基于代码上下文与大语言模型的自动化合并冲突解决方法

董政宇[1)] 张卫丰[1)] 许 蕾[2)] 何 华[3)] 梅光辉[4)] 玄跻峰[3)]

[1)](南京邮电大学计算机学院 南京 210023)

[2)](南京大学计算机学院 南京 210023)

[3)](武汉大学计算机学院 武汉 430072)

[4)](OPPO广东移动通信有限公司 广东 东莞 523846)

摘 要 现代协作软件开发中,版本控制系统如Git被广泛使用,但分支合并产生的代码冲突成为了一个常见难题。约12%的提交涉及合并冲突,在大型项目中这一比例高达50%。传统的人工解决方法耗时费力,迫切需要自动化的解决方案。本文提出了一种基于代码上下文的方法,利用ChatGPT自动解决代码合并冲突,并验证了不同类型的代码上下文对冲突消解的效果。研究结果表明,较低温度设置和标准思维链提示风格能够显著发挥代码上下文的性能。特别是基于静态分析的程序依赖上下文,其正确率提高了42%,显示出强大的冲突消解辅助能力。此外,实验还展示了不同类型代码上下文的有效性分析,证明了代码上下文在不同大语言模型上可以有效帮助消解冲突的通用性。本研究为自动解决代码合并冲突提供了新的思路,并强调了代码上下文的重要性,特别是在提高自动化工具准确性和稳定性方面的潜力。

关键词 分支合并;合并冲突自动消解;代码上下文;提示工程;ChatGPT

中图法分类号 TP391 **DOI号** 10.11897/SP.J.1016.2025.02468

Automated Merge Conflict Resolution Based on Code Context and Large Language Models

DONG Zheng-Yu[1)] ZHANG Wei-Feng[1)] XU Lei[2)] HE Hua[3)]

MEI Guang-Hui[4)] XUAN Ji-Feng[3)]

[1)](School of Computer Science, Nanjing University of Posts & Telecommunication, Nanjing 210023)

[2)](School of Computer Science, Nanjing University, Nanjing 210023)

[3)](School of Computer Science, Wuhan University, Wuhan 430072)

[4)](OPPO Guangdong Mobile Communications Co., Ltd., Dongguan, Guangdong 523846)

Abstract Version control systems have become indispensable in modern software development, enabling developers to track and manage changes to source code over time. These systems not only allow multiple developers to work concurrently on the same codebase but also provide a structured way to integrate their contributions through branching and merging mechanisms. However, one of the most persistent challenges associated with these systems is the occurrence of merge conflicts—situations where two or more developers modify the same part of the code in

收稿日期:2025-01-22;在线发布日期:2025-07-10。本课题得到国家自然科学基金面上项目(No. 62272214)、南京市国际合作项目(No. 202401006)资助。**董政宇**,硕士,中国计算机学会(CCF)会员,主要研究领域为人工智能在软件工程领域中的应用。E-mail:dzyown@foxmail.com。**张卫丰**(通信作者),博士,教授,中国计算机学会(CCF)会员,主要研究领域为代码管理、持续集成。E-mail:zhangwf@njupt.edu.cn。**许 蕾**(通信作者),博士,副教授,中国计算机学会(CCF)会员,主要研究领域为软件分析测试、智能化软件工程。E-mail:xlei@nju.edu.cn。**何 华**,硕士,主要研究领域为软件工程。**梅光辉**,硕士,主要研究领域为软件工程。**玄跻峰**,博士,教授,中国计算机学会(CCF)会员,主要研究领域为软件工程。

conflicting ways. Resolving such conflicts manually is both time-consuming and error-prone, particularly in large-scale software projects where merge conflicts occur in up to 50% of all commits. Therefore, there is a growing need for automated tools that can effectively assist in identifying and resolving merge conflicts with high accuracy and minimal human intervention. This paper presents an innovative approach to automating merge conflict resolution by leveraging the capabilities of large language models, such as ChatGPT. Our methodology goes beyond traditional approaches that focus solely on the conflicting code blocks by incorporating local code context into the resolution process. Specifically, we propose four types of code context: syntactic context, semantic context, program dependency context based on static analysis, and textual similarity context derived using BM25 retrieval techniques. These context types aim to provide the model with a broader understanding of the surrounding code structure, functionality, and relationships, thereby improving its ability to generate accurate and semantically meaningful resolutions. We conducted comprehensive experiments to evaluate how different prompting strategies—including temperature settings and prompt styles—affect the performance of LLMs in merge conflict resolution tasks. Our findings reveal that lower temperature values and standard chain-of-thought prompting significantly enhance the consistency and quality of generated solutions. At the line-level granularity, the use of program dependency slices extracted from static analysis improved resolution accuracy by approximately 42% compared to baselines without context. In contrast, at the token-level granularity, the BM25-based textual similarity context outperformed other methods in terms of perfect match counts and BLEU-4 scores, indicating its effectiveness in capturing fine-grained similarities between code segments. Moreover, our analysis shows that different types of code context have varying impacts on conflict resolution outcomes. Notably, the static analysis-based slice context introduced the least interference, with only 18.6% negative impact, demonstrating its robustness and reliability in supporting conflict resolution. Importantly, we also found that the benefits of incorporating code context are not limited to a single model but generalize well across different large language models, suggesting the broad applicability of our approach. In conclusion, this research contributes a promising framework for automating merge conflict resolution in collaborative software development environments. By integrating rich code context with powerful LLMs, our approach reduces the burden of manual resolution, minimizes the risk of errors, and ultimately enhances overall development efficiency.

Keywords　branch merging; automatic resolution of merge conflicts; code context; prompt engineering; ChatGPT

1　引　言

版本控制系统如Git在当前协作软件开发中非常常见，允许开发人员通过分支同时进行并行开发。分支的使用已成为一种普遍的趋势，因为开发人员能够在各自独立的工作空间中修改代码，在完成时将这些修改集成到主线中即可，而整合这些代码修改通常涉及合并源代码的多个版本。对GitHub上Java项目的大规模实证研究[1]表明，约12%的提交涉及分支合并。尤其在一些大型项目中，合并冲突占总合并的比例高达50%。此外，手动解决合并冲突非常耗时，因此需要探索自动化的解决方案。

机器学习和深度学习的发展使得自动解决合并冲突变得可能[2]。近年来，研究者们致力于探索如何更优地实现代码合并冲突的自动消解。最近的研究将机器学习算法和深度学习模型应用于合并冲突解决，提出了基于AI的冲突消解技术[3-4]，取得了不错的效果。比如DeepMerge[5]在研究中发现：大多数的冲突消解都只使用了冲突区域中的代码行[1]，基于此，DeepMerge将冲突块中的每个代码行作为

单位，使用一种类似指针网络[6]的神经网络结构对代码行进行重排列，将最后的排列结果作为冲突的解决方案，最终DeepMerge在它们的数据集上解决了约37%的冲突。而MergeBERT[3]在DeepMerge的基础上更进了一步，将冲突消解的粒度从行级别细化到了词元级别。他们的研究发现：几乎所有的词元级别冲突的解决方案可以被纳入到9个类别之中。因此MergeBERT将冲突消解转化为了分类问题。同时，Zhang[2]等人也开启了使用大语言模型（LLM）的强大能力来处理代码合并冲突的探索历程。他们设计了一种提示方法，将冲突块中相较于原内容发生了修改的部分标记为“＋”和“－”，然后再送入GPT-3中来生成答案。虽然GPT-3生成的答案效果不理想，但表明了大语言模型在处理冲突上的可行性。

然而，无论是定义固定合并模式的分类任务[5-7]，还是基于生成模型的自动回归生成任务，现有方法的输入信息以及所依赖信息通常都仅限于代码冲突块本身，而未涵盖冲突块之外的上下文信息[8]。例如在Zhang[2]、Svyatkovskiy A[3]和Dong J[4]的工作中，尽管冲突消解效果已显著提升，但其输入范式中仅包含独立的冲突块内容和编辑信息，未加入相关的代码上下文，因此这些方法的消解策略只能被归类为随机概率，而非证据充分的推理。实际上，代码冲突块的上下文信息对于解决合并冲突至关重要。上下文信息能够帮助开发人员理解代码变更的背景，明确冲突的起因及解决方法。通过正确理解上下文，开发人员可以判断哪些更改应保留，哪些更改需要处理。不过，并非所有上下文信息都对解决冲突具有积极作用，有些冗余信息反而会干扰消解过程。因此，检索出关键且有效的代码上下文对自动化解决代码合并冲突至关重要[9]。

当前大语言模型的发展速度相当惊人，其中，ChatGPT[10]是由OpenAI开发的大型语言模型，能够进行自然、流畅的对话。它不仅具有广泛的知识覆盖面，还在代码智能生成和理解任务中展现出强大的能力。ChatGPT能理解自然语言和编程语言的关系，生成符合语法和语义的代码，并能解析和理解给定的代码段，帮助开发人员理解代码的功能与逻辑，这使得利用LLM的强大能力来处理代码合并冲突成为可能。在自动解决合并冲突的任务中，关键的代码上下文信息可以作为ChatGPT的提示内容[11]，从而与其强大的语言理解和生成能力相得益彰。

本文旨在探索使用ChatGPT验证哪种类型的代码上下文在冲突自动消解中效果最佳，并探讨ChatGPT在此任务上的潜力。研究重点在于代码上下文的积极作用，而非冲突消解本身，具体关注以下问题：

（1）代码上下文是否能对合并冲突的消解起到积极作用？

（2）哪种类型的代码上下文对ChatGPT解决冲突效果最佳？

（3）基于最佳代码上下文，ChatGPT自动消解合并冲突的潜力有多大？

（4）上下文在其他大语言模型上的通用性如何？

为了回答上述问题，本文进行了系统、全面的实验，评估不同类型代码上下文在使用ChatGPT自动消解冲突中的表现。

本文的贡献包括以下几点：

（1）实证研究了代码上下文在代码合并冲突自动消解中的作用。

（2）提出了四种不同类型代码上下文的收集策略，为相关研究提供了有力支持。

（3）通过广泛实验评估了不同代码上下文在冲突消解中的有效性，结果表明关键的代码上下文能有效帮助解决冲突，尤其是基于静态分析的程序依赖上下文表现最佳。

论文接下来首先介绍背景与动机；接着详细介绍实证方法和策略；然后讨论数据集来源、评价指标以及实证结果；接着结合案例分析展开讨论；最后给出总结和展望。

2 背景与动机

本节主要介绍相关的背景知识与研究动机。

2.1 相关背景

在当前协作软件开发环境中，为了跨分支集成多个开发人员的代码变更，版本控制系统提供了合并算法。三路文件合并（例如“git merge”）是目前流行的合并算法[12]。三路合并算法以三个文件作为输入：公共祖先文件O，以及两个修改文件A和B。合并过程会产生以下两种结果：

