arXiv:2601.01271v1 [cs.SE] 3 Jan 2026

亮点

CATCHALL：知识引导的大语言模型实现仓库感知异常处理

Qingxiao Tao、Xiaodong Gu、Hao Zhong、Beijun Shen

- 首个利用大语言模型实现自动化异常处理的仓库感知方法。
- 整合三个层面的异常处理知识，包括 API 级异常、仓库级执行上下文和跨代码仓库异常处理知识，以指导异常处理生成。
- 自动挖掘大规模真实世界代码仓库，构建高质量的异常处理知识。
- 在仓库感知异常处理任务中的性能优于最先进的方法。

CATCHALL：知识引导的大语言模型实现仓库感知异常处理

QingxiaoTao[a]、XiaodongGu[a]、HaoZhong[a] 和Beijun Shen[a]，*

[a]上海交通大学计算机科学学院，中国上海

文章信息

关键词：异常类型预测 异常处理 知识驱动生成 代码仓库理解

摘 要

异常处理是许多编程语言中至关重要的前向错误恢复机制，使开发人员能够通过结构化构造（例如 try-catch 块）管理运行时异常。异常处理不当或缺失往往会造成严重后果，包括系统崩溃和资源泄漏。尽管大语言模型（LLM）在代码生成方面展现出强大的能力，但由于复杂依赖和上下文约束，它们在仓库级异常处理方面仍面临困难。在这项工作中，我们提出了 CATCHALL，一种新颖的基于大语言模型的仓库感知异常处理方法。CATCHALL 为大语言模型提供了三个互补层次的异常处理知识：（1）*API* 级异常知识，由经验构建的 API–异常映射获得，用于刻画真实世界代码库中 API 的异常抛出行为；（2）仓库级执行上下文，通过对目标代码周围的上下文调用轨迹建模来捕获异常传播；（3）跨仓库处理知识，从跨项目历史代码中挖掘的可复用异常处理模式中提炼而来。这些知识被编码为结构化提示，以引导大语言模型生成准确且具备上下文感知能力的异常处理代码。为评估 CATCHALL，我们构建了两个用于仓库感知异常处理的新基准测试：大型数据集 RepoExEval 和可执行子集 RepoExEval-Exec。实验表明，CATCHALL 的表现始终优于最先进的基线方法，其 CodeBLEU 得分为 0.31（而最佳基线为 0.27%）、意图预测准确率为 60.1%（而最佳基线为 48.0%），Pass@1 为 29%（而最佳基线为 25%）。这些结果证实了 CATCHALL 在真实世界仓库级异常处理中的有效性。

1. 引言

异常处理是现代编程语言中的关键机制，是开发健壮、可靠的软件系统的基石。 [1]当程序遇到运行时异常条件时，相应的异常处理机制会触发预定义的恢复操作。这种设计将错误处理逻辑与常规程序流程隔离开来，从而增强软件的健壮性、可读性和可维护性。尽管异常处理至关重要，但开发人员仍经常误解、很少使用或错误实现异常处理。 [2, 3,4]多项研究 [5, 6] 表明，与异常相关的代码往往存在严重缺陷，包括实现不完整、测试覆盖率不足或完全缺失，可能导致系统崩溃、数据损坏和安全漏洞等严重后果。这些挑战迫切要求开发自动化技术，帮助开发人员编写正确且易于维护的异常处理代码。 [7]。

该领域的早期研究侧重于采用基于规则的技术检测易发生异常的位置，并合成catch 块。这些方法依赖人工定义的启发式规则，包括方法调用、 [8]，控制流路径 [9]，和 API 契约 [10]。尽管这类方法具有可解释性，但由于其僵化且泛化能力有限，难以应对罕见或特定于项目的异常。近期研究采用基于学习的技术 [11, 12]，以实现数据驱动、与特征无关的异常处理。这些技术利用源代码中的统计模式，但在捕捉细微的语义和上下文信息方面仍面临挑战，尤其是在处理复杂异常场景时。

大语言模型（LLM）的出现为自动化异常处理开辟了新的解决方向。由于大语言模型具备通用的代码理解与生成能力，它们无需依赖人工设计的启发式规则来合成代码，也不需要大规模标注数据集来训练领域专用模型。相反，研究人员采用知识提示链技术 [13] 和多智能体系统 [14] ，引导大语言模型生成、验证和改进异常处理代码。这一研究方向在不同代码库之间具有很强的泛化能力。

尽管大语言模型在简单代码任务上取得了令人瞩目的成果，但近期研究 [15]报告称，当任务需要代码仓库级上下文时，其有效性会大幅下降。现有大语言模型主要依赖内部参数化知识，往往缺乏对异常语义和仓库感知的程序行为的深入理解。在现代软件系统中，异常处理因间接 API 调用、异步回调、跨方法的异常传播以及其他复杂控制流结构而变得复杂；这些情况都需要对代码库进行长程依赖分析和整体推理。这些挑战凸显了掌握代码仓库级代码语义和异常机制的必要性。因此，自动获取仓库感知的异常

*通讯作者：Beijun Shen涛_qingxiao@sjtu.edu.cn （Q. Tao）；xiaodong.gu@sjtu.edu.cn （X. Gu）；zhonghao@sjtu.edu.cn （H. Zhong）；bjshen@sjtu.edu.cn （B. Shen）ORCID：

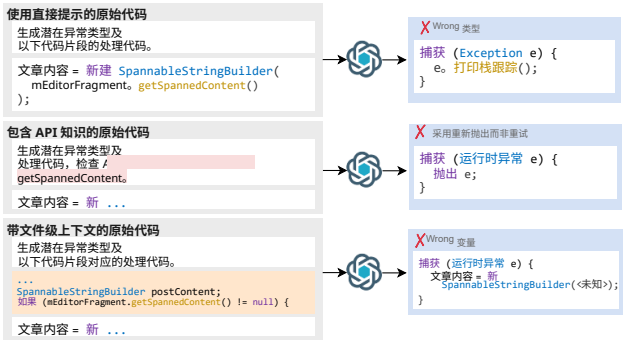

图 1： WordPress 安卓应用中异常处理的一个示例。异常源自调用链深处的`TextView.getText()`，并在导致应用故障之前通过回退逻辑进行处理。

知识对于生成准确且可行的异常处理建议至关重要。

我们通过一个示例说明现有基于大语言模型的生成方法的局限性。图1展示了安卓项目 WordPress [16] 中一个典型的异常处理场景。尽管该故障最终由调用链深处 `TextView.getText()` 抛出的 *运行时异常* 触发，现有方法却各有不足。直接提示方法忽略了异常语义，生成了无法恢复的处理代码。API 知识驱动的提示链方法 [13] 使模型能够识别正确的异常类型，但生成的处理逻辑退化为简单的重新抛出，无法实现有意义的恢复。相比之下，借助文件级上下文增强的方法 [12] 能够让模型推断出合理的处理意图，却无法将处理逻辑正确地关联到周边代码所需的具体变量和执行上下文。

这些局限性主要源于缺乏关键的代码仓库级知识，包括异常沿跨方法调用轨迹的来源、特定 API 关联的异常类型，以及现实世界代码库中相似异常的有效处理方式。为应对这些挑战，我们提出了 CATCHALL，一种新颖的基于大语言模型的方法，用于代码仓库感知的异常处理。CATCHALL 为大语言模型配备了三层异常处理知识：(1) *API* 级异常知识，通过实证挖掘的 API-异常映射刻画 API 的易发生异常行为；(2) 仓库级执行上下文，通过上下文调用轨迹捕获异常传播和控制流依赖；以及 (3) 跨仓库处理知识，从相似的历史代码示例中抽象出可复用的异常处理模式。
CATCHALL 完全自动化且可扩展，无需手动设计模板或规则，并使大语言模型能够生成同时明确了解 API 语义和特定代码仓库执行上下文的异常处理代码。

我们的工作贡献总结如下：

- 我们提出了 CATCHALL，这是首个面向自动化异常处理的仓库感知方法。该方法采用一种新颖的知识增强机制，整合 API 级异常知识、仓库级执行上下文，以及从大规模代码仓库中挖掘的跨仓库处理知识。
- 我们构建了 RepoExEval 和 RepoExEval-Exec，用于评测仓库感知异常处理。RepoExEval 包含从 GitHub 上 3,200 个开源安卓仓库收集的 12 万个异常处理代码块。RepoExEval-Exec 则包含来自四个具有代表性的代码仓库、可执行的代码；生成的 `try-catch` 块会被集成到原始代码库中，并通过单元测试进行验证，以评估功能正确性。
- 我们在 GPT-4o 和 DeepSeek-V3 两种大语言模型骨干上，将 CATCHALL 与六种具有代表性的方法进行全面评测，包括 Direct-Prompting、RepoCoder、ExAssist、Nexgen、Seeker 和 KPC。结果表明，CATCHALL 始终优于所有基线方法，在 CodeBLEU、意图预测准确率和 Pass@1 上分别提升 14.8~138.5%、25.2~230.2% 和 16~222.2%。

