课堂练习

程智镝，汪天佑，陈凌，刘辉

2024 年 5 月 27 日

任务分配

- **作业一**：陈凌、刘辉
- **作业二**：汪天佑
- **作业三**：程智镝

1 作业一

传统代码评审与现代代码评审相比，在在实施方式、范围、技术应用以及所面临的挑战等多个维度上存在差异，并且各自拥有特定的优势与局限性。

1.1 共同点

目标一致：无论采用传统还是现代的代码评审方式，其主要目标都是提高软件质量。通过发现和修复代码中的错误，确保代码的正确性、可靠性和可维护性。两种评审方法都力求在软件交付前识别和解决潜在的问题，从而减少生产环境中的错误和漏洞。

参与角色的基本组成相同：在传统和现代代码评审中，基本的参与角色都是代码的作者和评审者。作者负责撰写和提交代码，而评审者则负责检查代码，提供反馈和建议。尽管在具体的角色分配和职责上可能有所不同，但两者都强调通过团队协作来提高代码质量。

评审过程中的反馈机制：两种评审方法都强调通过反馈机制来改进代码质量。在评审过程中，评审者会对代码的各个方面提出具体的改进意见，

作者根据这些反馈进行修改。这种循环往复的反馈机制有助于不断提升代码质量，确保最终产品的稳定性和性能。

遵循编码标准和最佳实践：无论是传统还是现代代码评审，都要求代码遵循一定的编码标准和最佳实践。通过评审过程，确保代码风格一致，逻辑清晰，并遵循团队或行业的规范。这有助于提高代码的可读性和可维护性，降低后期维护成本。

提高代码的可维护性和可扩展性：通过代码评审，可以提前发现设计缺陷和潜在问题，确保代码结构合理，便于后期的维护和扩展。两种评审方式都注重代码的可维护性和可扩展性，通过系统的检查和改进，使代码在未来的开发和维护过程中更加稳定和高效。

发现潜在的性能和安全问题：代码评审不仅关注代码的功能和逻辑正确性，还会检查代码的性能和安全性。通过评审，可以识别出可能影响性能的低效代码，以及可能带来安全漏洞的隐患。两种评审方法都旨在提高代码的整体质量，确保其在实际运行中的性能和安全性。

1.2 差异

规模与形式：

- **传统代码评审：**规模较大，常用于关键模块或大项目，通常涉及线性逐行审查和正式的面对面会议。
- **现代代码评审：**更为轻量级，频繁，适用于小规模代码改动，依靠在线工具（如GitHub Pull Requests, GitLab Merge Requests）进行评审。

技术应用：

- **传统代码评审：**较少依赖自动化工具，更多依赖人工审查。
- **现代代码评审：**广泛利用自动化工具来辅助检查语法错误、代码风格一致性和安全漏洞等。

标准与流程：

- **传统代码评审：**遵循严格定义的检查列表和标准，流程固定。
- **现代代码评审：**流程更加灵活，可以根据团队具体需求定制，评审标准动态调整，更加注重团队共识。

角色与互动：

- **传统代码评审**：评审者通常是资深开发人员或特定的代码审查员，角色较为固定。
- **现代代码评审**：鼓励更广泛的团队成员参与，包括新手和不同职能的开发者，促进知识分享和团队协作。

技术应用：

- **传统代码评审**：较少依赖自动化工具，更多依赖人工审查。
- **现代代码评审**：广泛利用自动化工具来辅助检查语法错误、代码风格一致性和安全漏洞等。

1.3 优点

传统代码评审：

- **深度和详尽**：传统代码评审通常进行非常详细和全面的检查。评审者会逐行阅读代码，确保每个部分都符合规范，并且能够发现潜在的深层次设计和逻辑问题。这样的深度检查有助于提高代码的整体质量，减少生产环境中的错误。
- **面对面交流**：传统代码评审通常通过面对面的会议进行，这有助于评审者和作者直接沟通，及时澄清疑问和讨论改进方案。面对面的交流方式可以提高沟通效率，避免误解，确保评审意见的准确传达和执行。
- **严格遵循标准**：由于传统代码评审遵循严格的流程和标准，能够确保代码在各个方面都符合预定义的规范。这种严格性有助于维持代码库的一致性和高质量，尤其在大型项目和关键模块中尤为重要。
- **全面性**：传统代码评审通常涉及大规模的代码检查，能够在一个完整的上下文中评估代码的质量和性能。这种全面性的检查有助于发现系统级的问题，而不仅仅是局部的错误。

现代代码评审：

- **灵活高效**：现代代码评审适应快速迭代的开发环境，能够在代码提交后立即进行评审和反馈。这样的即时反馈机制能够迅速发现和修复问题，减少开发周期中的延迟，提高开发效率。