（1）如果两个更改在同一行产生冲突，则会发生“冲突”。

（2）如果合并成功，则生成一个合并后的文件M，包含A和B中的所有更改。

对于大多数协作项目，约10%-20%的代码合

并会产生冲突[1]，处理这些冲突耗时费力[13-14]。开发人员必须中断当前工作，先了解冲突原因，再确定正确的解决方案。因此，为了减轻开发人员的手动劳动负担，使其从冗长乏味的工作中解脱出来，迫切需要加大对自动解决合并冲突技术研究的投入。

与此同时，当前LLM的发展迅速。ChatGPT[10]是由OpenAI开发的大型语言模型，能够进行自然、流畅的对话[15]。它不仅具有广泛的知识覆盖面，还能在代码智能生成和理解任务中展现出强大的能力。ChatGPT能理解自然语言和编程语言的关系，生成符合语法和语义的代码，并能解析和理解给定的代码段[16-18]，帮助开发人员理解代码的功能与逻辑，这使得利用大语言模型的强大能力处理代码合并冲突成为可能。

鉴于ChatGPT在代码生成和理解任务中展现出的强大能力[19-21]，本文使用ChatGPT来验证哪种代码上下文在代码合并冲突中效果最佳，同时主要探讨ChatGPT在自动解决代码合并冲突方面的潜力。

另一方面，合并冲突块的代码上下文信息对于冲突消解至关重要。代码上下文是指所有可以影响到合并冲突内容的代码片段，这些信息帮助开发人员理解冲突的背景，并且有助于避免破坏其他功能或引入新的错误。而获取这些关键上下文的核心在于确定“关键查询”，即基于哪些核心信息来检索相关的代码上下文。整个检索过程使用Diff3[22]算法来实现分支合并，并定位那些无法正确合并的冲突块。然而，此时识别出的合并冲突块是以行为单位的，即行之间的编辑冲突。若直接以整个冲突块为基准在整个冲突文件中检索代码上下文，则面临两大挑战：其一，由于冲突块可能包含多达数十行代码，这使得以行粒度检索代码上下文变得异常复杂；其二，一行代码通常由多个词元组成，虽然一行代码本身蕴含了丰富的信息，但其中仅少数几个核心词元才是真正的关键冲突信息。因此，以行粒度进行查询不仅效率低下，还容易被大量无关信息所干扰。

为此，本文提出基于词元粒度的Diff3来获取关键查询。词元粒度的Diff3指的是：基于行粒度的Diff3生成的冲突块，将三个版本的每一行代码拆分为一个个词元，再将每个词元独占一行，存放在各自版本的文件（A、B、O）中，然后再次使用Diff3对齐三个文件，生成词元粒度的小冲突块，如图1所示。词元粒度的小冲突块大幅缩小了查询范围，聚焦于真正发生冲突的词元，从而提取出真正重要的信息。

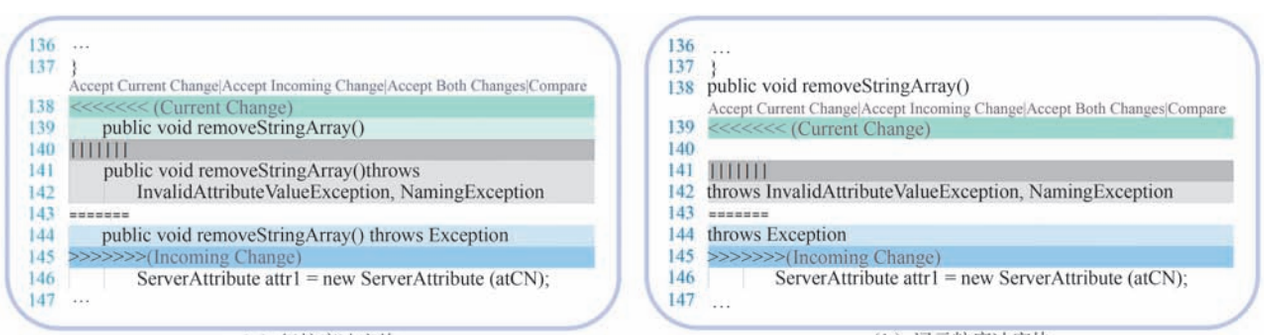

图1　行粒度与词元粒度的Diff3示例

2.2　研究动机

图1展示了一个开源项目源代码中简单的合并冲突块示例。当仅关注冲突标记内的代码行时，获取的信息仅限于双方分支在removeStringArray()方法中是否抛出异常及选择何种类型异常的差异。然而，难以理解当前分支为何选择不抛出异常，源分支为何决定抛出Exception异常的逻辑和目的。这种局限性源于无法访问冲突块之外的上下文代码，限制了对编码决策的深入理解。

事实上，冲突内的代码语句可能与冲突块之外未冲突部分进行交互，而这些交互对于理解冲突根源和解决方案至关重要。因此，仅依赖冲突块内的代码行是不够的。为全面理解并解决合并冲突，必须考虑冲突块之外的代码上下文，包括函数定义、变量声明及代码的逻辑结构等。通过综合分析完整的代码上下文，可以更准确地理解冲突的本质及各方代码变更的意图、目的，从而制定出更加有效的解决方案。

为了解决上述问题，本研究旨在探索如何通过提供适当的代码上下文信息来增强大型语言模型在自动消解代码合并冲突方面的表现。研究特别关注哪种类型的代码上下文最有助于解决冲突，并评估

了基于静态分析的程序依赖上下文等四种不同类型上下文的有效性。这一探索对于提高开发效率、减少人工干预具有重要意义，同时为未来利用大语言模型处理代码合并冲突的研究提供了有益的见解和策略。

3 研究设计

为了探索不同代码上下文对LLM自动解决代码合并冲突的影响，本文设计了四个研究问题(RQs)并进行了实验验证：

RQ1：不同提示风格和温度设置对ChatGPT解决合并冲突的影响如何？

RQ2：基于RQ1的最优设置，四种不同类型的代码上下文(3.1节)对ChatGPT解决合并冲突的效果如何？

RQ3：四种类型的代码上下文是否都能发挥积极作用？

RQ4：代码上下文的有效性能否推广到各种大语言模型？

RQ1旨在探索和理解提示风格及温度设置在代码合并冲突自动解决任务中对ChatGPT性能的影响。这为优化模型配置提供关键数据支持，帮助在后续研究中使用最佳设置，提升实验可靠性与有效性。

RQ2探讨在最优提示风格和温度设置下，不同类型的代码上下文对ChatGPT解决代码合并冲突的影响。通过比较这些上下文的有效性，研究旨在确定最有助于模型解决冲突的代码上下文类型，以进一步优化实际应用。

RQ3旨在探讨各类型代码上下文在合并冲突消解中是否都能发挥积极作用，并评估其中可能的消极影响。研究目标是识别并最小化对冲突消解产生不利影响的上下文类型，从而提高其准确性和可靠性。

RQ4旨在评估代码上下文对不同大语言模型的适用性，探讨其作用是否具有普遍性。

综合这四个研究问题，本文的最终目标是优化ChatGPT在代码合并冲突消解任务中的应用策略，结合最佳的提示风格、温度设置及最有效的代码上下文，以提升模型整体的冲突消解表现，解决当前方法中仅依赖冲突块内代码行所导致的信息不足问题。

3.1 研究问题设计

鉴于ChatGPT在代码智能生成和代码理解任务中展现出的强大语言理解和生成能力，本文主要使用ChatGPT来实证代码上下文在代码合并冲突中的效果，并探讨其在自动解决代码合并冲突上的潜力。

使用ChatGPT来实证和调查哪种类型的代码上下文对解决代码合并冲突最有效，其实验过程如图2所示。

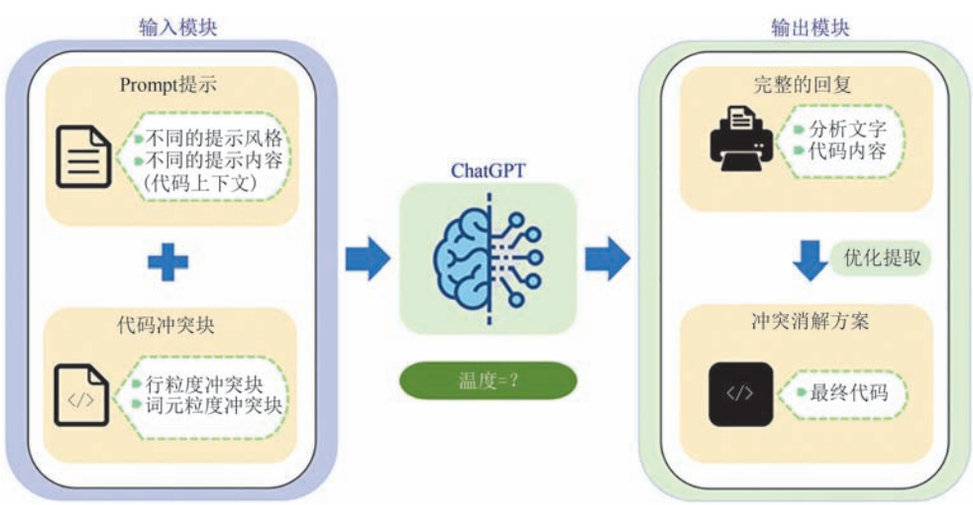

图2 总体方法概览图

数据输入主要包含两个部分：提示和冲突块。其中，提示是研究的重点，分为两个部分：提示风格和提示内容。首先，将代码上下文与冲突块组合为一个输入，使用OpenAI的API请求调用ChatGPT。选择使用API的原因在于其便于自定义参数，控制ChatGPT的准确性和稳定性，且在批量请求和反馈

处理上更为方便。其次，ChatGPT响应后会返回一个结果，但结果中除了包含解决方案代码，还包括一些冗余的文字信息（如分析文字）。最后，需要对反馈进行抽取，获取最终的冲突消解方案。

3.1.1　提　示

提示风格指的是把提示内容输入给ChatGPT时的输送形式。旨在探索何种形式的提示风格可以让ChatGPT更准确、更稳定地完成代码合并冲突消解任务的方法探索。这一研究目的是分析ChatGPT自身在解决代码合并冲突任务中的特性和潜力。

提示内容指的是各种类型的代码上下文。在本文中，采用了四种不同类型的代码上下文作为辅助信息，以研究哪种类型对协助ChatGPT解决代码合并冲突最有效。具体包括：

（1）基于冲突前后相邻的代码上下文：指位于Diff3冲突标记前后指定范围内的代码片段。这种上下文形式的核心在于获取冲突发生位置紧邻的代码行，以便开发人员能够通过这些相邻代码行快速了解冲突内容在代码文件中的定位及其相互关系。