2. 相关工作

2.1. 异常处理的实证研究

异常处理已通过实证研究在多个计算领域得到了广泛研究，其中 Java 生态系统尤其受到关注。Cabral 和 Marques [17] 对 Java 与 .NET 的异常机制进行了比较分析。Sena 等 [18]研究了 Java 库中的异常模式。Asaduzzaman 等 [3] 调查了 Java 应用程序中的异常使用情况。ExChain [19] 分析了大型 Java 系统中的异常处理故障。针对特定领域的分析显著拓展了人们对这一主题的理解。Koopman 和 DeVale [20] 研究了操作系统中的异常处理，Chen 等 [21] 分析了云计算中的异常相关缺陷，而 Bruntink 等 [22]则聚焦于嵌入式系统。Coelho等 [23] 和 Fan 等 [24] 均通过大规模实证分析研究了安卓平台。此外，Cacho 等 [25]和 Anderson等 [26] 通过调查软件版本间异常处理的演变，作出了重要贡献。

这些实证研究共同揭示了异常处理在软件可靠性和可维护性方面的关键作用。通过考察异常使用模式、故障原因和现实世界中的实践，它们为异常处理技术的发展提供了基础性见解。

2.2. 自动化异常处理

作为提高软件可靠性的重要方法，自动化异常处理 [27]近年来受到广泛关注。这些方法可分为基于规则的方法和基于学习的方法。

基于规则的方法。基于规则的方法 [19,28] 利用静态分析、控制流检查和人工设计的启发式规则，检测易发生异常的位置并生成`catch` 块。Acharya 和 Xie [8]挖掘了应出现在`catch` 子句中的方法调用，Jana 等人 [29] 则通过更先进的静态分析扩展了他们的工作。Rahman [30] 推荐了用于处理异常的方法调用和详细代码片段。Jo等人 [9] 使用控制流分析识别未捕获的异常，而 Fetzer 等人 [10]提出为易发生异常的方法生成包装器。较新的基于规则的工作，例如 Nguyen 等人 [31, 32] 利用手工设计的代码特征预测是否应捕获异常；ExAssist [33] 则对挖掘出的历史模式应用模糊逻辑，以推荐异常类型和恢复 API。ExceRef [34]通过利用控制流和依赖分析自动重构异常处理结构，推进了这一方向，显著降低了代码风险并提高了重构效率。Zhang等人 [35]设计了一种扩展的控制流图和符号执行框架，用于检测 C++ 程序中的异常处理错误。

基于学习的方法。基于学习的方法 s [36, 37] 近年来，基于学习的方法变得相当流行且功能强大，因为它们不依赖特定特征，无需显式定义检测规则或合成模式，就能从历史代码样本中自动学习如何处理异常。Jia 等人 [38] 训练了一个模型，根据方法调用抛出异常的可能性对其进行排序；Xu 和 Zhong [39] 则使用分类模型识别异常类型与消息之间的不一致。ThEx [11]进一步推广了这一方法，通过学习上下文特征（例如，抛出位置和变量名）来预测新上下文中应抛出的异常。Nexgen [12]结合双编码器和程序切片，先定位合适的 `try` 块，再合成相应的 `catch`子句。CodeHunter [40] 采用基于 BERT 的模型和 Bi-LSTM 来定位易发生异常的代码，并以出色的实证性能预测异常类型。类似地，Neurex [41] 微调 CodeBERT，以通过多任务学习检测 `try-catch`边界、选择可捕获语句并推荐异常类型，同时从大型代码库中重复出现的异常处理模式中学习。

然而，缺乏领域特定异常知识的通用大语言模型难以有效应对复杂的现实异常场景 [13, 14]。为克服这一局限，KPC [13] 采用知识提示链，利用详细的 API 文档知识，迭代验证并优化异常处理代码。同时，Seeker [14] 开发了一种多智能体协作框架，以提供异常处理建议，并由精心整理的异常处理知识库提供支持。在这些进展的基础上，CATCHALL 通过整合三种知识来源实现了具备代码仓库感知能力的异常处理：即易引发异常的 API、上下文调用轨迹和异常处理模式，这些知识均通过挖掘大规模现实代码仓库提取而来。

3. 方法

本文提出 CATCHALL，这是一种新颖的、面向代码仓库的自动化异常处理方法。CATCHALL 通过整合三类关键知识，增强大语言模型对仓库上下文的感知能力：易引发异常的 API、执行上下文，以及从示例中提取的处理模式。

3.1. 问题定义

给定当前代码仓库中的一个代码片段，异常处理旨在预测其异常类型，并用 `try-catch` 块将其包裹，从而得到异常强化代码。增强后的代码必须：（1）准确捕获潜在的运行时异常；（2）与仓库上下文保持语义一致；（3）遵循当前项目的异常处理惯例。

3.2. 方法概述

图 2 展示了 CATCHALL 的总体流程。该流程包含五个主要步骤，模拟开发人员的推理过程。（1）首先，为了让大语言模型具备 API 知识，CATCHALL 从现有项目的 try-catch 块中提取易引发异常的 API，建立 API 与异常的映射（第 3.3 节）。（2）对于给定的代码片段，CATCHALL 分析其执行上下文，即一条具备代码仓库感知能力的调用轨迹，用以模拟异常在函数边界间的传播。这条轨迹包括从 try 块所在方法向上的调用层次结构、相关函数的定义、函数内部的 API 调用，以及基于导入的依赖（第 3.4 节）。（3）结合易引发异常的 API 和上下文调用轨迹，CATCHALL 预测代码片段可能抛出的异常类型（第 3.5 节）。（4）根据预测的异常类型和函数调用栈，CATCHALL 从现有项目中检索相关的历史 try-catch 块，并将其归纳为结构化的处理模式（第 3.6 节）。（5）最后，CATCHALL 实例化该处理模式，合成针对原始代码片段量身定制的上下文敏感异常处理代码（第 3.7 节）。

3.3. 挖掘易引发异常的 API

为增强大语言模型对常见 API 用法的了解，CATCHALL 会自动从真实世界代码库中提取易引发异常的 API，并通过实证方式对实际软件系统中 API 调用与其可能抛出的异常之间的关系进行建模。我们的方法首先自动爬取 GitHub 上的大量代码仓库，并使用 Tree-sitter[1],这一健壮且与编程语言无关的工具静态提取其中的`try-catch`代码块。

[1]https://tree-sitter.github.io/tree-sitter/

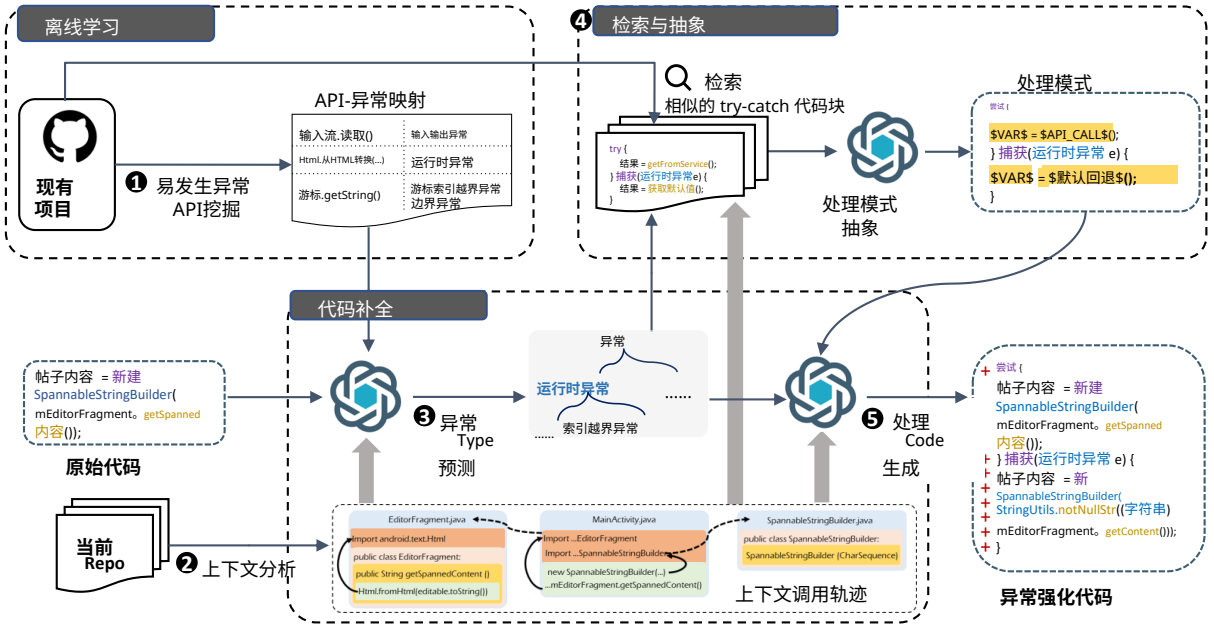

图 2： CATCHALL 概览。

解析器，可准确构建抽象语法树。对于每个try代码块，我们应用仓库感知上下文分析算法（算法1），以推导出完整的 API 使用轨迹。该轨迹不仅包含try 代码块内的 API，还涵盖其整个调用栈及周边方法中的 API。

