solar-Lezama。2017。QuixBugs：基于 Quixey挑战赛的多语言程序修复基准集（SPLASH Companion 2017）。计算机协会，美国纽约州纽约，55–56。[33] Kui Liu、Anil Koyuncu、Dongsun Kim 和 Tegawendé F. Bissyandé。2019。TBar：重新审视基于模板的自动程序修复。载于《第28届ACM SIGSOFT国际软件测试与分析研讨会论文集》（ISSTA 2019）。ACM，美国纽约州纽约，31–42。[34] Kui Liu、Anil Koyuncu、Dongsun Kim 和 Tegawendé F. Bissyandé。2019。AVATAR：利用静态分析违规的修复模式修复语义错误。载于《第26届IEEE软件分析、演化与再工程国际会议论文集》。IEEE，456–467。[35] Pengfei Liu、Weizhe Yuan、Jinlan Fu、Zhengbao Jiang、Hiroaki Hayashi 和 Graham Neubig。2023。预训练、提示与预测：自然语言处理提示方法的系统综述。《计算调查》55，9（2023），1–35。[36] Fan Long 和 Martin Rinard。2015。通过条件合成进行分阶段程序修复。载于《2015年第10届软件工程基础联合会议论文集》（意大利贝加莫）（ESEC/FSE 2015）。美国纽约州纽约，166–178。[37] Fan Long 和 Martin Rinard。2016。对生成与验证型补丁生成系统搜索空间的分析。载于《第38届国际软件工程会议论文集》。702–713。[38] Yiling Lou、Ali Ghanbari、Xia Li、Lingming Zhang、Haotian Zhang、Dan Hao 和 Lu Zhang。2020。自动程序修复能否改进故障定位？一种统一调试方法。载于《第29届ACM SIGSOFT国际软件测试与分析研讨会论文集，ISSTA 2020，美国加利福尼亚州洛杉矶，2020年7月18日至22日》。12页。[39] Thibaud Lutellier、Hung Viet Pham、Lawrence Pang、Yitong Li、Moshi Wei 和 Lin Tan。2020。CoCoNuT：通过集成组合上下文感知神经翻译模型实现程序修复。载于《第29届ACM SIGSOFT国际软件测试与分析研讨会论文集》（线上活动，美国）（ISSTA 2020）。计算机协会，美国纽约州纽约，101–114。[40] Matias Martinez、Thomas Durieux、Jifeng Xuan、Romain Sommerard 和 Martin Monperrus。2015。真实缺陷的自动修复：Defects4J数据集的经验报告。arXiv:1505.07002。[41] Matias Martinez 和 Martin Monperrus。2016。ASTOR：一个用于 Java 的程序修复库（演示）。载于《第25届国际软件测试与分析研讨会论文集》（德国萨尔布吕肯）（ISSTA 2016）。计算机协会，美国纽约州纽约，441–444。[42] Sergey Mechtaev、Jooyong Yi 和 Abhik Roychoudhury。2016。Angelix：通过符号分析实现可扩展的多行程序补丁合成。载于《第38届国际软件工程会议论文集》（得克萨斯州奥斯汀）（ICSE '16）。691–701。[43] Hoang Duong Thien Nguyen、Dawei Qi、Abhik Roychoudhury 和 Satish Chandra。2013。Semfix：通过语义分析实现程序修复。载于《2013年第35届国际软件工程会议（ICSE）》。IEEE，772–781。[44] Erik Nijkamp、Bo Pang、Hiroaki Hayashi、Lifu Tu、Huan Wang、Yingbo Zhou、Silvio Savarese 和 Cai ming Xiong。2022。Codegen：一个支持多轮程序合成的开放式代码大语言模型。arXiv预印本 arXiv:2203.13474（2022）。[45] Long Ouyang、Jeffrey Wu、Xu Jiang、Diogo Almeida、Carroll Wainwright、Pamela Mishkin、Chong Zhang、Sandhini Agarwal、Katarina Slama、Alex Ray 等。2022。训练语言模型遵循指令并利用人类反馈。《神经信息处理系统进展》35（2022），27730–27744。[46] Julian Aron Prenner、Hlib Babii 和 Romain Robbes。2022。OpenAI 的 Codex 能修复缺陷吗？——在 QuixBugs 上的评估。载于《2022年IEEE/ACM自动程序修复国际研讨会（APR）》。69–75。[47] Alec Radford、Karthik Narasimhan、Tim Salimans、Ilya Sutskever 等。2018。通过生成式预训练提升语言理解能力。（2018）。[48] Baishakhi Ray、Vincent Hellendoorn、Saheel Godhane、Zhaopeng Tu、Alberto Bacchelli 和 Premkumar Devanbu。2016。论有缺陷代码的"自然性"。载于《第38届国际软件工程会议论文集》（得克萨斯州奥斯汀）（ICSE '16）。计算机协会，美国纽约州纽约，428–439。[49] John Schulman、Filip Wolski、Prafulla Dhariwal、Alec Radford 和 Oleg Klimov。2017。近端策略优化算法。arXiv:1707.06347。