具体来说，冲突前后相邻代码上下文包括当前合并冲突的冲突标记前、后紧邻若干行的代码行。这些代码行内容能够提供相邻的上下文信息来帮助大语言模型判断冲突的产生原因及其解决方式，具有非常高的直观性。

（2）基于BM25文本相似度的代码上下文：指利用BM25算法对冲突相关的代码片段进行文本相似度计算，进而提取出与当前冲突位置高度相关的代码上下文。

其中BM25（Best Matching 25）是一种经典的信息检索算法，用于衡量文档与查询之间的相关度。在当前的消解合并冲突场景中，BM25可以通过计算当前合并冲突代码行与其他非冲突代码片段（如代码库中相似函数或模块）的相似度，来帮助进一步识别出与当前合并冲突相似且存在关系的上下文代码片段。通过这种方式，大语言模型能够基于相关代码信息，预测合并冲突的可行解决方案。

（3）基于代码空间向量余弦相似度的代码上下文：指需要使用预训练的向量空间模型（如CodeBERT等）将代码片段转换为高维空间向量，并通过向量相似算法来度量不同代码片段之间的相似程度。

该方法的核心思想是结合词嵌入技术将代码的语法和语义信息编码为向量表示，因为在高维的向量信息中，不仅仅存在着文本上的相似信息还存在着丰富的结构信息。基于所生成的向量表示，可以通过计算冲突代码与其他非冲突代码片段间的相似分数，进而从当前的代码文件中找到与当前冲突相关的代码行（不仅是文本相关，而且是代码语义相关），从而提供更丰富的上下文信息。基于向量空间模型的上下文能捕捉代码中的潜在语义和结构相似性，有助于大语言模型更精确地理解冲突的原因和背景。

（4）基于静态分析的程序依赖代码切片上下文：指依赖于静态分析技术，依据程序的控制流图（CFG）、数据流图（DFG）等对程序进行依赖切片。代码切片通过分析程序中变量、函数和数据之间的依赖关系，提取出与冲突直接相关的代码片段。

这种上下文不仅关注冲突代码行本身，还涵盖了引发冲突的上下游代码片段，以及可能影响冲突解决的其他部分。应用静态分析技术，能够揭示潜在的数据依赖关系和控制依赖关系，提供准确的、与程序行为紧密相关的上下文信息，并且能够帮助大语言模型避免在解决合并冲突时破坏程序的整体逻辑。

每种类型代码上下文的获取和检索方法将在后续RQ2的详细实验设置中进行说明。因此，提示内容的研究是在确定最优提示风格后，进一步探索不同代码上下文的效果。

3.1.2　合并冲突块

数据输入的另一部分是合并冲突块，指通过版本控制系统在分支合并过程中产生的、带有特定冲突标记的代码片段。冲突块通常以标准格式呈现出现，包含两个分支的修改内容以及公共祖先文件的原始内容，这些内容被特定的冲突标记符号分隔，从而在整体上呈现为一个合并冲突块的形式。

在本文中，合并冲突块分为两种粒度：行粒度合并冲突块和词元粒度合并冲突块。如图1所示，行粒度冲突块是经过三路文件合并算法后所呈现的默认形式，其以完整代码行为单位，比较直观且易于识别出冲突位置。然而，行粒度下的冲突内容可能包含着大量无关信息。为此，本文引入了更细致的词元粒度冲突块，通过对已冲突代码行的进一步拆解合并，呈现出更加细粒度的冲突内容，方便更加聚焦于发生冲突的核心词元，从而大幅减少冗余信息干扰。

两种粒度的合并冲突块在不同场景下各有优势：行粒度适合快速定位大范围冲突，而词元粒度则

在精确提取冲突关键信息方面表现出色。本文在实验中对两种冲突块形式进行了对比，尝试探索词元粒度的改进对ChatGPT生成高质量冲突解决方案是否具有积极作用。

3.1.3 研究设计详述

RQ1：不同提示风格与温度设置对ChatGPT解决合并冲突的影响分析

此研究问题关注不同设置对ChatGPT消解合并冲突的性能影响：不同的提示风格和温度设置如何影响ChatGPT在代码合并冲突自动解决任务中的表现。ChatGPT在代码智能任务中的有效性很大程度上取决于实验中使用的提示风格和具体温度。因此，本文设计了3种具体的提示风格，分别考虑是否在提示中提供详细的任务要求或者专业设定的工作场景。同时，本文选择了4个温度设置来探讨相应不同温度区间对于ChatGPT冲突消解能力的影响分析，从0到1.5之间的范围，每组间隔0.5，即0.01、0.5、1.0和1.5共4个设置。通过在部分选定的冲突数据集进行相关实验，来评估和比较这3种提示风格和4种温度设置的具体表现。另外，在后续所有的RQ评估中，ChatGPT的所需参数皆基于此RQ1中获得的最佳提示风格和温度进行设置。

RQ2：基于最佳配置下不同类型代码上下文的作用分析

本研究问题的目的是基于RQ1中确定的最优提示风格和温度设置下，进一步评估不同类型的代码上下文是否能够对ChatGPT自动解决代码合并冲突产生有效作用以及最优效果。

本文定义并收集了以下四种不同类型的代码上下文：基于冲突前后相邻的代码上下文、基于BM25文本相似度的代码上下文、基于代码空间向量余弦相似度的代码上下文、基于静态分析的程序依赖上下文。所有类型的代码上下文均基于RQ1结果中的最优提示风格和最优温度设置开展具体实验。同时，本研究通过分别对比行粒度与词元粒度下的冲突消解方法，来进一步验证代码上下文在解决合并冲突中的必要性和有效性。

RQ3：各代码上下文的积极作用与潜在负面影响评估

此研究问题旨在探讨分析各类代码上下文在辅助大语言模型消解冲突时是否都可以产生积极作用，或者是否存在一些上下文不仅没有任何效果，反而降低了原本无任何上下文时的冲突消解正确率的情况。

例如，在某些情况下，ChatGPT在没有上下文的裸合并冲突块中已经可以对合并冲突进行完美正确消解，但在加入某些代码行之后，反而扰乱了大语言模型的冲突解决节奏，最终导致生成了错误的解决方案。因此，为了深入探讨各类型代码上下文的积极与消极影响，本文分析了四种不同上下文分别与无上下文情况下大语言模型生成完美匹配解决方案的交集占比情况，以评估量化每种代码上下文的消极作用，并最终综合确定哪种类型的代码上下文可以带来最小的消极影响。

RQ4：代码上下文跨语言模型的有效性验证

此研究问题旨在评估在不同的通用大语言模型上，本文所提出的四种代码上下文是否能够保持一致的有效性以及一致的影响趋势。具体来讲，本文将考察多种国内外大语言模型（如GPT系列、通义千问系列、混元系列等）在相同四种代码上下文增强输入下的合并冲突消解表现，并分析相同代码上下文在不同模型下的影响效果，以验证关键的代码上下文跨模型的通用有效性。同时，本研究还将针对相同系列的不同参数以及不同领域模型（专业代码大模型）在基于代码上下文时冲突消解的表现进行深入对比分析，以探究是否参数越大的模型或是越专业的代码模型会更好地从代码上下文中汲取知识。

3.2 代码切片上下文

代码切片[23]可以获取与已冲突代码直接相关的代码行，将这些相关的非冲突代码行与已冲突代码合并，形成丰富的代码上下文信息。具体来说，这类代码上下文是通过组合已冲突的代码语句和依赖与被依赖于已冲突代码的非冲突代码语句创建的。

如图3所示，基于静态分析的程序依赖代码切片上下文的获取过程主要包括构造程序依赖图、提取代码切片，并最终构造代码切片上下文信息，详细的代码切片上下文获取步骤如算法2所述。其具体过程和技术说明如下。

首先，使用静态分析工具Joern[24]针对冲突代码所在的源文件构建程序依赖图。由于冲突块包含三个主体内容（当前分支修改、源分支修改、公共祖先提交内容），本文分别针对3个冲突内容对应的源文件独立构建依赖图，并单独提取切片。接着，从程序依赖图中提取与已冲突语句存在程序依赖关系的代码行。切片操作包括：