随后，我们分析try 块及其扩展上下文中的所有 API 调用，并将其与相应catch块中捕获的异常类型关联起来。我们采用数据驱动的方法，根据实际代码中的经验性使用模式来定义易发生异常的 API。具体而言，凡是在 try-catch 块上下文中出现的方法调用，包括其完整的上游调用轨迹，均被指定为易发生异常。随后，将该上下文中捕获的具体异常类型映射到该 API，从而建立以代码仓库为依据的 API-异常关联。此映射全面涵盖标准库（例如 JDK/Android SDK）、第三方 API 以及用户定义的方法。给定代码仓库的覆盖范围由调用轨迹分析的深度动态决定，而该深度由算法 *d* 中的参数1控制。这确保生成的映射专门适用于正在分析的代码库。

此过程生成多对多 API-异常映射，其中每个 API 都关联多个异常类型，反之亦然。生成的映射会以结构化领域知识的形式存储，既有助于异常类型预测，也能自动生成异常处理代码。

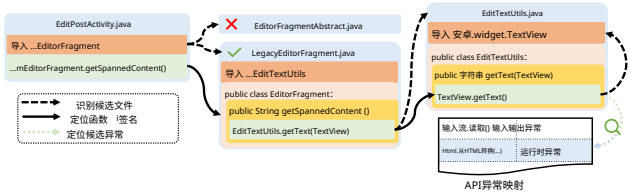

图 3： 一个跨多个文件的上下文调用轨迹示例。

3.4. 分析执行上下文

为更好地理解异常原因和处理策略，我们从原始代码片段中提取仓库感知的执行上下文。我们递归构建多层调用轨迹，以模拟异常如何在代码仓库上下文中的函数边界间传播。该轨迹覆盖多个文件和函数，反映导入依赖和跨文件调用如何影响与异常处理相关的运行时行为，如图 3 所示。分析具体考察：

- 导入：当前文件中声明的所有导入语句。

- 调用层次结构：通过沿调用者层次结构向上递归追踪 try 块所属的封闭函数，获取完整的调用栈。函数解析采用抽象语法树（AST）分析，并结合编辑距离进行模糊名称匹配，以匹配局部定义和导入符号。

算法1： 上下文调用轨迹构建

输入： 代码片段 T 位于代码仓库 R_T 中的文件 F
输出： 执行上下文 C_T

1 $C_T \leftarrow \emptyset$;
2 $C_T = C_T \cup F.import$ // 从中提取所有导入语句 F 并存储在 C_T 中;
3 查找 T 在 F 中所属的外围类和函数，并将它们的名称记录在 C_T 中;
4 提取 T 中的所有 API 调用语句，并初始化一个栈 S;
5 **当** S 不为空且深度 $\leq d$ **时执行**
6 f = 弹出(S)；// 从栈中弹出一个函数调用
7 使用导入名称在 R_T 中识别候选文件来自 C_T;
8 从这些文件中定位 f 的定义;
9 **对于每个**定义 d 的 f **执行**
10 提取 d中的所有函数调用;
11 将 d 及其调用追加到 C_T;
12 将新的函数调用压入 S;
13 **返回** C_T;

- 函数签名：调用栈中所有函数的签名信息（例如函数名称、参数）。

- 过滤后的函数体：调用层级中的函数体，其中仅保留调用语句（即函数调用）；非调用代码会被丢弃。

- 词法作用域：外围类名（如适用）以及包含`try`块的直接函数名。

算法1 概述了构建执行上下文的过程。给定当前代码仓库 R_T 中文件 F 内的代码片段 T，算法首先初始化一个空的上下文对象（第 1 行），并从F中提取静态结构属性，包括导入依赖、外围类名和方法名（第 2–3 行）。随后，它识别 T 中的所有 API 调用语句，并初始化一个调用栈 S，以便进行代码仓库级分析（第 4 行）。算法的核心是在函数调用图上进行深度受限遍历（最大深度为 d）。每次迭代中，都会从栈 S 中弹出一个函数调用 f（第 6 行）。算法根据 C_T 中收集的导入名称，在代码仓库 R_T 中识别候选文件，并分析这些文件以找到实际的函数定义，从而定位 f 的函数签名（第 7–8 行）。对于每个解析出的定义 d，CATCHALL 会提取内部函数调用（第 10 行），将该定义及其被调用函数追加到上下文 C_T中（第 11 行），并将新的函数调用压入栈 S（第 12 行）。遍历会持续进行，直到栈为空或达到最大调用深度。

提取的上下文调用轨迹通过利用导入关系解析跨文件的函数调用，模拟代码仓库级别的执行行为。这使得系统能够捕获整个项目中具有代表性的 API 使用模式，从而增强语言模型在真实世界代码仓库中进行异常处理的泛化能力。

3.5. 异常类型预测

利用收集到的易引发异常的 API 和执行上下文，我们为输入代码片段预测异常类型。我们采用标准的检索增强生成（RAG）框架进行异常类型预测。该 RAG 框架包含两个阶段——检索阶段和生成阶段。

在检索阶段，CATCHALL 首先提取代码片段调用轨迹中出现的所有 API 调用。随后，将这些 API 签名用作搜索查询，在预先构建的 API–异常映射数据库中进行检索，从而获取已识别 API 在执行期间可能抛出的所有潜在异常类型。

在生成阶段，CATCHALL 使用大语言模型，通过零样本提示预测最有可能的前 k 个异常。该提示包含：（1）代码片段；（2）其上下文调用轨迹；（3）候选异常集合。此预测提示旨在引导模型根据异常与给定代码上下文的相关性准确地对其进行排序。通过将预测范围限制在候选集合内，我们的方法兼顾了预测准确性和覆盖率，进而引导模型生成更精准、具备上下文感知能力的异常处理代码。

异常类型预测提示词

系统提示词：
你是 Java 异常处理方面的专家。给定一个代码片段、一组可能的异常类型以及调用轨迹，你的任务是从候选项中选出最合适的异常类型。仅输出一个格式正确的 catch 块，其中包含选定的异常类型以及用于异常处理逻辑的占位符。

用户提示词：
// 异常类型候选项:
$exceptions_list$
// 带有周边上下文的代码片段:
$code_snippet$
/* 根据提供的代码上下文和函数调用 , fill 在 <mask0> token 中填入一个从上述候选项中选出的合适异常类型。输出必须严格遵循以下格式：*/
catch (<掩码0> <掩码1>) {
<mask2>
}

3.6. 检索和抽象先前的处理样例

为了生成稳健且符合惯例的异常处理代码，我们通过先前处理样例中提取的示例知识来增强大语言模型。

3.6.1. 检索相似的 *try-catch* 块

给定当前代码仓库中的代码片段、其上下文调用轨迹以及预测的异常类型，CATCHALL 会从现有代码库中检索相关的历史 `try-catch` 代码块。我们从三个方面定义异常处理代码的相似度指标：

• 异常类型相似度：根据预测的异常类型与历史样本在异常继承树中的类型之间的层级距离计算。

• *API* 序列相似度：通过计算原始代码片段与历史代码片段中所用 API 集合之间的 Jaccard 相似度得出。

• *try* 块相似度：使用 CodeBLEU 衡量 [42]，以捕捉结构和语义上的接近程度。

这三项分数通过加权平均进行汇总。CATCHALL 随后从现有项目中检索最相似的前K 个`try-catch` 块。我们根据研究问题4（第$K = 20$ 5.4节）中的超参数优化结果，凭经验设定。

3.6.2. 抽象处理模式

先前研究发现，有限的代码示例可能会限制模型的泛化能力 [43]。为解决这一局限，我们从类似的历史代码中提炼可复用的异常处理模式。提取出的模式与代码片段的预测行为及执行上下文相一致。我们采用一种由大语言模型驱动的方法，通过结构化的多阶段抽象流程，从检索到的前K 个`try-catch` 块中提取可复用的异常处理模式：

1. 结构特征提取：将每个`catch`块分解为各个组成元素，包括异常类型、日志记录机制、恢复过程、清理操作、返回值处理和副作用。2. 模式识别：分析样本，发现反复出现的做法，并分离可变组件。3. 模式合成：使用参数化占位符（例如，`${{LogMethod}}`、`${{RecoveryAction}}`）生成通用模式。4. 模式形式化：以语法有效的代码格式生成最终模式，确保其可用于后续异常处理代码生成。

用于抽象异常处理模式的提示

系统提示：
你是一位经验丰富的 Java 异常处理专家。给定一组 try-catch 块，你的目标是分析并抽象出一种通用异常处理模式，以概括各示例共有的结构和差异。

用户提示：
请按照以下步骤进行推理：
步骤 1：提取特征
对于每个 catch 块，在表格中提取并总结 e：
- try 块抽象语法树
- 异常类型
- 日志记录方法
- 恢复或回退逻辑
- .....