- **知识共享**：现代代码评审鼓励团队内的广泛参与，不仅限于资深开发人员。通过让更多团队成员参与评审，能够促进知识共享，提升整体技术水平，并且新成员也能更快地熟悉代码库和项目规范。

- **自动化工具**：现代代码评审广泛利用自动化工具来辅助检查，如静态代码分析工具、持续集成工具等。这些工具能够自动发现语法错误、违反编码规范的问题以及潜在的安全漏洞，大大提高了评审的效率和准确性。

- **协作性强**：现代代码评审通常依赖在线平台进行，支持异步沟通和协作。团队成员可以在自己的时间对代码进行评审和讨论，这种灵活的协作方式有助于提高团队的整体工作效率。

- **易于追踪和管理**：通过使用版本控制系统和代码评审工具，现代代码评审能够记录每次评审的具体内容和历史变更。这些记录有助于追踪代码的修改过程，为日后的维护和审计提供了宝贵的参考资料。

1.4 缺点

传统代码评审：

- **耗时长、资源密集**：传统代码评审通常需要花费大量时间进行详细的检查，并且需要多名资深开发人员的参与。这种方式不仅耗时长，而且消耗大量人力资源，特别是在大规模项目中，评审过程可能成为瓶颈。

- **限制敏捷开发**：由于传统代码评审周期较长，通常不适应快速迭代的敏捷开发模式。在快速变化的项目环境中，长时间的评审过程会延缓反馈和改进，影响开发效率。

- **灵活性不足**：传统代码评审依赖固定的流程和标准，缺乏灵活性。对于一些新兴的开发方法和工具，传统评审方式可能无法及时适应和调整，从而限制了创新和改进的空间。

- **可能忽视团队协作**：传统代码评审往往由少数资深开发人员主导，可能忽视其他团队成员的参与和贡献。这种方式可能导致知识孤岛，限制团队内部的知识共享和协作。

现代代码评审：

- **缺乏深度**：由于现代代码评审通常处理较小规模的代码变更，评审过程可能不够深入。评审者可能更关注代码的表面问题，而忽略了深层次的设计和架构问题，这可能影响代码的长期可维护性和稳定性。
- **碎片化**：现代代码评审的高频率和小规模特性可能导致评审工作碎片化。评审者需要频繁切换角色和任务，这种碎片化的工作模式可能降低整体的工作效率和专注度。
- **质量参差不齐**：由于现代代码评审强调广泛参与，评审者的经验和技术水平可能参差不齐。这种多样性虽然有助于知识共享，但也可能导致评审质量不稳定，影响代码的整体质量。
- **依赖工具**：现代代码评审广泛使用自动化工具，虽然提高了效率，但过度依赖这些工具可能忽视复杂的逻辑问题和设计缺陷。此外，自动化工具的配置和维护也需要投入大量资源。
- **协调难度大**：在大规模项目中，现代代码评审涉及多个团队和角色的参与，需要有效的协调和沟通机制。管理和组织这样的大规模协作评审过程可能会面临较大的挑战，增加项目管理的复杂性。

2 作业二

收集代码评审面临的挑战和最佳实践代码评审（Code Review）是软件开发过程中确保代码质量、提高代码可维护性和团队知识共享的重要步骤。然而，代码评审也面临许多挑战。以下是一些常见的挑战以及最佳实践，以帮助团队克服这些挑战。

2.1 代码评审面临的挑战

2.1.1 时间限制

开发团队经常面临紧迫的项目截止日期，这种时间的紧迫可能来自开发者,由于没有把握好开发节奏,未能预留出足够的时间进行代码评审,也有可能来自reviewer,由于自身繁重的开发任务导致用于为他人评审代码的时间被无情压缩,导致代码评审时间不足,评审不充分。

2.1.2 评审标准不一致

不同的评审者可能有不同的评审标准和关注点，另外团队成员之间的经验和技能差距可能影响评审效果，尤其是新手可能不敢提出问题,评审者的代码水平与coding习惯也可能与开发者有着较大差距,导致评审质量不一致。

2.1.3 过度批评或人身攻击

评审过程中可能出现批评过度或带有个人色彩的评论，评审者不自觉地将评审的重心从代码质量偏移到开发者能力上, 影响团队士气和协作。

2.1.4 工具和流程不完善

缺乏合适的代码评审工具, 例如代码评审展示效果不佳,comments仅供展示,功能单一, 代码评审不支持上下文跳转功能, 这都会导致代码评审过程的效率低下。流程不完善，例如设置了两人评审的指标,但是在没完成该指标时就可以成功代码合并。甚至没有进行代码评审就成功将隐患代码上线。