（1）提取对已冲突代码具有数据依赖和控制依赖的语句。

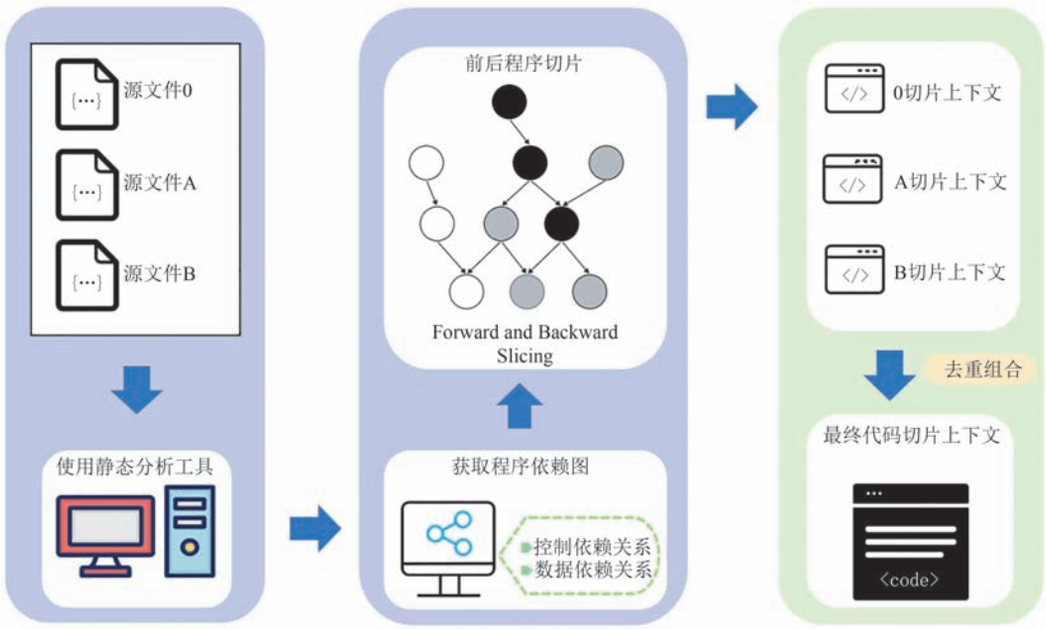

图3　代码切片上下文获取过程

(2)基于冲突行在程序依赖图中进行向后和向前切片。

(3)应用跨方法切片技术,确保与函数外密切相关的语句也可以被检索。

本文默认切片深度为1层,以避免获取过多冗余的上下文信息,减少资源消耗和不可控性,仅提取最直接相关的代码切片上下文,详细的程序依赖切片过程如算法1所述。

最后,通过去重组合3个独立切片结果,形成最终的冲突块程序依赖代码切片上下文。

算法1.　程序依赖切片算法

输入:节点路径 *nodes_path*,边路径 *edges_path*,冲突行 *change_lines*,深度 *depth*

输出:依赖行列表 *lines*,冲突行列表 *c_lines*

1. IF *change_lines* 为空 THEN
2.　　RETURN *fault*;
3. END IF
4. IF *nodes_path* 或 *edges_path* 不存在 THEN
5.　　RETURN *fault*;
6. END IF
7. 打开 *nodes_path* 文件并读取节点数据 *nodes*;
8. *nodes* = [*n* FOR *n* IN *nodes* IF '*lineNumber*' IN *n*];
9. 打开 *edges_path* 文件并读取边数据 *edges*;
10. IF *nodes* 的长度等于 10 或 *edges* 的长度小于 20 THEN
11.　　RETURN *fault*;
12. END IF
13. FOR *line* IN *change_lines* DO
14.　　IF *lines_code* 不为空且 *line* 不在 *lines_code* 中 THEN
15.　　CONTINUE;
16.　　END IF
17.　　*c_lines*.append(*line*);
18. END FOR
19. *lines_dependence*. update (get_dependece_line (*change_lines*, *nodes*, *edges*, *depth*));//获取依赖行
20. *lines*=sorted(list(*lines_dependence*));//对依赖行排序
21. RETURN [*lines*, c_*lines*];

算法2.　代码切片上下文获取算法

输入:冲突代码块包含:当前分支修改 *a*,源分支修改 *b*,公共祖先内容 *o*,当前分支文件 *currentBranch*,源分支文件 *sourceBranch*,公共祖先文件 *commonAncestor*

输出:最终的程序依赖代码切片上下文 *finalSliceContext*

1. *finalSliceContext* = [] *tempSliceResults* = [] // 初始化存储
2. FOR each content IN [*currentBranch*, *sourceBranch*, *commonAncestor*] DO
3.　　*lineNumbers*=*DFS*([*a*, *b*, *o*], *content*)//获取行号
4.　　*PDG*=*buildPDG*(*content*)//构建程序依赖图PDG
5.　　*sliceResult* = [] // 初始化当前内容的切片结果
6.　　FOR each *line* IN *lineNumbers* DO
7.　　//执行向前向后切片,并应用跨方法切片技术
8.　　*localSlice*=*forwardBackwardSlice* (*PDG*,

line)+crossMethodSlice(*PDG*, *line*)
9. *sliceResult*.append(*localSlice*)
10. END FOR
11. *slice*=*limitDepth*(*sliceResult*, depth=1)//切片深度
12. *tempSliceResults*.append(*slice*)
13. END FOR
14. *finalSliceContext*=*mergeAndDeduplicate*(*tempSliceResults*)
15. RETURN *finalSliceContext*

4 实 验

4.1 数据集

本文在优质Java项目数据集[1]的基础上，评估了ChatGPT在不同类型代码上下文协助下自动解决代码合并冲突的能力。该数据集由Gleiph Ghiotto等人[1]精心构建，经过严格筛选，从近200万个高质量开源软件存储库中挑选出具有代表性的项目，最终涵盖了2731个活跃且受欢迎的项目，并完整保留了所有合并冲突的提交记录。数据集不仅规模庞大，还具有丰富的多样性和代表性，共包含25328个失败的合并和175805个冲突块，覆盖了不同复杂度的合并场景。每个失败的合并记录详细记录了两个冲突版本A和B、公共祖先版本O，以及开发人员精心设计的解决方案M。

由于2731个项目过于庞大，且部分项目的代码合并冲突数量较少，会增加数据收集的成本。为此，本文仅选择了冲突数量最多的前50个项目，过滤后进行数据收集。这50个项目共收集了73779个行粒度的代码合并冲突块。考虑到生成式大模型(ChatGPT)的API收费基于token数量，为避免资源浪费，本研究从73779个合并冲突中按分类比例随机提取1500条数据进行后续实验测试。

分类规则采用了Gleiph Ghiotto等人[1]提出的6大类代码合并冲突解决方案，包括：

(1)Version A (A)：采用版本1的代码。

(2)Version B (B)：采用版本2的代码。

(3)Concatenation AB (CC12)：按AB顺序串联两个版本的代码行。

(4)Concatenation BA (CC21)：按BA顺序串联两个版本的代码行。

(5)Combination CB (CB)：以某种交错顺序从两个版本中选择代码行，不编写新行或修改选定的行。

(6)New code (NC)：将一个或两个版本的现有代码与新编写的代码混合使用。

具体解决策略的类型占比数据分布情况见表1。其中NULL表示无法精确收集到对应解决方案的数据类型。

表1 数据集中各解决策略类型的占比分布

解决策略	个数	占比	分配个数
A	29171	47.69%	705
B	13509	22.08%	345
CB	6653	10.88%	165
CC12	1289	2.11%	30
CC21	482	0.79%	15
NC	10062	16.45%	240
NULL	12613	-	-

为了在节省资源的同时确保实验测试的真实性，本文根据这6类冲突的占比，按比例随机抽取相应数量的合并冲突块，以尽量还原数据集中的实际冲突情况，使得实验结果更贴近实际情况。

4.2 评价指标

4.2.1 BLEU

BLEU(Bilingual Evaluation Understudy)是一种用于评估机器翻译质量的指标[25]。通过比较生成样本与参考样本的相似度，评估指定任务的准确性和流畅度。BLEU分数范围为0到1，分数越接近1，表示生成的代码合并冲突解决方案越接近开发人员的实际解决方案，即解决方案质量越高。本文采用的是BLEU-4，表示使用4-gram(4个连续词)作为评估单位。相比标准BLEU，BLEU-4更加注重长序列的准确性，因此可以更好地评估冲突解决方案的流畅性和连贯性。

BLEU-4的标准公式如公式(1)所示：BP为简短惩罚(Brevity Penalty)，用于惩罚生成的翻译过短的情况，其中c为生成的冲突解决方案(代码)的长度，r为真实解决方案(代码)的长度。ω_n是每个n-gram的权重，对于BLEU-4，通常每个$\omega_n=1/4$，即各个n-gram的权重均等。p_n是n-gram的精确度，定义为生成冲突解决方案中n-gram匹配的比例。

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{4} \omega_n \log p_n\right) \tag{1}$$

其中，$BP=\begin{cases}1, & \text{if } c>r\\ \exp\left(1-\dfrac{r}{c}\right), & \text{if } c\leqslant r\end{cases}$。

4.2.2 完美匹配

完美匹配(Exact match)：在生成代码合并冲突解决方案的任务中，完美匹配指标用于评估生成的

解决方案与真实解决方案是否完全相同。具体而言，它衡量生成的解决方案中每个元素（如代码片段、注释等）是否与真实解决方案中的对应元素完全匹配。如果所有元素都完全匹配，则得分为1；否则得分为0。完美匹配指标有助于确定生成的解决方案与真实解决方案的完全一致性，对于评估解决方案的质量和准确性至关重要。

完美匹配的标准公式如公式（2）所示：Exact match初始值为0，若生成的消解方案x与开发者的真实解决方案y完全一致，则完美匹配计数加1，否则不变。

$$Exact\ match=\begin{cases}1+Exact\ match, & \text{if } x=y\\ Exact\ match, & \text{if } x\neq y\end{cases} \tag{2}$$

4.3　RQ1：提示设计的影响

4.3.1　具体实现

温度（Temperature）和提示（Prompt）是影响ChatGPT在代码智能生成任务中性能的两个关键参数[26-27]。为确定这些参数的最优值，本文通过实验评估它们对自动解决代码合并冲突任务的影响。需要注意的是，这里的提示是指提示风格，而非影响冲突的具体内容。具体提示内容将在RQ2中讨论。

在ChatGPT API[28]中，温度参数用于控制生成文本的多样性。在执行代码合并冲突解决任务时，调整温度参数会影响解决方案的多样性和创造性。具体而言，较低的温度值会生成更加保守和可靠的解决方案，可能更符合开发人员的期望，但缺乏创新性。较高的温度值则会生成更多样化和创新的解决方案，可能包含更多变体和非传统方法，但也可能出现与预期不符的结果。在本次研究中，选择了0.01、0.5、1.0和1.5这四种温度设置，旨在覆盖从低随机性到高随机性之间的不同场景，以全面评估温度参数对代码合并冲突消解任务的影响。因此，本次研究选择的四种温度设置足以覆盖不同场景下的性能表现，并为后续研究提供了合理的参数范围。

为选择最合适的提示风格，本文遵循了最佳实践[29]，其中提示可包含三种类型的内容：指令、输入数据和输出指标。本文设计了三种不同类型的提示风格，并在不同的冲突消解任务中比较其表现效果，具体如图4所示。实验通过API中的角色扮演功能指定具体的提示风格，而非直接影响提示内容。该做法的好处是将提示风格与提示内容分开研究，避免相互干扰。

P1: Simple Prompt
You are a senior Java program developer with extensive experience in resolving code conflicts caused by branch merging. Firstly, the code lines outside of the conflict block are given to you, which can help you understand the conflict and further resolve it; Secondly, what I am giving you is code merging conflict blocks. Finally, please output the final code after merging and resolving conflicts.

P2: COT Prompt
You are a senior Java program developer with extensive experience in resolving code conflicts caused by branch merging. Please strictly follow the following steps to respond to user input:
Step 1： The user uses the XML tags <ConflictBlock> </ConflictBlock> to wrap the entire code and merge the conflict block provided to you. Please understand this conflict first and find its essence.
Step 2： The user uses XML tags <context> </context> to wrap multiple context code lines that are discrete outside of this conflict block and provide them to you. Please first analyze whether each context code line inside it is directly related to the conflict content within this conflict block.
Step 3: Please use the essence of this conflict in step 1 and the context code lines directly related to the conflict content in step 2 as a basis to infer how to resolve this code conflict.
Step 4: Based on the above analysis and requirements, please use the XML tags <resolution> </solution> to output the final conflict resolved code within the conflict block.

P3: COT Only Code Prompt
P2+Do not output any additional analytical text, and do not output any content other than code.