Step 2：识别 y 模式
确定这些代码块中的共享组件和不同部分。

步骤 3：综合模式
使用占位符设计一个统一模板，例如 VAR，LOG_METHOD、$DEFAULT_FALLBACK$ 等。

步骤 4：形式化模式
示例格式：

```
try {$VAR$ = $API_CALL$(); }
catch (运行时异常 e) { $LOG_
METHOD$(e); $VAR$ = $DEFAULT_
FALLBACK$(); } 输入代码示例：$
try-catch_blocks$
```

3.7. 生成异常处理代码

CATCHALL 将抽象处理模式实例化，生成针对原始代码片段定制的上下文感知的异常处理代码。生成提示纳入了上下文调用轨迹（包括相关的 API 调用序列和周围的代码片段）、预测的异常类型，以及从类似历史案例中抽象出的泛化处理模式。在实例化过程中，CATCHALL 指示语言模型通过将占位符替换为上下文感知的内容，在模式中生成一个 `catch` 块，从而确保符合异常处理惯例和项目特定的语义约束。

生成异常处理代码的提示

系统提示：
你是一位精通 Java 异常处理的专家，负责生成健壮且能感知上下文的 Java catch 块。给定一个包含异常类型的 try 块、其上下文调用轨迹以及处理模式，你的任务是填入该模式，并仅输出能够有意义地处理该异常的适当 catch 块。

用户提示：
// 填写 catch 块时遵循的参考异常处理模式：$catch_块_模式$// 相关 API 调用序列$api_调用_序列$// 来自代码库的相关代码片段$相关_代码_片段$/* 根据 API 调用序列和代码上下文，并遵循上面的模式，填入上述异常处理模式以补全以下代码： */ try {$try_块_代码$}catch ($异常_类型$ e) {

4. 实验设置

我们开展实验以评估 CATCHALL 的有效性，旨在回答以下研究问题。

• **RQ1**：CATCHALL 如何提升当前最先进水平?

• **RQ2**：预测的异常类型有多准确?

- **研究问题3**：各个组成部分分别会产生什么影响?

- **研究问题4**：检索历史样本数量会产生什么影响?

4.1. 对比方法

我们将 CATCHALL 与四类异常处理代码生成方法进行比较：基于规则的补全方法（ExAssist）、知识增强推理方法（KPC）、多智能体协作方法（Seeker）和神经机器翻译方法（Nexgen）。此外，我们还将一种最先进的通用仓库级代码生成方法（RepoCoder）改造用于异常处理。我们也将一种不使用仓库上下文的直接基于大语言模型的生成方法作为基线方法。

具体而言，我们将 CATCHALL 与以下基线方法进行比较：

- **ExAssist** [33]：一种基于规则的方法，从数千个高质量程序中学习模糊逻辑规则，以预测潜在的运行时异常，随后生成`try-catch` 代码块来处理异常并从异常中恢复。

- **Nexgen** [12]：一种基于神经机器翻译（NMT）的方法，采用编码器-解码器架构来定位易发生异常的代码并生成完整的`catch` 块。该方法使用从 GitHub 获取的大规模 Java 方法数据集进行训练。为确保评估中的公平比较，我们向模型提供`try` 块的真实位置，并仅要求其生成`catch`块。

- **KPC** [13]：一种知识驱动的提示链方法，利用 API 文档中的细粒度异常处理知识，通过有针对性的异常处理提示迭代验证并重写生成的代码，直至妥善处理所有异常。

- **Seeker** [14]：一种多智能体协作框架，协调五个专门的智能体（规划器、检测器、捕获器、排序器和处理器），以模拟专家推理、协作推断异常处理意图，并生成符合上下文的处理代码。

- **RepoCoder** [44]：一种通用的仓库级代码补全方法。为使其适用于我们的任务，我们将输入代码置于`try` 块中，并将`catch`块留空，提示 RepoCoder 推断异常类型并生成相应的处理逻辑。为确保公平比较，我们还实现了 RepoCoder-RG1，即 RepoCoder 的单轮变体，该变体禁用其迭代式优化机制，与 CATCHALL 的单轮 RAG 策略保持一致。

- **直接提示**：一种基于大语言模型的方法，可根据独立代码片段生成异常处理代码，不依赖任何仓库特定数据或上下文。

所有基线方法均使用其官方发布的代码实现。对于*RepoCoder*、*KPC*和*Seeker*等基于大语言模型的方法，我们按照各自的任务设置和指令，将相同的输入代码片段嵌入每种方法原有的提示构造流程，以确保公平比较。因此，每种方法都获得相同的功能上下文，同时保留各自的提示设计。此外，我们将其原有的骨干模型替换为 GPT-4o[2] 或 DeepSeek-V3[3]，以统一模型能力。

4.2. 评估指标

我们采用三个广泛使用的指标来衡量异常处理的性能：

Pass@1 [45], 通过在原生项目环境中运行相关单元测试，评估合成代码的功能正确性。每个生成的 try-catch 块都会集成到其原始项目中，并在原生环境下运行。如果生成的处理器使专门用于触发目标异常的测试用例通过，则认为该样本正确。形式化定义如下：

$$\text{Pass@1} = \frac{\text{\# of correctly executed handlers}}{\text{\# of total testable cases}} \tag{1}$$

CodeBLEU [42],这是一种专为代码生成设计的指标，结合了加权 n-gram 匹配、语法树相似度和语义数据流对齐，相较于普通 BLEU，能够更敏锐地评估结构和正确性。

IntentAcc（意图预测准确率）[46], 用于衡量预测的处理意图（例如日志记录、重试、错误恢复、返回、重新抛出）与真实标注的匹配准确度。IntentAcc 的计算方式如下：

$$\text{IntentAcc} = \frac{1}{N}\sum_{i=1}^{N}\frac{|Y_i \cap \hat{Y}_i|}{|Y_i \cup \hat{Y}_i|} \tag{2}$$

其中，N 是异常处理案例的数量，Y_i和 $\hat{Y}_i$ 分别表示真实标注意图集和预测意图集。

4.3. 数据集构建

由于缺乏公开可用的仓库级异常处理数据集，我们基于真实世界的安卓项目创建了两个全新的基准测试：Re- poExEval 和 RepoExEval-Exec。RepoExEval 是一个大规模基准测试，收集自 3,200 个代码仓库，旨在使用 CodeBLEU 和意图准确率指标评估生成的异常处理

[2] https://platform.openai.com/docs [3] https://api.deepseek.com

代码。由于缺少单元测试且依赖项繁重，无法进行大规模的基于执行的评估。为此，我们从四个代码仓库中创建了一个紧凑的可执行基准 RepoExEval-Exec，用于使用 . 评估功能正确性。

数据集构建过程包括以下步骤：

(1) 代码仓库收集与筛选。我们首先在 GitHub 上搜索同时将“Android”和“Java”作为主要关键词的代码仓库，从而涵盖广泛的应用领域和项目结构。为确保选取的项目质量高、真实可靠且在时间上有效，我们依次应用一系列筛选条件：

- 代码质量指标：我们优先选择星标数量较多的代码仓库，以此作为代码质量、社区认可度和积极维护情况的衡量指标。
- 真实性验证：我们将每个代码仓库与 F-Droid 开源目录进行交叉核验，以确认其真实性，并排除已弃用、质量低劣或非真实的安卓项目。
- 时间有效性：为避免大语言模型预训练语料库造成数据污染，我们排除在严格截止日期之后更新的项目（GPT-4o 的截止日期为 2023年9月，DeepSeek-V3 的截止日期为 2024年6月）。

经过筛选，我们得到一个精心整理的数据集，其中包含 3,200 个高质量、积极维护的安卓代码仓库，在领域代表性与尽量减少过时或低质量项目的纳入之间取得了平衡。我们从这些代码仓库中提取了 120,000 个异常处理代码块，每个代码块均包含`try` 和 `catch` 代码块、引发的异常类型、相关调用轨迹以及上下文元数据（例如外围类、方法、文件和代码仓库）。为便于进行代码仓库感知推理，我们还使用算法 1 构建了过程间和代码仓库级上下文。

（2）*RepoExEval* 构建。我们从完整数据集中随机抽取 1,000 个异常处理实例，构建用于评估的基准测试集。其余样本则作为参考语料库保留，用于支持我们的方法提取 API–异常映射并检索历史案例。基准测试集中的所有异常处理意图（例如日志记录、重试、恢复、返回或抛出）均由人工标注，以确保标签准确。对于训练集和验证集，意图标签通过基于规则的启发式方法生成，随后再由人工核验，以保证质量。