2.2 最佳实践

首先针对上述的几项挑战,可以给出以下几个解决方案

- 时间限制：设定明确的代码评审时间窗口，确保代码评审是开发流程中的固定部分, 以此来保证开发者和reviewer都预留出足够的时间让代码经受代码评审。

- 评审标准不一致：通过培训和指导提升团队整体的代码评审能力，制定一定的多维度的评审标准, 例如”必须..”,”建议...”,”禁止...”等等, 鼓励新人参与并提出问题。
- 过渡批评或人身攻击：强调建设性的反馈，聚焦于代码问题而非个人，营造开放和尊重的评审氛围。
- 工具和流程不完善：使用专业的代码评审工具，如 GitHub PR、GitLab Merge Requests 或 Crucible，优化评审流程。结合内部的效能平台和办公平台, 使流程与办公,交流结合, 提高评审流程的效率和便捷性。

让我们走近某大厂的CodeReview来看看:

2.2.1 在流程上:

codeReview作为开发任务中的固定部分,其日期也会进入开发者和reviewer的日程中
在发起codeReview之前,内部的代码变更可视化系统会对变更代码进行分析,包括并不限于:静态扫描,单测以及代码覆盖率,用例通过率,从而为代码赋予风险级别,另外通过代码覆盖率锁定到没有覆盖的代码,并在代码评审中给予标记.
在codeReview中,打造舒适的review界面,高低风险代码的分层展示,打造上下文跳转的能力,将comment标记为待办.审批申请,评审详情,评审结果通过办公软件通知,提高流程的效率等等
在coderReview后,在评审中相关指标达到该项目预期指标后,即可发起合并.该指标包括且不限于:代码覆盖率,reviewer数量,评论数等

2.2.2 在角色上:

因为多reviewer的设立, 每个人都有机会参与到codeReview的过程中来.reviewer可以变换,由于在评审过程中可能发现变更代码是由另一位开发人员参与设计编码的,适时地切换reviewer可能让评审具有更好的效果

3 作业三

介绍至少3种提升代码评审的智能化技术。介绍需包括但不限于技术提出的背景、技术的理论基础和实现过程、技术的评估方式(数据集、i评估指标等)和评估结果等。介绍至少3种提升代码评审的智能化技术.介绍需包括但不限于技术提出的背景、技术的理论基础和实现过程、技术的评估方式(数据集、I评估指标等)和评估结果等. 从角色和过程两个视角切入。

3.1 自动代码审查工具

自动代码审查工具（Automated Code Review Tools）是通过静态分析、代码风格检查和性能优化建议等方式，自动检测代码中的潜在问题，从而减少人工审查的负担，提高代码质量和开发效率。

3.1.1 技术提出的背景

传统的代码审查依赖于人工，往往耗时且容易出现人为疏忽。尤其在大型项目中，代码量庞大，人工审查难以全面覆盖所有细节。为了提高审查效率并减少漏审，自动代码审查工具应运而生。

3.1.2 技术的理论基础和实现过程

自动代码审查工具主要基于静态代码分析（Static Code Analysis）技术，利用预定义的规则和模式来扫描代码，以检测潜在的错误和不一致。其实现过程一般包括以下几个步骤：

- **代码解析**：首先，将源代码解析成抽象语法树（Abstract Syntax Tree, AST），以便于后续的分析和处理。
- **规则匹配**：利用一组预定义的规则对AST进行匹配，以发现代码中的问题。这些规则可以涉及代码风格、潜在错误、安全漏洞等方面。
- **报告生成**：最后，生成审查报告，列出发现的问题和建议的修改方案，帮助开发者及时修正代码。

常见的自动代码审查工具包括SonarQube、ESLint和Pylint等。这些工具不仅支持多种编程语言，还提供可视化的报告和集成的开发环境（IDE）插件，以便于开发者在编码过程中实时获取审查反馈。

3.1.3 技术的评估方式

为了评估自动代码审查工具的有效性，通常采用以下方式：

- **数据集**：使用来自开源项目的代码库，如Apache、Mozilla等。这些项目代码量大且质量较高，适合作为评估基准。
- **评估指标**：包括检测到的问题数量、误报率（False Positive Rate）、漏报率（False Negative Rate）和用户满意度等。

3.1.4 评估结果

研究表明，自动代码审查工具在提高代码质量方面具有显著效果。例如，SonarQube在某些大型项目中能够检测出超过80%的潜在问题。以下是一些评估结果的具体数据：

- 在一个包含100万行代码的项目中，SonarQube检测到了5000个潜在问题，其中90%被确认是有效问题。
- ESLint在JavaScript项目中减少了约60%的代码风格不一致问题，大幅提升了代码可读性。
- Pylint在Python项目中的误报率控制在5%以内，漏报率则低于2%。

自动代码审查工具不仅提高了代码审查的效率，还显著提升了代码的质量和一致性，使得开发者能够专注于更高层次的设计和实现工作。

3.2 机器学习辅助代码审查

机器学习辅助代码审查（Machine Learning-assisted Code Review）通过学习历史代码和审查记录，提供智能化的代码审查建议，弥补了传统静态分析工具的局限性。