图4　三种提示风格具体指令

（1）简单提示（Simple Prompt）：仅提供基于冲突块和上下文生成解决方案的基本需求和场景，没有额外描述。

（2）思维演绎（COT）：在简单提示的基础上，提供更完整的提示风格。例如思维链[30]、step by step、严格输入输出、显示思考过程等[31]。

（3）思维演绎仅有代码（COT only code）：基于完整的提示风格，但要求输出时只输出解决方案代码，不包含思考过程。

为评估不同提示风格和温度设置对ChatGPT在代码合并冲突自动解决任务中的影响。本文从行粒度和词元粒度两个视角展开实验。具体而言，基于预先定义的提示风格来对ChatGPT的API发起调用，分别对行粒度和词元粒度冲突数据集进行合

并冲突的自动消解。为了提升执行效率以及优化资源利用，本文在基础数据集中随机选择了100个合并冲突数据条目进行相关实验。同时考虑到ChatGPT生成结果的随机性，本文针对每组合并冲突数据元组独立请求3次对话。对于完美匹配指标，本文取3次对话中的最佳结果作为最终结果；而对于BLEU-4指标，则是取3次对话的平均分数作为最终结果。

4.3.2 实验结果

(1)温度设置分析

表2展示了不同温度设置下，ChatGPT在执行冲突消解任务时的性能评估结果。该评估基于随机选取的100个测试实例，考察了ChatGPT在三种不同的提示风格下，针对无上下文信息的行级别和词元级别的冲突块生成解决方案的能力，并通过与真实解决方案对比计算出的平均BLEU-4分数进行量化。本次实验特意聚焦于无上下文条件下的冲突块，旨在排除其他可能变量(如上下文信息)的干扰，从而精确分析温度参数对模型冲突消解效果的影响。在这一评估中，较高的BLEU-4分数表明所提出的解决方案更接近于理想的真实解决方案。

表2 不同温度下ChatGPT执行冲突消解任务的BLEU值

提示风格	冲突块	0.01	0.5	1.0	1.5
Simple Prompt	行粒度	0.7005	**0.7010**	0.6901	0.6857
	词元	**0.7382**	0.7352	0.7246	0.7191
COT	行粒度	**0.6847**	0.6845	0.6762	0.5736
	词元	**0.7341**	0.7338	0.7189	0.5615
COT only code	行粒度	**0.7185**	0.7079	0.7032	0.6459
	词元	**0.7426**	0.7367	0.7397	0.7263

评估结果表明，将温度设置为0.01时，ChatGPT的合并冲突消解能力在三种提示风格下的行粒度和词元粒度下均能获得最佳性能和稳定性。随着温度的升高，ChatGPT的性能显著下降。例如，温度设置为1.5时，表现最差。这是因为生成合并冲突的解决方案是一项复杂且精确的任务，高温度会导致不稳定和随机的生成结果，虽然更具创造性，但不可靠，这一现象在COT提示风格下表现尤为明显。上述结果表明，在该任务中，较低的温度设置能够更好地平衡生成结果的准确性和稳定性。然而，这并不意味着温度设置越低越好。温度参数的选择需要根据具体任务的需求进行权衡。对于代码合并冲突消解任务，由于其对准确性和稳定性的高要求，较低的温度设置(如0.01)更为合适。但在其他需要更多创新和多样性的任务中，适度提高温度会更有益。由于本研究的主要目标是探索哪种代码上下文能更有效地协助解决冲突，而ChatGPT在此仅作为工具，因此需要其尽力保障冲突消解任务的稳定性。

总体而言，较低的温度设置有助于代码生成任务的稳定性和输出质量。因此，在后续的提示风格实验中，本文基于温度0.01进行相关研究。

(2)提示风格分析

本部分实验的主要目标是探索哪种提示风格能够使得ChatGPT最有效地利用代码上下文信息，从而优化冲突消解的性能。因此，本文特别增加了四种包含代码上下文的场景，以进一步分析不同提示风格的效果。与上述探讨温度设置影响不同，本部分的研究重点在于识别最有效的提示风格，以最大化上下文信息的效用。

表3展示了在温度设置为0.01的条件下，不同提示风格对ChatGPT执行冲突消解任务的影响评估。由于合并冲突消解中，任何细微错误都会导致代码运行出错甚至编译失败，因此使用严格的完美匹配指标来衡量提示风格的优劣。表中的结果值是生成解决方案与真实解决方案完全匹配的个数。

表3 在不同提示风格下ChatGPT完美消解冲突的个数

提示风格	Simple Prompt	COT	COT only code
行粒度无上下文	17	**36**	24
行粒度前后相邻上下文	17	**20**	16
行粒度文本相似上下文	27	**51**	32
行粒度空间相似上下文	23	**38**	31
行粒度程序切片上下文	22	**46**	32
词元无上下文	20	20	**21**
词元前后相邻上下文	**19**	13	13
词元文本相似上下文	18	**31**	21
词元空间相似上下文	27	**35**	28
词元程序切片上下文	25	**31**	28

表3的结果显示，无论在缺乏上下文支持还是在有各种类型上下文情境下，采用COT提示风格所获得的完美匹配解决方案数量均高于其他提示风格。这表明，相较于其他两种提示风格，COT提示风格不仅能更充分地激发ChatGPT的冲突消解潜能，还能更加有效地利用代码上下文信息。

然而，最后一列的COT仅输出代码却效果不佳。尽管提示风格相似，但仅要求ChatGPT输出解决方案完整代码的设定，违背了“提示工程设计中显

示思考过程”这一原则，导致结果不理想。本文最终选择COT提示风格进行后续实验。

4.4　RQ2: 不同代码上下文的效果

4.4.1　具体实现

基于RQ1中的最佳参数（使用温度为0.01、COT提示风格），本文在不同类型的上下文数据集上评估了它们对ChatGPT进行合并冲突自动消解任务的协助效果。具体来说，本文研究了以下四种类型的代码上下文。

基于冲突前后相邻代码上下文：获取方式较为简单。首先定位冲突块，然后分别向前和向后延伸获取相邻的n行代码，在本研究中n设为3。

基于BM25的文本相似代码上下文：如图5左所示，使用BM25算法寻找相关代码行。BM25算法考虑文档长度和查询项频率，适用于文本相似度匹配。首先，删除合并文件中的冲突块，避免检索到冲突内代码行。然后，应用词元粒度Diff3获取冲突块的词元作为查询项，利用Lucene算法库[32]对非冲突代码行建立索引。最后，通过BM25[33]算法快速检索最相似的代码行。

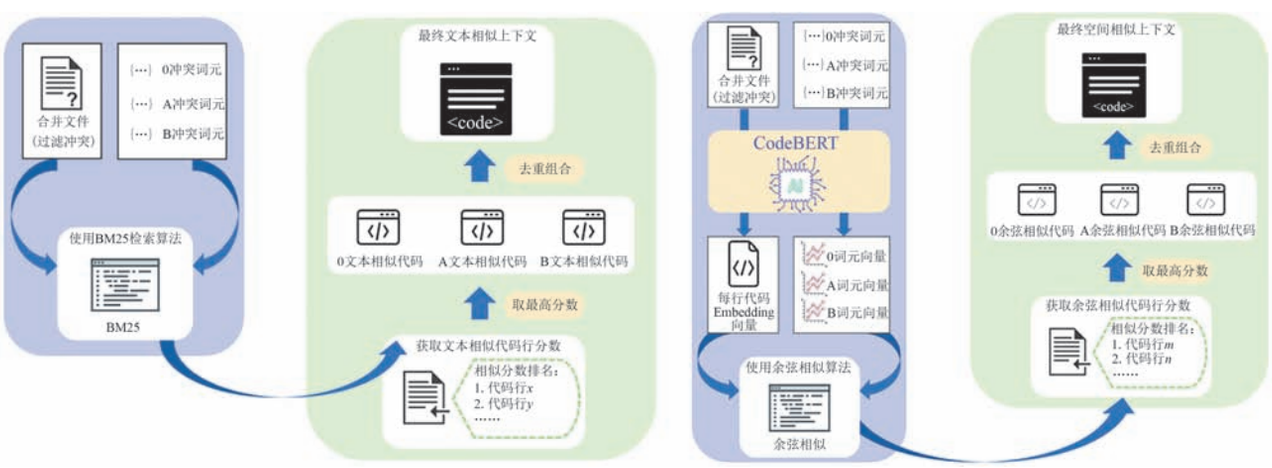

图5　其他各类型代码上下文的获取流程图

基于代码空间向量的余弦相似代码上下文：如图5右所示，首先使用预训练代码模型CodeBERT[34]表征合并文件中的每一行代码，过滤掉冲突块。然后，应用词元粒度Diff3提取冲突块的词元集合，利用CodeBERT提取特征向量作为查询。最后，计算查询向量与代码行向量的余弦相似度，获取最相似的代码行。

基于静态分析的程序依赖代码切片上下文：通过静态分析工具Joern[24]构建程序依赖图，并使用代码切片技术提取对冲突代码语句具有数据依赖和控制依赖的代码行。此过程包括向前和向后切片，以及跨过程切片，确保相关代码被包含。为了避免过多上下文导致的不稳定性，本文只获取一层最直接相关的代码切片上下文。

在上述各种代码上下文的收集方法中，其实现复杂性和计算效率都各具特点。首先，基于冲突前后相邻代码上下文获取方式比较简单直接，只需定位冲突块并向前后扩展n行代码即可，其计算开销最小且获取时间最快，但提供的上下文信息有限。其次，BM25文本相似度方法利用经典的信息检索技术，通过Lucene库快速索引和检索与冲突相关的代码片段，尽管其具有处理大规模数据的能力，但在检索过程中需要独立建立倒排索引结构，因此其检索速度也并没有非常大的优势，并且其结果质量高度依赖于索引的质量和查询词的选择。第三种方法基于代码空间向量余弦相似度方法，其获取过程需要使用预训练模型将代码表征为高维向量，最终通过计算余弦相似度的分数来找到最相关的代码行，由于该方法需要大量的空间向量计算，以及对应大量内存和计算资源的需求，因此较为耗时。最后，基于静态分析的程序依赖代码切片上下文需要首先通过静态分析工具来构建程序依赖图，并使用代码切片技术来提取直接相关代码行，虽能提供详细上下文帮助理解冲突，但其中的静态分析工具的程序依赖图构建是属于运行时间最长的一类，尤其是在较为复杂的项目中。

获取这四种类型的代码上下文后，本文按照图4所示的预先定义的提示风格向ChatGPT发起API请求调用。具体来说，输入中包含两个主要部分：一个是使用<Context></Context>包裹的上

下文代码行，另一个是使用＜ConflictBlock＞＜/ConflictBlock＞包裹的具体合并冲突块(行粒度或者词元粒度)。其中，实验中的具体提示风格本文是通过角色扮演的形式输入，这一步骤在代码上下文输入之前进行，可以提前帮助大语言模型理解具体的输入格式和内容。

4.4.2 实验结果

(1)行粒度(line-level)