(3)*RepoExEval-Exec* 构建。为评估生成的异常处理代码在功能上的正确性，我们构建了 RepoExEval-Exec，这是一个由四个项目组成的小规模可执行基准：*Aria2App*、*openScale*、*Overchan-Android* 和 *Signal-Android*。我们从这些项目中选取了 100 个真实的异常处理实例，每个实例都配有能够触发相应异常的单元测试。生成的 catch 块将被集成到原始代码库中，并通过这些测试执行，以评估运行时正确性和行为适当性。

表 1 RepoExEval 和 RepoExEval-Exec 基准测试的统计数据

指标	RepoExEval	RepoExEval-Exec
代码仓库数	3,200	4
Try-Catch 实例数	120,000	100
# 单元测试	N/A	174
平均调用轨迹深度	7.2	7.7
平均跨文件跨度	5.7	6.1
平均代码仓库规模（Java 类数）	249	665

表2仓库感知的异常处理方法在 RepoExEval 上的性能

模型	方法	Pass@1	CodeBLEU	意图准确率
-	ExAssist	9%	0.13	33.9%
	Nexgen	14%	0.25	37.6%
GPT-4o	直接提示	13%	0.24	46.8%
	RepoCoder RG1	21%	0.27	36.9%
	RepoCoder	25%	0.27	48.0%
	KPC	9%	0.13	18.2%
	Seeker	15%	0.23	43.3%
	CatchAll（我们的方法）	**29%**	**0.31**	**60.1%**
DeepSeek-V3	直接提示	10%	0.24	43.4%
	RepoCoder RG1	21%	0.26	38.1%
	RepoCoder	23%	0.27	47.3%
	KPC	9%	0.13	18.2%
	Seeker	15%	0.24	43.8%
	CatchAll（我们的方法）	26%	**0.29**	**53.9%**

表1总结了两个基准的统计数据。在 RepoExEval 中，平均每个项目包含 249 个 Java 类，每个异常处理实例的平均调用轨迹深度为 7.2，平均涉及 5.7 个不同的文件。RepoExEval-Exec 的复杂度更高，平均每个项目包含 665 个类，调用轨迹深度平均为 7.7，跨文件跨度平均为 6.1。

5. 结果与分析

5.1. 总体性能（RQ1）

设置。我们使用大规模 RepoExEval 基准和精简、可执行的 RepoExEval-Exec，对比 CATCHALL 与多种自动化异常处理基线方法。对比对象包括专有的（GPT-4o）和开源的（DeepSeek-V3）最先进基础模型。

结果。如表2所示，在两种大语言模型配置下，CATCHALL 在异常处理代码生成方面均优于所有基线方法，并取得最高的 CodeBLEU 和 Pass@1 分数。具体而言，使用 GPT-4o 时，CATCHALL 的 CodeBLEU 得分为 0.31，Pass@1 为 29%，分别比表现最强的大语言模型基线方法 RepoCoder（CodeBLEU 为 0.27，Pass@1 为 25%）高出 14.8% 和 4 个百分点。使用 DeepSeek-V3 时也观察到类似优势：CATCHALL 的 Pass@1 为 26%，优于 RepoCoder（23%）和

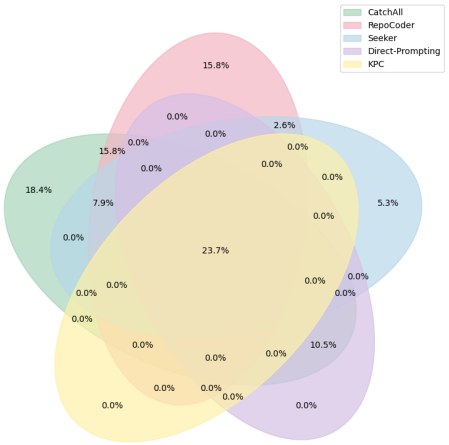

图 4： GPT-4o 下不同方法正确预测结果的维恩图。每个区域表示由特定方法组合正确处理的测试实例所占的比例。

直接提示（10%）分别高出 3 个和 16 个百分点，表明其设计能够有效泛化至专有模型之外。在意图预测（意图准确率）方面，CATCHALL 在 GPT-4o（60.1%）和 DeepSeek-V3（53.9%）下同样领先，且显著优于所有其他方法。

基线方法存在明显局限。ExAssist 依赖静态的基于规则的 API–异常映射，并忽略仓库上下文，因此准确率较低（Pass@1 为 9%）。KPC 仅重新抛出捕获到的异常，缺乏策略多样性。Nexgen 使用局部上下文，但缺失代码仓库级信息，Pass@1 仅达到 14%。直接提示会生成缺乏特定于异常的知识支撑的通用代码，而 Seeker 的多智能体协作仅带来边际提升（15%）。RepoCoder 通过利用代码仓库范围内的上下文，实现了最强的基线方法性能，但由于缺乏显式的异常感知推理，其性能提升受到限制。采用更简单检索策略的 RepoCoder-RG1 表现更差，凸显了精准上下文检索的重要性。

为进一步考察性能差异，图4展示了 GPT-4o 下各代表性方法正确处理实例的重叠情况。尽管所有方法都能解决一个不可忽视的共同案例子集，但 CATCHALL 覆盖了数量最多的独有正确实例，这表明其改进并非仅仅源于解决简单或普遍受到关注的示例。具体而言，有相当数量的案例只有 CATCHALL 能够正确处理，而其他强基线方法（如 RepoCoder 和 Seeker）均未能解决。这表明，CATCHALL 能够处理仅凭通用提示或单独进行代码仓库级检索无法解决的异常处理场景。相比之下，各基线方法彼此之间存在大量重叠，反映出它们在有效处理策略方面的多样性有限。

表 3使用 GPT-4o 模型进行异常类型预测的准确率

方法	类型准确率	父类型准确率	子类型准确率
直接提示	39.2%	40.4%	55.1%
RepoCoder RG1	29.0%	29.5%	37.7%
RepoCoder	40.2%	40.3%	49.7%
ExAssist	27.2%	31.2%	35.2%
Nexgen	27.8%	32.0%	30.0 %
KPC	41.1%	44.3%	59.7%
Seeker	34.8%	36.0%	54.8%
CatchAll（我们的方法）	**53.6%**	**54.5%**	**67.6%**

这些覆盖层面的差异为 CATCHALL 得益于互补知识来源而非单一主导因素提供了确切证据。这些稳定的提升凸显了 CATCHALL 中三个互补知识渠道的有效性：(1) 捕获仓库感知执行上下文的上下文调用轨迹；(2) 缩小搜索空间的易引发异常的 API；(3) 从真实 try-catch 示例中抽象出的处理模式，为生成提供指导。消融研究（研究问题3）进一步验证了各组件的贡献。

RQ1 的回答： CATCHALL 在仓库感知异常处理方面始终优于所有基线方法；与表现最强的基线方法 RepoCoder 相比，它在 CodeBLEU、意图预测准确率和 Pass@1 上分别高出 14.8%、25.2% 和 4%（GPT-4o）/3%（DeepSeek-V3）。

5.2. 类型预测准确率（RQ2）

设置。 我们进一步在 RepoEx-Eval 基准测试中评估 CATCHALL 和其他基线方法的异常类型预测准确率。考虑到 Java 的异常继承层次结构，我们采用三项指标，在不同粒度级别上衡量预测性能：(1) 类型准确率 评估预测异常类型与真实异常类型是否完全匹配；(2)父类型准确率 接受与异常层次结构中任一超类匹配的预测；(3)子类型准确率 则将与更具体的后代异常匹配视为有效预测。鉴于 CATCHALL 在 RQ1 中表现出色，本次评估采用 GPT-4o 作为基础模型。

结果。 表 3 展示了我们的方法与所有基线方法在异常类型预测方面的性能对比。总体而言，CATCHALL 在所有指标上均持续取得最佳结果。具体来说，CATCHALL 的类型准确率达到 53.6%，比表现最佳的基线方法 KPC（41.1%）高出 12.5%，在父类型准确率和子类型准确率方面也分别提升了 10.2% 和 7.9%。

现有方法通常存在两个关键局限：对异常类型的了解不足，以及缺乏仓库级上下文感知。例如，KPC 将关注范围限制在本地代码块中的 API，并高度依赖基础模型内置的 API 知识，因此覆盖范围有限。Seeker 纳入了来自文档和基础

表 4GPT-4o 模型的消融研究结果

变体	Pass@1	CodeBLEU	意图准确率
CatchAll（我们的方法）	29%	0.31	60.1%
无上下文调用轨迹	23% (↓ 20.7%)	0.29 (↓ 6.5%)	57.1% (↓ 5.0%)
无 API-异常映射	9% (↓ 67.0%)	0.25 (↓ 19.4%)	48.3% (↓ 19.6%)
无异常处理模式	21% （↓ 27.6%）	0.28 (↓ 9.7%)	48.8% (↓ 18.8%)