[50] John Schulman、Barret Zoph、Jacob Hilton、Christina Kim、Jacob Menick、Ji- ayi Weng、Juan Felipe Ceron Uribe、Liam Fedus、Luke Metz、Michael Pokorny、Rapha Gontijo Lopes、Shengjia Zhao、Arun Vijayvergiya、Eric Sigler、Adam Perel- man、Chelsea Voss、Mike Heaton、Joel Parish、Dave Cummings、Rajeev Nayak、Valerie Balcom、David Schnurr、Tomer Kaftan、Chris Hallacy、Nicholas Turley、Noah Deutsch、Vik Goel、Jonathan Ward、Aris Konstantinidis、Wojciech Zaremba、Long Ouyang、Leonard Bogdonoff、Joshua Gross、David Medina、Sarah Yoo、Teddy Lee、Ryan Lowe、Dan Mossing、Joost Huizinga、Roger Jiang、Carroll Wainwright、Diogo Almeida、Steph Lin、Marvin Zhang、Kai Xiao、Katarina Slama、Steven Bills、Alex Gray、Jan Leike、Jakub Pachocki、Phil Tillet、Shantanu Jain、Greg Brockman 和 Nick Ryder。2022。ChatGPT：优化用于对话的语言模型。（2022）。https://openai.com/blog/chatgpt/。[51] Jessica Shieh。2023。使用 OpenAI API 进行提示工程的最佳实践。https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-openai-api。[52] Edward K Smith、Earl T Barr、Claire Le Goues 和 Yuriy Brun。2015。治愈是否比疾病更糟？自动化程序修复中的过拟合。载于*2015*年第*10*届软件工程基础联合会议论文集。532–543。[53] Ilya Sutskever、Oriol Vinyals 和 Quoc V. Le。2014。神经网络的序列到序列学习。arXiv:1409.3215。[54] Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N. Gomez、Lukasz Kaiser 和 Illia Polosukhin。2017。Attention Is All You Need。（2017）。arXiv:1706.03762。[55] Ming Wen、Junjie Chen、Rongxin Wu、Dan Hao 和 Shing-Chi Cheung。2018。用于改进自动化程序修复的上下文感知补丁生成。载于第*40*届国际软件工程会议论文集（瑞典哥德堡）（*ICSE '18*）。1–11。[56] W. Eric Wong、Ruizhi Gao、Yihao Li、Rui Abreu 和 Franz Wotawa。2016。软件故障定位综述。*IEEE*软件工程汇刊42，8（2016），707–740。[57] Chunqiu Steven Xia、Yuxiang Wei 和 Lingming Zhang。2023。大型预训练语言模型时代的自动化程序修复。载于第*45*届*ACM/IEEE*国际软件工程会议（*ICSE '23*）论文集。[58] Chunqiu Steven Xia 和 Lingming Zhang。2022。少些训练，多些修复：通过零样本学习重新审视自动化程序修复。载于第*30*届*ACM*欧洲软件工程联合会议和软件工程基础研讨会论文集（*ESEC/FSE 2022*）。[59] Chunqiu Steven Xia 和 Lingming Zhang。2023。对话式自动程序修复。*arXiv* 预印本 *arXiv:2301.13246*（2023）。[60] Frank F. Xu、Uri Alon、Graham Neubig 和 Vincent Josua Hellendoorn。2022。对代码大型语言模型的系统评估。载于第*6*届*ACM SIGPLAN*机器编程国际研讨会论文集（美国加利福尼亚州圣迭戈）（*MAPS 2022*）。计算机协会，美国纽约州纽约，1–10。[61] He Ye、Matias Martinez、Xiapu Luo、Tao Zhang 和 Martin Monperrus。2022。SelfAPR：结合测试执行诊断的自监督程序修复。载于第*37*届*IEEE/ACM*国际自动化软件工程会议（*ASE22*）。计算机协会，文章92，13页。[62] He Ye、Matias Martinez 和 Martin Monperrus。2022。基于执行的反向传播神经程序修复。载于*2022*年*IEEE/ACM*第*44*届国际软件工程会议（*ICSE*）。1506–1518。[63] Junjie Ye、Xuanting Chen、Nuo Xu、Can Zu、Zekai Shao、Shichun Liu、Yuhan Cui、Zeyang Zhou、Chao Gong、Yang Shen 等。2023。GPT-3 和 GPT-3.5 系列模型的综合能力分析。*arXiv* 预印本 *arXiv:2303.10420*（2023）。[64] Shafiq Joty Yue Wang、Weishi Wang 和 Steven C.H. Hoi。2021。CodeT5：面向代码理解与生成的标识符感知统一预训练编码器-解码器模型。载于*2021*年自然语言处理经验方法会议论文集（*EMNLP 2021*）。[65] Qihao Zhu、Zeyu Sun、Yuan-an Xiao、Wenjie Zhang、Kang Yuan、Yingfei Xiong 和 Lu Zhang。2021。用于神经程序修复的语法引导编辑解码器。载于第*29*届*ACM*欧洲软件工程会议和软件工程基础研讨会联合会议论文集。ACM，美国纽约州纽约，341–353。[66] Daniel M. Ziegler、Nisan Stiennon、Jeffrey Wu、Tom B. Brown、Alec Radford、Dario Amodei、Paul Christiano 和 Geoffrey Irving。2019。基于人类偏好的语言模型微调。arXiv:1909.08593。