表4显示了在不同类型代码上下文提示下，ChatGPT在行粒度下执行冲突消解任务的评估结果。本文将无代码上下文和冲突前后相邻上下文的冲突消解效果作为基线标准，因为当前主流的冲突消解方法[3]通常不提供额外的代码上下文(或仅基于前后延伸的代码行)，因此将其作为评估的基线。表格中的结果基于随机抽取的1500条数据，显示了生成解决方案与真实解决方案之间的*n*-gram分数、平均BLEU-4分数以及完全匹配的解决个数。

表4 在不同类型代码上下文提示下对ChatGPT从行粒度执行冲突消解任务的评估结果

代码上下文	Exact match	1-gram	2-gram	3-gram	4-gram	BLEU-4	提升幅度	正确率
行粒度无上下文	384	0.8358	0.7341	0.6669	0.6133	0.6525	--	25.60
行粒度＋前后相邻上下文	306	**0.9025**	**0.8284**	**0.7800**	**0.7395**	**0.6880**	↓20.31%	20.40
行粒度＋文本相似上下文	491	0.8371	0.7392	0.6728	0.6217	0.6464	↑27.86%	32.73
行粒度＋空间相似上下文	528	0.8029	0.6933	0.6236	0.5712	0.6116	↑37.50%	35.20
行粒度＋程序切片上下文	**544**	0.8334	0.7347	0.6709	0.6205	**0.6485**	**↑41.67%**	**36.27**

从表4可见，除了前后相邻代码上下文外，所有类型的代码上下文都使冲突解决个数比基线无上下文输入提高了27%-46%。这表明提供与冲突块相关的代码上下文有助于模型更好地理解冲突背景及语境，进而更准确地解决冲突。然而，加入前后相邻上下文反而使消解效果变差，说明虽然提供更多信息有助于解决冲突，但必须避免冗余信息，聚焦于与冲突块紧密相关的上下文。

另外，基于代码空间向量的余弦相似上下文和程序依赖代码切片上下文在完美匹配个数上表现相近，均显示出较强的协助潜力。但从n-gram分数和BLEU-4指标来看，余弦相似上下文的分数较低，表明其生成方案的随机性较大，稳定性不足。而对于前后相邻代码上下文，其完美匹配个数明显低于其他方法，甚至比无上下文的基线低约20%。然而，其n-gram分数以及BLEU-4分数却最高，这表明该上下文虽然不适合完全自动化的消解任务，但在提供解决方案以供人工参考方面具有显著优势。因此，在不同使用场景中，应根据需求选择适合的代码上下文。

综上所述，基于静态分析的程序依赖代码上下文在所有代码上下文中表现最佳，相较于无上下文，冲突解决个数平均增加了42%，且其平均BLEU-4分数也较高，表明该上下文不仅提高了创造性解决方案的生成效率，还保证了稳定性。这主要归因于程序依赖图能够清晰地展示代码中变量之间的数据依赖关系以及控制流中的条件依赖关系。这种依赖关系的捕捉使得模型能够更准确地理解代码的执行逻辑，从而在解决冲突时能够更好地识别和修复冲突。同时，静态分析工具可以精确捕捉数据依赖和控制依赖关系，帮助模型准确理解冲突上下文并提出合理解决方案，相比基于相似度的上下文，其一致性和稳定性更高一些。

(2)词元粒度(token-level)

表5显示了在不同类型代码上下文提示下，ChatGPT在词元粒度下执行冲突消解任务的评估结果。本文发现，词元粒度的结果与行粒度不同。在行粒度下，基于静态分析的程序依赖代码切片上下文效果最佳，但在词元粒度下，BM25文本相似代码上下文表现最好，无论是完美匹配个数还是n-

表5 在不同类型代码上下文提示下对ChatGPT从词元粒度执行冲突消解任务的评估结果

代码上下文	Exact match	1-gram	2-gram	3-gram	4-gram	BLEU-4	提升幅度	正确率
词元粒度 无上下文	330	0.8406	0.7532	0.6911	0.6400	0.6579	--	20.00
词元粒度 ＋ 前后相邻上下文	341	**0.9025**	**0.8365**	**0.7945**	**0.7581**	**0.7086**	↑3.33%	22.73
词元粒度 ＋ 文本相似上下文	**479**	0.8377	0.7472	0.6849	0.6362	**0.6642**	**↑45.15%**	**31.93**
词元粒度 ＋ 空间相似上下文	456	0.8130	0.7161	0.6491	0.5969	0.6336	↑38.18%	30.40
词元粒度 ＋ 程序切片上下文	**470**	0.8364	0.7463	0.6844	0.6336	0.6513	**↑42.422%**	**31.33**

gram以及BLEU-4平均分数。其原因是本文在收集BM25文本相似代码上下文时，先进行了词元粒度的冲突合并，再以关键词元作为检索源，因此检索出的上下文对词元粒度的冲突消解尤为敏感且适合。

另一方面，本文观察到行粒度下的完美匹配解决个数普遍高于词元粒度，但词元粒度的BLEU-4平均分数更高。这意味着词元粒度的消解方案更接近真实解决方案，然而在接近真实方案的最后一步常常会出现细微误差，导致未能达到完美匹配的标准。这是由词元粒度冲突块的特性决定的，Diff3处理后将行粒度的冲突块分为未冲突词元集和冲突词元集，其中未冲突词元集与真实解决方案往往高度相似。因此，行粒度的解决个数更优，但词元粒度的相似度分数则明显高于行粒度。

(3)结果分析

本研究通过对比不同粒度(行粒度与词元粒度)的冲突消解方法，验证了代码上下文在解决合并冲突中的必要性和有效性。实验结果显示，在两种粒度下，代码上下文均能提升冲突消解的效果，但行粒度的方法相较于词元粒度表现出了更优越的性能。尽管如此，即使在最佳代码上下文的支持下，ChatGPT的最高准确率仅为37%，这不仅反映了本研究中数据集筛选标准的严格性，也凸显了自动化解决代码合并冲突任务的复杂性。

此外，虽然引入代码上下文信息显著提升了大语言模型的性能，其总体正确率仍不及当前最先进的其他方法。例如，DeepMerge利用代码行重排解决了80%的冲突，但其解决率仅达37%；而Zhang等人使用GPT-3在自己的数据集上取得了44.1%的正确率。最引人注目的是MergeGen，它采用Transform架构直接处理细粒度的冲突表示，达到了66.7%的正确率，展示了生成式模型的强大潜力。

随着技术的发展，如ChatGPT和GPT-4等模型的进步预示着LLM在自动冲突消解领域的能力有望得到显著增强。本研究表明，尽管目前LLM的表现尚不及经过特别优化的模型，但由于相关工作的实验环境无法完全复现，因此不能彻底否认LLM在此领域的潜力。更重要的是，本研究通过纳入多种类型代码上下文信息，特别是基于静态分析的程序依赖关系，显著提高了冲突解决的成功率，为未来LLM在自动化冲突消解中的应用展示了广阔的前景。

4.5　RQ3:代码上下文有效性的验证

在研究代码上下文对解决合并冲突的影响时，本文分析了四种不同类型的代码上下文及其与无上下文的对比。本节根据实验结果定义了三种影响关系：积极影响、消极影响和模糊积极影响。具体而言：

(1)积极影响：指原本在没有上下文的情况下无法成功消解的冲突，在引入特定代码上下文后得以成功解决。

(2)消极影响：指原本在没有上下文的情况下可以成功消解的冲突，在引入特定代码上下文后反而未能成功解决。

(3)模糊积极影响：指原本在没有上下文的情况下能成功解决的冲突，在引入特定代码上下文后仍能成功解决，但并未显著改善消解结果。

如图6所示，通过对比基于BM25文本相似性、代码空间向量余弦相似性以及静态分析程序依赖代码切片等方法获得的代码上下文，发现它们不仅有助于保持大部分无上下文情况下成功的冲突消解，还额外解决了许多无上下文下未能解决的冲突，显示出强烈的积极作用。

然而，部分上下文信息引入后反而导致了一些本可成功解决的冲突出现错误消解方案，表明这些上下文具有消极影响。特别地，基于静态分析的程序依赖代码切片上下文表现出最低的消极影响(仅18.6%)，而基于前后相邻上下文的方法消极影响最大(接近43%)。本文认为其原因是后者获取代码行的方式过于简单，导致冗余信息干扰了解决过程。另外，BM25方法有时会产生负面影响，这是因为BM25主要依据文本内容的相似性来评估代码片段的相关性，忽略了代码的功能性和结构性意义。对于一些复杂的编程语言特性或模式，仅靠文本相似度难以准确捕捉其意图和功能，从而导致错误的上下文关联。同时，相比于代码切片方法，BM25不考虑代码的逻辑结构和控制流，当冲突涉及到深层逻辑或依赖关系时，BM25提供的上下文不够精确甚至误导，导致消解方案出现偏差。因此，在选择代码上下文获取策略时，需综合考虑各种方法的特点和适用场景。

综上，通过对四种代码上下文在冲突消解中的影响进行比较分析，发现代码上下文的消极影响与上下文的获取方式、信息相关性以及冗余程度密切相关。为了降低代码上下文的消极影响，可以尝试从以下方向去探索：(1)优化上下文检索策略：消极