模型的某些先验知识，但它缺乏对真实世界异常分布的了解，也未能考虑更广泛的调用上下文。直接提示完全依赖基础模型的通用先验知识，未整合任何特定于代码仓库的信号。值得注意的是，Nexgen 和 ExAssist 的表现均不及 CATCHALL，类型准确率分别绝对低了 25.8 和 26.4 个百分点。性能差距源于：Nexgen 从真实世界代码仓库中学习异常知识，却忽略了长距离上下文依赖；而 ExAssist 则通过从真实项目中提炼的手工编写规则来编码异常知识，但仍未考虑上下文。

相比之下，CATCHALL 利用大规模的 API-异常映射来建模统计关联，这对于在单个代码仓库中表示稀疏的 API 尤为有用。此外，通过纳入上下文调用轨迹，我们的方法整合了代码仓库范围内的上下文，这对于准确预测异常类型至关重要，因为许多异常都源自跨文件的深层 API 调用。这种代码仓库感知推理提升了整体预测准确率，也解释了子类型准确率始终较高的表现。即使无法确定确切的异常类型，CATCHALL 也常能识别语义相关的后代类型，展现出先进的类型推断能力。

对研究问题2的回答： CATCHALL 显著提高了异常类型预测准确率，类型准确率达到 53.6%，比表现最佳的基线方法绝对提升了 12 个百分点。

5.3. 知识组件的贡献（研究问题3）

设置。 我们开展消融研究，以考察三个核心知识组件各自的贡献，包括上下文调用轨迹、API-异常映射和处理模式。在本研究中，我们每次移除一个组件，同时保持其他组件不变，然后衡量由此产生的消融变体对性能的影响。在整个实验过程中，为与主要结果保持一致，我们继续使用 GPT-4o 作为基础模型，并使用 RepoExEval 作为评估基准，同时采用与研究问题1相同的指标。

结果。 结果如表 **??**。三种知识组件均对最终性能有积极贡献。移除任一组件都会导致所有指标显著下降，而将它们结合起来则能实现最稳健且能感知上下文的异常处理生成。具体而言，移除上下文调用轨迹、处理模式和 API-异常映射后，整体 Pass@1 分数分别从 29% 降至 23%、21% 和 9%。这表明每种信息渠道都有助于确保可执行正确性，而 API-异常映射对于确保功能有效性不可或缺。

性能损失的程度因组件而异，凸显了各组件功能的不同：

- *API*-异常映射。该组件至关重要。移除该组件会导致性能下降幅度最大：CodeBLEU 降低 19.4%，IntentAcc 降低 19.6%，Pass@1 则骤降至仅 9%。若缺少有关易引发异常的 API 的知识，模型便无法识别相关异常类型，进而影响样本检索和模式抽象，导致整体性能下降。

- 上下文调用轨迹。移除上下文调用轨迹后，性能出现明显下降：CodeBLEU 降低 6.5%，IntentACC 降低 5.0%，Pass@1 下降 6 个百分点（从 29% 降至 23%）。这表明，上下文调用轨迹及相应的函数签名提供了宝贵的语义信息，这些信息对于跨过程异常推理至关重要。

- 处理模式。用少样本示例替代结构化处理模式会导致性能显著下降：CodeBLEU 降低 9.7%，IntentACC 降低 18.8%，Pass@1 下降 8 个百分点（从 29% 降至 21%）。这表明，抽象模式提供了可复用的知识，对于生成可泛化代码至关重要，其作用远不止于简单模仿示例。

研究问题3的回答： 这三个知识组件对 CATCHALL 都至关重要。API–异常映射最为关键：移除它会使 Pass@1 从 29% 降至 9%，这表明它是将代码仓库语义与可执行正确性联系起来的核心机制。上下文轨迹和处理模式则进一步提升了结构质量和语义对齐。

5.4. 检索样本数量的影响（研究问题4）

设置。 作为 CATCHALL 核心知识组件的异常处理模式，是从历史代码语料库中检索到的最相似的前 k 个样本中抽象得出的。由于模式质量对 k 高度敏感，我们

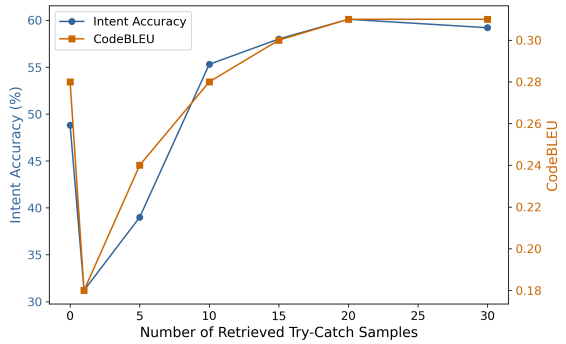

图 5： 不同检索数量的 try-catch 样本下 CatchAll 的性能。

通过将 k 从 0 调整到 30 来评估其影响，以确定最优的检索样本数量。其他所有设置和评估指标均与研究问题4保持一致。

结果。 如图 5所示，与零样本基线方法相比，仅使用 1 个或 5 个样本会导致性能明显下降。这种性能下降表明，较小的样本量可能产生嘈杂或有偏差的模式，从而误导后续代码生成。

性能随着 k 的增加而提升，检索到至少 10 个样本后，性能显著提高，这表明更高的多样性有助于实现更稳健的模式抽象。性能在 $k = 20$ 左右达到峰值，之后意图准确率略有下降（例如，在 $k = 30$ 时），这可能是无关或相互冲突的样本所带来的噪声导致的。同时，*Pass@1*随着 k 增大而稳步提升，表明更丰富的上下文多样性有助于生成更易执行且语义更准确的处理程序；然而，随着相似度趋于饱和，收益会逐渐递减，超过最佳范围后甚至会产生负面影响。

这些结果揭示了抽象可靠性与样本量之间的关键权衡：检索示例不足或过多，都可能分别因欠拟合和过拟合而损害生成质量。

对研究问题4的回答： CATCHALL 在检索约 20 个样本时性能最佳。少于 10 个样本时，多样性不足；而更大的样本集带来的收益则逐渐递减——不过，*Pass@1* 会持续小幅提升，直至 30 个样本，这反映了功能健壮性的增强。

5.5. 案例研究

为了直观展示 CATCHALL 的有效性，我们以 WordPress Android 项目中的一个代表性案例为例，如图 6所示。代码使用文章编辑器中的内容实例化了一个 `SpannableStringBuilder` 。跟踪`getSpannedContent()` 可以发现，调用链跨越三个文件，起始于安卓 SDK 的 `textView.getText()`。完整的跟踪包含四个方法：`getSpannedContent()` → `getText()`→ `EditTextUtils.getText(TextView)` →`TextView.getText()`。如果在 `textView` 为 null 时对其进行解引用，该调用链就可能引发运行时异常，这是一个需要同时运用跨文件分析和异常知识才能正确处理的复杂场景。

现有方法在处理此类情况时存在明显局限。直接提示无法预测具体的异常类型，通常会默认使用通用的 `Exception`catch 块，而该代码块只会打印堆栈跟踪。由于缺少仓库感知上下文和特定于异常的知识，语言模型无法准确推断错误的根本原因，也无法生成有意义的恢复行为。RepoCoder 在生成有意义的恢复逻辑和方法调用方面表现相对较好，并能准确重建回退行为。然而，它仍无法预测正确的异常类型，而是退回到通用的 `Exception`，这反映出它对 API 特定异常语义的理解有限。此外，它的生成过程耗时较长，需要 40 秒以上，而其他方法耗时不到 5 秒。与此同时，由于采用以抛出异常为中心的重写策略，KPC 会持续重新抛出异常，不支持恢复或回退机制等替代处理策略。这一局限显著缩小了它在实际场景中的适用范围，而这些场景往往需要实现平稳的错误恢复。

相比之下，CATCHALL 能够准确预测异常类型，并生成恰当的恢复逻辑。这得益于三个知识渠道的协同作用：（1）上下文调用轨迹 通过跨文件分析，定位 TextView. 获取文本() 处的异常风险；（2）*API*-异常映射 利用学习到的 API 行为，将该风险关联到 运行时异常；（3）异常处理模式 从代码仓库中以往的`try-catch` 代码块提取可复用模板。抽象出的模式指出，当方法抛出运行时异常时，应通过替代方法为变量赋予安全的回退值。因此，CATCHALL 将有风险的`getSpannedContent()`调用替换为 `StringUtils.notNullStr((String) mEditorFragment.getContent())`，这是一个稳健且符合代码仓库风格的回退方案。此案例证实，整合调用轨迹、API异常映射和处理模式，能够实现精准的异常预测与上下文感知代码生成。

6. 讨论

6.1. 为什么聚焦于`Try-Catch`?