3.2.1 技术提出的背景

虽然静态分析工具在检测代码错误方面非常有效，但它们基于固定规则，难以动态适应新出现的代码模式和开发规范。随着机器学习技术的进步，研究人员开始探索利用机器学习算法来自动分析和审查代码，从而提供更智能和自适应的代码审查工具。

3.2.2 技术的理论基础和实现过程

机器学习辅助代码审查的实现过程包括以下几个步骤：

- **数据收集**：收集大量的代码片段和相应的审查记录。这些数据可以来自开源代码库、企业内部项目或在线代码托管平台（如GitHub）。
- **特征提取**：从代码中提取有用的特征，如代码结构、变量命名、注释内容等。这一步通常涉及代码的解析和分析。
- **模型训练**：使用提取的特征训练机器学习模型。常用的模型包括深度学习模型（如卷积神经网络、循环神经网络）和传统的机器学习算法（如支持向量机、随机森林）。
- **模型应用**：将训练好的模型应用于新代码，自动检测潜在问题并提供修复建议。

3.2.3 技术的评估方式

评估机器学习辅助代码审查工具的有效性通常采用以下方式：

- **数据集**：使用来自GitHub等平台的开源项目数据集。这些数据集通常包含丰富的代码实例和历史审查记录。
- **评估指标**：包括准确率（Accuracy）、召回率（Recall）、精确率（Precision）、F1分数（F1 Score）等。为了全面评估模型的性能，还可以考虑模型的计算效率和资源消耗。

3.2.4 评估结果

研究显示，机器学习模型在代码审查中的应用可以发现一些静态分析工具无法检测到的问题。例如，某些深度学习模型在测试中达到了80%以上的准确率和召回率。以下是具体的评估结果：

- 在一个包含50万行代码的项目中，深度学习模型检测到了3000个潜在问题，其中85%被确认是有效问题。
- 支持向量机模型在Java项目中的准确率达到了78%，召回率为75%。

- 结合历史审查记录的增强学习模型在Python项目中的F1分数超过了82%。

机器学习辅助代码审查工具能够显著提升代码审查的智能化水平，提高问题检测的准确性和全面性，使得开发团队可以更高效地发现和修复代码缺陷。

3.3 基于自然语言处理的代码审查

基于自然语言处理（Natural Language Processing, NLP）的代码审查技术通过理解和分析代码中的自然语言部分，如注释和文档，进一步提升代码审查的全面性和准确性。

3.3.1 技术提出的背景

代码审查不仅涉及代码本身，还包括对代码注释和文档的审查。注释和文档对于代码的可读性和维护性至关重要，但往往容易被忽视或写得不规范。传统的静态分析工具主要关注代码的逻辑和结构，难以有效处理自然语言部分。基于NLP的技术可以填补这一空白。

3.3.2 技术的理论基础和实现过程

基于NLP的代码审查技术的实现过程包括以下几个步骤：

- **文本预处理**：对代码中的注释和文档进行分词、词干提取、去除停用词等预处理操作，以便于后续分析。
- **语义分析**：利用NLP技术，如词向量（Word Embeddings）、词袋模型（Bag of Words）、TF-IDF（Term Frequency-Inverse Document Frequency）和深度学习模型（如BERT、GPT），理解注释和文档的语义。
- **一致性检查**：检查代码与其注释、文档的一致性，发现潜在的误导或不一致之处。例如，代码功能描述与实际实现不符，或者注释过于简略或冗长。
- **报告生成**：生成审查报告，列出发现的问题和改进建议，帮助开发者提高注释和文档的质量。

3.3.3 技术的评估方式

评估基于NLP的代码审查技术的有效性通常采用以下方式：

- **数据集**：使用包含详细注释和文档的开源代码库，如TensorFlow、Linux Kernel等。这些项目通常有较为规范的注释和文档，有助于评估NLP模型的效果。
- **评估指标**：包括语义一致性得分、注释质量得分、用户评价等。此外，还可以使用BLEU（Bilingual Evaluation Understudy）、ROUGE（Recall-Oriented Understudy for Gisting Evaluation）等NLP评估指标来衡量注释生成的质量。

3.3.4 评估结果

基于NLP的代码审查工具在提升注释和文档质量方面表现出色。例如，一些研究成果显示：

- 在一个包含20万行代码的项目中，NLP模型发现了500个注释与代码不一致的问题，其中90%被确认是有效问题。
- 使用BERT模型分析注释的一致性得分达到了85%以上，显著高于传统方法。
- 在用户评价方面，开发者普遍认为基于NLP的审查工具能帮助他们更好地理解代码，提高了开发效率和代码维护性。

基于NLP的代码审查技术能够有效提升代码注释和文档的质量，帮助开发者发现潜在问题，确保代码的可读性和可维护性。