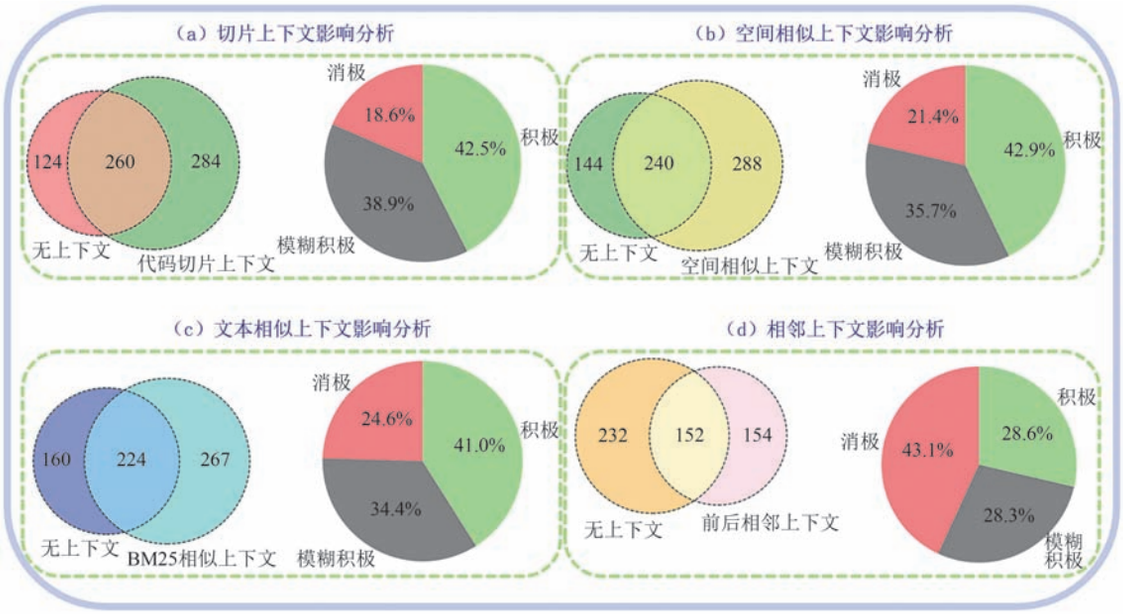

图6 各类型代码上下文对冲突消解的影响情况

影响的出现往往是因为上下文信息中包含了与冲突解决无关的冗余内容，这些内容干扰模型的理解和生成过程。(2)模型的上下文理解能力提升：当前的大型语言模型虽然在代码理解和生成方面表现出色，但在处理复杂的上下文信息时仍存在局限性。(3)上下文的增量式输入：当前的实验中，上下文信息是作为整体输入到模型中的。然而，这种一次性输入的方式导致模型在处理大量上下文信息时出现过载现象。因此，可以尝试采用增量式输入的方法，逐步向模型提供上下文信息，让模型在每一步中动态调整其对上下文的利用。(4)上下文的多源融合：在实际的冲突解决场景中，单一类型的上下文无法完全覆盖冲突的背景信息。因此，可以考虑将多种类型的上下文信息进行融合，以提高上下文的整体有效性。通过上述讨论，期望来降低代码上下文的消极影响，从而进一步提高合并冲突解决的成功率。

4.6 RQ4：代码上下文的跨模型适用性

为了评估代码上下文在大语言模型自动消解代码合并冲突任务中的通用有效性，本研究选取了多种大语言模型进行实验，具体模型的详细信息见表6，其中包括通用大语言模型和专门针对代码的语言模型(Qwen2.5-coder-32b-instruct[35])。

图7展示了在无上下文与不同代码上下文辅助下，多个大型语言模型在执行合并冲突消解任务时的表现。结果显示，除了前后缀上下文外，各类型代码上下文相较于无上下文输入，均显著提升了模型的冲突消解能力。具体而言，文本相似性上下文、空间相似性上下文及切片上下文的应用，平均提高了完美匹配数量17.1%、13.6%和21.2%。这表明，代码上下文不仅具有广泛的适用性，能够有效应用于不同的大型语言模型中，而且在自动解决代码合并冲突方面表现尤为出色，其中以切片上下文的效果最为显著。

表6 RQ4中用于评估的大语言模型

模型名称	厂商
Qwen2.5-32b-instruct[36]	阿里巴巴
Qwen2.5-coder-32b-instruct[35]	阿里巴巴
Hunyuan-large[37]	腾讯科技
Hunyuan-standard[37]	腾讯科技
DeepSeek V2.5[38]	深度求索
ERNIE-3.5-128K[39]	百度
GPT-3.5-turbo-1106[28]	Openai
GPT-4o-mini[28]	Openai
GPT-4o[28]	Openai

此外，通过观察图7的数据，发现在通义千问系列(如Qwen2.5-32b-instruct、Qwen2.5-coder-32b-instruct)中，当模型参数一致时，专门针对代码优化的语言模型相比通用型模型在性能上更为优越，并且在使用代码上下文辅助时提升幅度更大。这对于自动解决合并冲突的任务来说，利用高质量代码数据集进行微调的模型能带来更佳的表现。同时，对比国内外的大型语言模型，GPT系列(包括gpt-3.5-turbo、gpt-4omini、gpt-4o)在利用代码上下文增强任务执行效果方面表现出更强的能力，反映了其在理解用户指令方面的优势。这一结果进一步强调

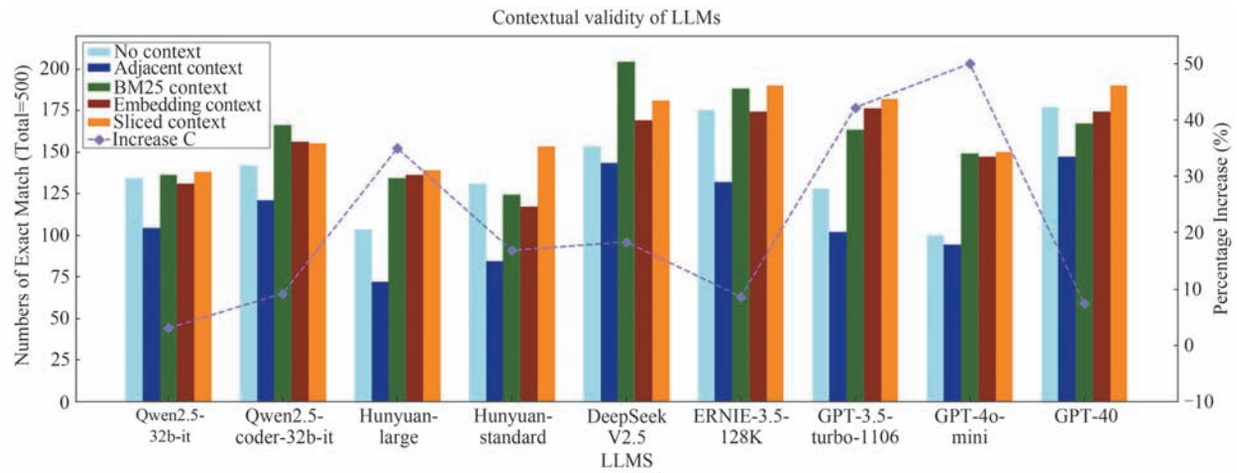

图7　切片上下文在各大语言模型的效果对比图

了模型架构和训练数据集对任务表现的重要性。

最后，从gpt-4omini与gpt-4o在合并冲突消解任务上的对比来看，较大规模的模型似乎拥有更强的表达能力，从而在该任务中表现更好。然而，在分析代码上下文的辅助效果时，注意到gpt-4omini的性能提升比gpt-4o更加明显，这是由于gpt-4o即使在没有上下文的情况下也具有较强的处理能力，因此未能完全体现出上下文带来的全部潜力。所以，较大规模的模型通常拥有更强的表达能力，但在某些情况下，较小规模的模型通过利用代码上下文也能实现显著提升，显示出它们在特定情境下的灵活性和适应性。综上所述，代码上下文的跨模型适用性不仅表明了其在提升大型语言模型处理合并冲突任务中的重要性，还展示了不同类型的代码上下文（如文本相似性、空间相似性和切片上下文）对多种模型的有效性。

5　讨　论

5.1　失败案例分析

为了深入理解ChatGPT在处理代码合并冲突时遇到的问题，本文识别并分析了四类典型错误模式，并基于此进行了研究：

（1）上下文错误引导：此类错误源于提供的代码上下文中包含了过多冗余信息或不相关的代码片段。这些额外的信息可能导致大语言模型在解析过程中产生误导，进而偏离正确的解决方案。

（2）上下文信息不足：当所提供的代码上下文不足以让大语言模型准确识别出解决冲突所需的关键信息时，就会出现这种类型的错误。在这种情况下，即使有上下文支持，大语言模型也无法做出最佳的决策，导致提出多种有可能但不确定的解决方案。

（3）大模型自身推理能力上限：这类错误表明，尽管提供了比较丰富的上下文信息，但对于某些特别复杂或独特的冲突案例，现有的大语言模型技术仍存在局限性，无法完全准确地解析并生成正确的解决方案。

（4）大模型输出中的细微瑕疵：虽然大模型能够生成接近正确的解决方案，但在输出格式上存在细微瑕疵，如多余的注释或其他非实质性差异。这些问题通常不会影响代码的功能实现，但在严格的评估标准下会被标记为“失败”。

为进一步探究上述问题，从基于静态分析的程序依赖代码切片上下文运行的数据中随机抽取了40个冲突消解失败案例进行了人工分析。通过细致分析每个失败案例，识别并分类了导致冲突消解失败的主要原因，结果如下：

如表7所示，首先，在所分析的40个失败案例中，上下文错误引导现象出现了3次。3个案例中都是由于某些与当前冲突块无关的代码行被包含在上下文中，导致大语言模型错误地将其视为解决冲突的关键因素。这种情况不仅增加了不必要的复杂性，还导致生成的解决方案不符合实际需求。

其次，上下文信息不足是出现频率最高的问题

表7　代码切片上下文中LLM失败案例统计

错误类别	数量/个数
上下文错误引导	3
上下文信息不足	21
大模型自身推理能力上限	8
大模型输出中的细微瑕疵	8

类型，在40个案例中有21例属于此类情况。在这21例失败案例中都是由于提供的上下文仅限于冲突所在文件内的代码段，而不是跨文件的相关代码段，不能涵盖所有相关依赖关系，但合并冲突块却涉及多个模块之间的交互。因此，大语言模型难以全面理解冲突的本质，从而提出了多种看似合理但实际上并非最优的选择。

第三，有8个案例显示出大模型自身推理能力上限的现象。这是由于案例中的合并冲突涉及深层次的数据结构变更或是高度抽象化的逻辑操作，即便是像ChatGPT这样先进的大语言模型也因推理能力的限制而无法正确处理。

最后，大模型输出中的细微瑕疵出现在8个案例中。其中大部分案例是由于大模型自动添加了结束符号（如}或)），虽然它们符合语法规范，但与真实的代码消解方案并不一致。尽管这些额外字符不影响程序运行，但它们的存在使得解决方案在形式上不符合完美匹配的标准。

综上所述，通过对40个冲突消解失败案例的详细分析，本文明确了导致失败的主要原因及其分布情况。本研究为进一步优化基于上下文的大语言模型冲突消解方法提供了实证依据，并探讨了一些可能的改进方向，包括但不限于：优化上下文选择策略，扩展模型的知识库，以及增强模型的推理能力等。未来工作将聚焦于如何有效减少上述各类错误的发生，以期提高自动化冲突消解的准确性和可靠性。

5.2 总结展望

本文使用ChatGPT探讨哪种代码上下文在解决代码合并冲突中具有最佳协助效果。结果如下：

首先，在自动消解合并冲突任务中，不同提示风格和温度设置对ChatGPT的完美匹配分数有显著影响。较低温度设置能产生更好、更稳定的结果，标准思维链提示风格有助于提高ChatGPT的性能。