CATCHALL 主要致力于合成具备代码仓库感知能力的`try-catch` 块，并将其作为异常处理的主要机制。这一设计选择符合异常处理研究中的既有实践 [10, 47],，其中构建`try-catch` 块始终被视为一项基础任务。促使我们聚焦于此的因素有几个：

- **实践中的普遍性**。在 Java 和安卓生态系统中，`try-catch` 构造是最常见的显式异常处理范式。我们对 3,200 个安卓代码仓库的分析证实了其主导地位，结果显示，超过 85% 的显式异常处理代码使用了 `try-catch` 块。

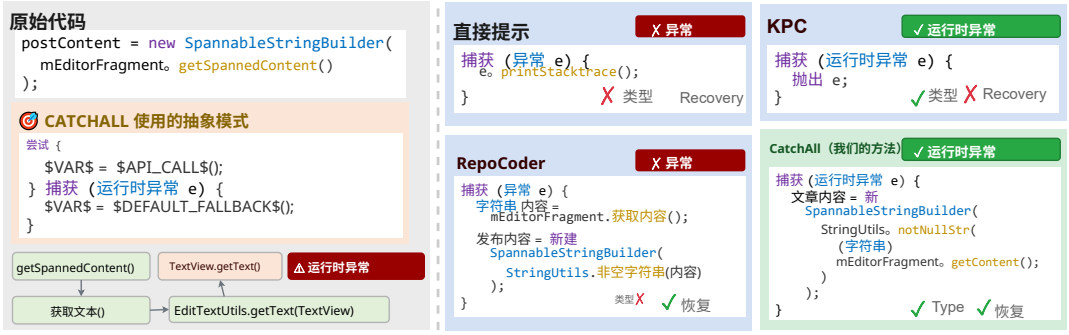

图 6： 示例 针对 WordPress Android 代码片段，不同方法生成的 catch 块 t.

• **异常处理的及时性**。在异常发生处使用try-catch捕获异常，可确保程序优雅地处理错误，并安全地继续运行或终止；同时，通过将错误处理逻辑封装在靠近错误源的位置，还能使代码更加自包含，更易于理解。因此，我们主张在适当的最早时机拦截并处理异常流，在此过程中主动构建健壮性并提升可维护性。

•**语义清晰度**。 通过指定具体的异常类型、处理策略和恢复逻辑，try-catch块在正常执行流与错误处理逻辑之间建立了清晰的语义边界。这种明确的分离提升了结构清晰度，进而有助于静态程序分析，以及异常类型预测和异常处理代码生成等由大语言模型辅助完成的任务。

我们承认，实际的错误管理还涵盖 try-catch之外的策略。我们当前的方法不会合成预防性处理 [48]（例如，使用前置条件来避免异常）、日志记录与可观测性 [49]（用于事后分析）、异常传播与转换 [50]，或结构化资源管理（例如，try-with-resources）。扩展 CATCHALL 以纳入更广泛的错误处理模式，是未来一项有价值的研究方向，可提升该工具在实际应用中的实用性，并扩大其对真实世界实践的覆盖范围。

6.2. CATCHALL 的计算效率如何?

我们分析了 CATCHALL 的计算效率和实际可扩展性。其成本结构包括一次性前期投入，以及每次合成查询极低的边际成本：

• **离线知识构建**。从包含 3,233 个代码仓库的语料库中提取 API 与异常的映射关系，约需 97 小时的计算时间。这是构建基础知识库的一次性成本，后续使用时可将其分摊。

•**每次查询的推理成本**。在推理阶段，每次查询的调用轨迹分析平均耗时 4.6 秒。随后，大语言模型推理通常会消耗约 1,800 个输入 token，并生成约 50 个输出 token。

这种成本结构确保了可扩展性。前期投入使其能够在 IDE 插件等环境中实际部署，而较低的单次查询开销则支持实时交互式开发者辅助。

6.3. 有效性威胁

我们指出本研究存在以下局限性和潜在的有效性威胁：

数据泄漏： 我们的方法使用大语言模型生成异常处理代码。随着这些模型规模不断扩大，其训练数据的覆盖范围也随之增加，从而提高了测试数据集中样本暴露的风险。为缓解这一问题，我们采用基于时间的过滤技术，尽可能减少训练集与测试集之间的数据重叠。我们还对 CodeSearchNet 进行了精确匹配验证。CodeSearchNet 是大语言模型预训练中常用的代表性语料库，验证结果发现仅有 4.4% 重叠（3,233 个代码仓库中有 143 个）。然而，即便采取了这些防护措施，仍存在一定的数据泄漏风险。

生成代码的评估： 我们主要使用 CodeBLEU 和意图预测准确率，在大规模 RepoExEval 基准上评估 CATCHALL。为评估功能正确性，我们还构建了可执行基准 RepoExEval-Exec，并使用 GPT-4o 对其进行评估。然而，由于时间和计算资源有限，当前版本的 RepoExEval-Exec 仅涵盖四个代码仓库。未来的工作将扩展这一基准，纳入更多代码仓库，并在更广泛的模型范围内开展更全面的基于测试的评估。

7. 结论

本文介绍了 CATCHALL，这是一种新颖的基于大语言模型的仓库感知异常处理方法。CATCHALL

采用知识增强框架，提取并整合上下文调用轨迹、易引发异常的 API 和处理模式，从而增强大语言模型在仓库上下文中合成准确 `try-catch` 块的能力。结果表明，CATCHALL 显著优于最先进的基线方法，在异常类型预测和异常处理代码生成方面均取得了更出色的性能。未来，我们将开发一个具备测试验证能力的多语言仓库感知基准测试框架，以支持多方面的评估，并推动仓库感知异常处理领域的研究进展。

用于复现我们研究的源代码和数据集可在此处获取：https://github.com/q4x3/CatchAll。

参考文献

[1] S. Thummalapenta、T. Xie，将异常处理规则挖掘为序列关联规则，载于：2009 年 IEEE 第 31 届国际软件工程大会，2009 年，第 496–506 页。[2] H. Shah、C. Gorg、M. J. Harrold，理解异常处理：新手与专家的观点，《IEEE 软件工程汇刊》36 (2) (2010) 150–161。[3] M. Asaduzzaman、M. Ahasanuzzaman、C. K. Roy、K. A. Schneider，开发人员如何在 Java 中使用异常处理？，载于：MSR 论文集，2016 年，第 516–519 页。[4] H. Melo、R. Coelho、C. Treude，揭示 Java 开发人员采用的异常处理指南，arXiv 预印本 arXiv:1901.08718 (2019)。[5] M. B. Kery 、 C. Le Goues 、 B. A. Myers，考察程序员在局部处理异常时的实践，载于：第 13 届国际软件仓库挖掘大会论文集，MSR '16，2016 年，第 484–487 页。[6] G. B. De Pádua、W. Shang，研究异常处理反模式的普遍性，载于：2017 年 IEEE/ACM 第 25 届程序理解国际会议 (ICPC)，2017 年，第 328–331 页。[7] G. B. de Pádua 、 W. Shang，借助异常流分析重新审视异常处理实践，arXiv 预印本 arXiv:1708.00817 (2017)。[8] M. Acharya、T. Xie，从源代码中挖掘 API 错误处理规范，载于：FASE 论文集，2009 年，第 370–384 页。[9] J.-W. Jo、B.-M. Chang、K. Yi、K.-M. Choe，Java 未捕获异常分析，《系统与软件杂志》72 (1) (2004) 59–69。[10] C. Fetzer、P. Felber、K. Hogstedt，非原子异常处理的自动检测与屏蔽，《IEEE 软件工程汇刊》30 (8) (2004) 547–560。[11] H. Zhong，我们应该抛出哪种异常？，载于：第 37 届 IEEE/ACM 国际自动化软件工程大会，ASE 2022，美国密歇根州罗切斯特，2022 年 10 月 10–14 日，ACM，2022 年，第 116:1–116:12 页。[12] J. Zhang、X. Wang、H. Zhang、H. Sun、Y. Pu、X. Liu，学习处理异常，载于：第 35 届 IEEE/ACM 国际自动化软件工程大会 (ASE)，澳大利亚墨尔本，2020 年 9 月 21–25 日，IEEE，2020 年，第 29–41 页。[13] X. Ren、X. Ye、D. Zhao、Z. Xing、X. Yang，从误用到精通：通过知识驱动的 AI 链式调用增强代码生成，载于：第 38 届 IEEE/ACM 国际自动化软件工程大会，ASE 2023，卢森堡，2023 年 9 月 11–15 日，IEEE，2023 年，第 976–987 页。[14] X. Zhang、Y. Chen、Y. Yuan、M. Huang，Seeker：采用基于大语言模型的多智能体方法增强代码中的异常处理，CoRR abs/2410.06949 (2024)。[15] S. Liang、N. Jiang、Y. Hu、L. Tan，语言模型能取代程序员编写代码吗？RepoCod 表示“还不能”，载于：ACL 论文集，2025 年，第24698–24717页。[16] wordpress-mobile，WordPress-Android，https://github.com/wordpress-mobile/WordPress-Android （2025）。[17] B. Cabral，P. Marques，《异常处理：Java 和 .net 中的一项实地研究》，载于：ECOOP 论文集，2007年，第151–175页。[18] D. Sena，R. Coelho，U. Kulesza，R. Bonifácio，《理解 Java 库的异常处理策略：一项实证研究》，载于：MSR 论文集，2016年，第212–222页。[19] A. Li，S. Lu，S. Nath，R. Padhye，V. Sekar，《ExChain：用于根因诊断的异常依赖分析》，载于：第21届 USENIX 网络系统设计与实现研讨会（NSDI 24），2024年，第2047–2062页。[20] P. Koopman，J. DeVale，《POSIX 操作系统的异常处理有效性》，IEEE 软件工程汇刊 26（9）（2000）837–848。[21] H. Chen，W. Dou，Y. Jiang，F. Qin，《理解大规模云系统中的异常相关缺陷》，载于：ASE 论文集，2019年，第339–351页。[22] M. Bruntink，A. Van Deursen，T. Tourwé，《发现基于惯用模式的异常处理中的故障》，载于：ICSE 论文集，2006年，第242–251页。[23] R. Coelho，L. Almeida，G. Gousios，A. Van Deursen，C. Treude，《安卓中的异常处理缺陷风险》，实证软件工程 22（3）（2017）1264–1304。[24] L. Fan，T. Su，S. Chen，G. Meng，Y. Liu，L. Xu，G. Pu，Z. Su，《安卓应用中框架特定异常的大规模分析》，载于：ICSE 论文集，2018年，第408–419页。[25] N. Cacho，E. A. Barbosa，J. Araujo，F. Pranto，A. Garcia，T. Cesar，E. Soares，A. Cassio，T. Filipe，I. Garcia，《异常处理行为如何演变？对 Java 和 C# 应用程序的一项探索性研究》，载于：ICSME 论文集，2014年，第31–40页。[26] A. Oliveira，J. Correia，L. Sousa，W. K. G. Assunção，D. Coutinho，A. Garcia，W. Oizumi，C. Barbosa，A. Uchôa，J. A. Pereira，《别忘了异常！：通过考察健壮性变化识别设计问题》，载于：2023年 IEEE/ACM 第20届国际软件仓库挖掘大会（MSR），2023年，第417–429页。[27] A. da Silva，R. G. Vieira，D. P. P. Mesquita，J. P. P. Gomes，L. S. Rocha，《迈向异常处理缺陷的自动标注：对 Apache Hadoop 十年缺陷修复的案例研究》，实证软件工程 29（4）（2024）85。[28] D. Marcilio，C. A. Furia，《抛出了什么？轻量且精确地自动提取 Java 方法中的异常前置条件》，载于：2022年 IEEE 国际软件维护与演化会议（ICSME），2022年，第340–351页。[29] S. Jana，Y. J. Kang，S. Roth，B. Ray，《利用错误规范自动检测错误处理缺陷》，载于：USENIX 安全会议论文集，2016年，第345–362页。[30] M. M. Rahman，C. K. Roy，《在推荐异常处理代码示例时使用上下文》，载于：SCAM 论文集，2014年，第285–294页。[31] T. Nguyen，P. Vu，T. Nguyen，《推荐异常处理代码》，载于：ICSME 论文集，2019年，第390–393页。[32] T. Nguyen，P. Vu，T. Nguyen，《面向异常处理的代码推荐》，载于：ESEC/FSE 论文集，2020年，第1027–1038页。[33] T. Nguyen，P. Vu，T. Nguyen，载于：ESEC/FSE '20：第28届 ACM 欧洲软件工程大会与软件工程基础研讨会，2020年11月8日至13日，美国线上活动，ACM，2020年，第1027–1038页。[34] Y. Zhang，Y. Xue，《ExceRef：自动重构以实现异常处理》，载于：第15届亚太互联网软件研讨会论文集，Internetware '24，2024年，第239–248页。[35] H. Zhang，J. Luo，M. Hu，J. Yan，J. Zhang，Z. Qiu，《检测 C++ 程序中的异常处理缺陷》，载于：第45届国际软件工程会议论文集，ICSE '23，IEEE Press，2023年，第1084–1096页。[36] R. Li，B. Chen，F. Zhang，C. Sun，X. Peng，《利用深度代码表示学习与基于注意力机制的图神经网络检测运行时异常》，载于：2022年 IEEE 国际软件分析、演化与再工程会议（SANER），2022年，第373–384页。

[37] X. Jia，S. Chen，X. Zhou，X. Li，R. Yu，X. Chen，J. Xuan，《应该在哪里处理异常？从全局视角推荐异常处理位置》，载于：2021年IEEE/ACM第29届程序理解国际会议（ICPC），2021年，第369–380页。[38] X. Jia，S. Chen，X. Zhou，X. Li，R. Yu，X. Chen，J. Xuan，《应该在哪里处理异常？从全局视角推荐异常处理位置》，载于：ICPC 论文集，2021年，第369–380页。[39] L. Xu，H. Zhong，《检测不一致的抛出异常》，载于：ICPC 论文集，2021年，第391–395页。[40] C. Zhang，Q. Tao，L. Chen，M. Zhang，《基于 BERT 的代码学习，用于异常定位和类型预测》，AAAI人工智能大会论文集 39（1）（2025）1040–1047。[41] Y. Cai，A. Yadavally，A. Mishra，G. Montejo，T. N. Nguyen，《利用 CodeBERT 实现异常处理编程助手》，载于：2024年 IEEE/ACM第46届软件工程国际会议（ICSE），2024年，第1146–1158页。[42] S. Ren，D. Guo，S. Lu，L. Zhou，S. Liu，D. Tang，N. Sundaresan，M. Zhou，A. Blanco，S. Ma，《CodeBLEU：一种用于代码合成自动评估的方法》，CoRR abs/2009.10297（2020）。[43] Y. Peng，S. Gao，C. Gao，Y. Huo，M. Lyu，《领域知识至关重要：利用修复模板改进 Python 类型错误修复提示》，载于：IEEE/ACM第46届软件工程国际会议论文集，ICSE ’24，2024年。[44] F. Zhang，B. Chen，Y. Zhang，J. Keung，J. Liu，D. Zan，Y. Mao，J. Lou，W. Chen，《RepoCoder：通过迭代检索与生成实现仓库级代码补全》，载于：2023年自然语言处理实证方法会议论文集，EMNLP 2023，2023年12月6日至10日，新加坡，2023年，第2471–2484页。[45] M. Chen，J. Tworek，H. Jun，Q. Yuan，H. P. de Oliveira Pinto，J. Kaplan，H. Edwards，Y. Burda，N. Joseph，G. Brockman，A. Ray，R. Puri，G. Krueger，M. Petrov，H. Khlaaf，G. Sastry，P. Mishkin，B. Chan、S. Gray、N. Ryder、M. Pavlov、A. Power、L. Kaiser、M. Bavar- ian、C. Winter、P. Tillet、F. P. Such、D. Cummings、M. Plappert、F. Chantzis、E. Barnes、A. Herbert-Voss、W. H. Guss、A. Nichol、A. Paino、N. Tezak、J. Tang、I. Babuschkin、S. Balaji、S. Jain、W. Saunders、C. Hesse、A. N. Carr、J. Leike、J. Achiam、V. Misra、E. Morikawa、A. Radford、M. Knight、M. Brundage、M. Murati、K. Mayer、P. Welinder、B. McGrew、D. Amodei、S. McCandlish、I. Sutskever、W. Zaremba，对基于代码训练的大语言模型进行评估(2021)。arXiv:2107.03374。网址https://arxiv.org/abs/2107.03374[46] S. Parikh、M. Tiwari、P. Tumbade、Q. Vohra，《探索用于意图分类的零样本和少样本技术》，载于：ACL 2023，加拿大多伦多，2023年7月9日至14日，2023年，第744–751页。[47] H. Zhong，《我们应该抛出哪种异常？》，载于：ASE论文集，2022年，第1–12页。[48] H. Chang、L. Mariani、M. Pezzè，用于修复基于组件的系统的异常处理程序，ACM软件工程与方法学汇刊，22卷4期（2013年10月）。doi:10.1145/2522920.2522923。网址https://doi.org/10.1145/2522920.2522923[49] Q. Fu、J. Zhu、W. Hu、J.-G. Lou、R. Ding、Q. Lin、D. Zhang、T. Xie，开发人员在哪里记录日志？工业界日志记录实践的实证研究，载于：第36届国际软件工程会议伴随论文集，ICSE Companion 2014，计算机协会，纽约州纽约市，2014年，第24–33页。doi:10.1145/2591062.2591175。网址https://doi.org/10.1145/2591062.2591175[50] P. Zhang、S. Elbaum，扩充测试以验证异常处理代码：移动应用领域的一项扩展研究，ACM软件工程与方法学汇刊，23卷4期（2014年9月）。doi:10.1145/ 2652483。网址https://doi.org/10.1145/2652483