其次，在行粒度的冲突消解中，提供代码上下文显著提升了消解效果，尤其是基于静态分析的程序依赖代码切片上下文表现最佳，与无上下文相比，正确率提升了42%。但在词元粒度任务中，BM25文本相似代码上下文效果最优，完美匹配个数和BLEU-4分数均优于其他方法。整体来看，行粒度的消解效果略优于词元粒度，但词元粒度方案的相似度更高。

最后，不同类型代码上下文对大模型解决合并冲突的影响各不相同。基于静态分析的程序依赖代码切片上下文对原任务的干扰最小，消极影响占比仅为18.6%；而基于前后相邻代码上下文的干扰最大，消极影响占比高达43%。综合考虑，研究表明基于静态分析的程序依赖代码切片上下文在协助冲突消解中表现最佳。此外，结果表明，代码上下文在不同大语言模型中具有通用的有效性。

本文研究提出的基于大语言模型结合多类型代码上下文的自动化代码合并冲突解决方法，为软件开发流程中的冲突消解提供了新的思路和有效手段。通过实验验证，基于静态分析的程序依赖代码切片上下文能够显著提高冲突解决的准确性和稳定性，可以将这一技术集成到现有的自动化合并工具中，以提高合并效率并减少人为错误。同时，研究表明，不同类型的代码上下文对冲突解决效果有显著影响，可以根据项目特性定制化上下文支持，优化模型性能。总的来说，本文的研究成果为软件开发中代码合并冲突的自动化解决提供了科学依据和应用价值，能够有效优化软件开发流程，提升开发效率，并推动软件质量的提升。

未来的研究应进一步探索更高效的代码上下文筛选与优化方法，以提升冲突消解的准确性和鲁棒性。同时，需要推动大语言模型在代码冲突消解领域的专业化发展，减少对外部提示的依赖，增强其自主性和适应性。此外，随着软件开发流程的不断演进，如何将自动化冲突消解技术与更广泛的开发工具链和生态系统深度融合，也将成为重要的研究方向。这些努力将为实现更智能、更高效的软件开发流程奠定基础，推动软件工程领域的持续创新与进步。

参 考 文 献

[1] Ghiotto G, Murta L, Barros M, et al. On the nature of merge conflicts: A study of 2,731 open source Java projects hosted by GitHub. IEEE Transactions on Software Engineering, 2018, 46(8): 892-915

[2] Zhang J, Mytkowicz T, Kaufman M, et al. Using pre-trained language models to resolve textual and semantic merge conflicts (experience paper)//Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. Virtual, Republic of Korea, 2022: 77-88

[3] Svyatkovskiy A, Fakhoury S, Ghorbani N, et al. Program merge conflict resolution via neural transformers//Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Singapore, 2022: 822-833

[4] Dong J, Zhu Q, Sun Z, et al. Merge Conflict Resolution: Classification or Generation?//2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). Luxembourg, 2023: 1652-1663

[5] Dinella E, Mytkowicz T, Svyatkovskiy A, et al. Deepmerge: Learning to merge programs. IEEE Transactions on Software Engineering, 2022, 49(4): 1599-1614

[6] Vinyals O, Fortunato M, Jaitly N. Pointer networks. Advances in neural information processing systems, 2015, 28

[7] Elias P, Junior H S C, Ogasawara E, et al. Towards accurate recommendations of merge conflicts resolution strategies. Information and Software Technology, 2023, 164: 107332

[8] Nelson N, Brindescu C, McKee S, et al. The life-cycle of merge conflicts: processes, barriers, and strategies. Empirical Software Engineering, 2019, 24: 2863-2906

[9] Vu T T, Do T D, Vo H D. Context-Encoded Code Change Representation for Automated Commit Message Generation. arXiv preprint arXiv:2306.14418, 2023

[10] Wu T, He S, Liu J, et al. A brief overview of ChatGPT: The history, status quo and potential future development. IEEE/CAA Journal of Automatica Sinica, 2023, 10(5): 1122-1136

[11] Brown T, Mann B, Ryder N, et al. Language models are few-shot learners. Advances in neural information processing systems, 2020, 33: 1877-1901

[12] Ji T, Chen L, Yi X, et al. Understanding merge conflicts and resolutions in Git rebases//2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE). Coimbra, Portugal, 2020: 70-80

[13] Bird C, Zimmermann T. Assessing the value of branches with what-if analysis//Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. Cary, USA, 2012: 1-11

[14] Ellis M, Nadi S, Dig D. Operation-based refactoring-aware merging: An empirical evaluation. IEEE Transactions on Software Engineering, 2022, 49(4): 2698-2721

[15] Han K, Xiao A, Wu E, et al. Transformer in transformer. Advances in neural information processing systems, 2021, 34: 15908-15919

[16] White J, Hays S, Fu Q, et al. ChatGPT Prompt Patterns for Improving Code Quality. Refactoring, Requirements Elicitation, and Software Design. arXiv e-prints, 2023

[17] Chen E, Huang R, Chen H S, et al. GPTutor: a ChatGPT-powered programming tool for code explanation//International Conference on Artificial Intelligence in Education. Cham: Springer Nature Switzerland, Tokyo, Japan, 2023: 321-327

[18] Liu C, Bao X, Zhang H, et al. Improving ChatGPT prompt for code generation. arXiv preprint arXiv:2305.08360, 2023

[19] Tian H, Lu W, Li T O, et al. Is ChatGPT the ultimate programming assistant--how far is it?. arXiv preprint arXiv: 2304.11938, 2023

[20] Ma W, Liu S, Wang W, et al. The scope of ChatGPT in software engineering: A thorough investigation. arXiv preprint arXiv:2305.12138, 2023

[21] Surameery N M S, Shakor M Y. Use chat gpt to solve programming bugs. International Journal of Information technology and Computer Engineering, 2023 (31): 17-22

[22] Khanna S, Kunal K, Pierce B C. A formal investigation of diff3//FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science: 27th International Conference, New Delhi, India, 2007: 485-496

[23] Weiser M. Program slicing. IEEE Transactions on software engineering, 1984 (4): 352-357

[24] Joern, https://docs.joern.io/quickstart/ 2024, 12,1

[25] Papineni K, Roukos S, Ward T, et al. Bleu: a method for automatic evaluation of machine translation//Proceedings of the 40th annual meeting of the Association for Computational Linguistics. Philadelphia, USA, 2002: 311-318

[26] Liu P, Yuan W, Fu J, et al. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. ACM Computing Surveys, 2023, 55(9): 1-35

[27] White J, Fu Q, Hays S, et al. A prompt pattern catalog to enhance prompt engineering with ChatGPT. arXiv preprint arXiv:2302.11382, 2023

[28] OpenAi API, https://openai.com/api/ 2024,12,1

[29] Fatih Kadir Akın. 2023. Awesome ChatGPT Prompts. https://github.com/f/awesome-ChatGPT-prompts

[30] Wei J, Wang X, Schuurmans D, et al. Chain-of-thought prompting elicits reasoning in large language models. Advances in neural information processing systems, 2022, 35: 24824-24837

[31] Gao A. Prompt engineering for large language models. Available at SSRN 4504303, 2023

[32] Białecki A, Muir R, Ingersoll G, et al. Apache lucene 4// SIGIR 2012 workshop on open source information retrieval. sn, Portland, USA, 2012: 17

[33] Robertson S, Zaragoza H. The probabilistic relevance framework: BM25 and beyond. Foundations and Trends® in Information Retrieval, 2009, 3(4): 333-389

[34] Feng Z, Guo D, Tang D, et al. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv: 2002.08155, 2020

[35] Hui B, Yang J, Cui Z, et al. Qwen2. 5-coder technical report. arXiv preprint arXiv:2409.12186, 2024

[36] Bai J, Bai S, Chu Y, et al. Qwen technical report. arXiv preprint arXiv:2309.16609, 2023

[37] Sun X, Chen Y, Huang Y, et al. Hunyuan-Large: An Open-Source MoE Model with 52 Billion Activated Parameters by Tencent. arXiv preprint arXiv:2411.02265, 2024

[38] Bi X, Chen D, Chen G, et al. DeepSeek llm: Scaling open-source language models with longtermism. arXiv preprint arXiv: 2401.02954, 2024

[39] ERNIE Bot API, https://qianfan. cloud. baidu. com/ modelbuilder 2024,12,1

DONG Zheng-Yu, M. S. His mainly research interests focus on artificial intelligence applications in software engineering.

ZHANG Wei-Feng, Ph. D., professor. His mainly research interests include code management and continuous integration.

XU Lei, Ph. D., associate professor. Her mainly research interests include software analysis and testing, and intelligent software engineering.

HE Hua, M. S. His mainly research interests focus on software engineering.

MEI Guang-Hui, M. S. His mainly research interests focus on software engineering.

XUAN Ji-Feng, Ph. D., professor. His mainly research interests focus on software engineering.

Background

This study addresses the issue of automatic resolution of code merge conflicts in version control systems. With the widespread adoption of tools like Git in collaborative development, branch-based parallel development has become the standard practice. However, code merge conflicts are a common challenge, particularly in large-scale projects, where conflicts may account for up to 50% of merges. Traditional manual conflict resolution is time-consuming and labor-intensive, making the development of efficient automated solutions a pressing need.

Internationally, various methods have been proposed to tackle the problem of code merge conflicts. Early research focused on speculative merging, line-based conflict resolution, and framing conflict resolution as a classification task. For example, DeepMerge employed pointer network-like structures to reorder code lines within conflict regions, resolving approximately 37% of conflicts. MergeBERT further refined this approach by reducing the conflict resolution granularity to the token level, significantly improving effectiveness. In recent years, advancements in large language models (LLMs) have opened new possibilities for addressing this problem. Zhang et al. pioneered the use of GPT-3 to process conflict regions by designing prompts to generate resolutions, demonstrating the feasibility of LLMs for this task. However, these methods are generally limited to the content within conflict regions, overlooking the context outside of the conflict blocks, which is critical for understanding the background of code changes and resolving conflicts effectively.

This paper proposes a ChatGPT-based method for automated conflict resolution, with a focus on the role of code context in resolving conflicts. Through comprehensive experiments, the study evaluates the impact of different prompt styles and temperature settings on ChatGPT's performance, as well as the effectiveness of four types of code context. The results show that providing relevant code context significantly improves conflict resolution. Among the tested contexts, static analysis-based program dependency context performed the best, increasing accuracy by 42%. In contrast, some contexts, such as those based on adjacent code, introduced significant interference, with negative impacts reaching up to 43%.

This study not only confirms the positive impact of code context on conflict resolution but also identifies the most effective type of context for enhancing LLM performance. The findings provide valuable insights into utilizing ChatGPT and other LLMs for automated code merge conflict resolution, while also shedding light on the general applicability of context information to other language models, laying a foundation for future research. This work was supported by the National Natural Science Foundation of China (62272214), Nanjing International Cooperation Project (202401006).