MUFIN：利用反向翻译改进神经修复模型

André Silva 瑞典皇家理工学院 andreans@kth.se

João F. Ferreira 里斯本大学 INESC-ID 与 IST，葡萄牙 joao@joaoff.com

He Ye 瑞典皇家理工学院 heye@kth.se

Martin Monperrus 瑞典皇家理工学院 monperrus@kth.se

arXiv:2304.02301v1 [cs.SE] 5 Apr 2023

摘要

自动化程序修复旨在自动修复软件缺陷。该领域一种颇具前景的方向是自监督学习，这是一种无需使用表示缺陷/修复配对的提交来训练修复模型的学习范式。在自监督神经程序修复中，这些缺陷/修复对通过某些方式生成。主要难题是生成有意义且多样化的配对，以最大限度地提升训练效果。为解决这一问题，我们提出采用回译技术，这项技术源自神经机器翻译。我们设计并实现了 MUFIN，这是一种用于程序修复的回译训练技术，并专门设计了代码评判器来筛选高质量训练样本。结果表明，MUFIN 的回译循环能够以完全自动化的自监督方式生成有价值的训练样本，共生成超过五十万组缺陷/修复对。代码评判器的设计至关重要，因为回译过程中存在一项根本性的权衡：评判器的限制程度与可用于优化的样本数量之间需要取得平衡。

关键词

自动化程序修复、自监督学习、回译

ACM 引用格式：

André Silva、João F. Ferreira、He Ye 和 Martin Monperrus。2023。MUFIN：通过回译改进神经修复模型。载于 *ACM* 会议论文集（*Conference'17*）。ACM，纽约州纽约市，10 页。https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 引言

在过去几十年里，软件系统不断演进，成为人类有史以来最复杂的人工制品之一。开发软件通常是一项多步骤的工作，由技能各异的团队成员共同完成。调试，即发现并修复软件缺陷的活动，是软件开发中最具挑战性的工作之一 [1]。它要求开发人员分析并理解代码和错误日志、自己没有编写的代码，并找到能修复错误的程序修改，也就是补丁。

自动程序修复（APR） [2] 是一个活跃的研究领域，致力于自动查找软件缺陷的修复补丁，无需耗费开发人员的时间。人们预计，APR 能够节省宝贵的人力工时，降低软件系统开发和维护成本，并减少此类工作所需的人员数量，最终推动世界走向全面自动化。

传统的 APR 方法（例如 [3–9]）要么依赖人工定义的程序修改规则，要么基于约束推断与求解，因此适用范围仅限于部分软件缺陷类型。另一方面，较新的基于学习的 APR 方法（例如 [10–17]）不依赖特定缺陷类型，而是从训练数据中学习修复变换，无需人工干预。

基于学习的 APR 对数据需求很大。此前针对 APR 的监督学习研究投入了大量精力来构建缺陷及其修复的数据集，通常取材于代码仓库（例如 GitHub、GitLab）中的提交、学生作业和编程竞赛。虽然收集磁盘上的代码可以规模化进行，但由于缺少配置文件和执行环境，执行故障代码及其修复版本却难以规模化。因此，汇总得到的可执行代码数据集总共只有几千个样本 [18–21]。

自监督学习（SSL）为收集可执行的缺陷/修复代码对这一问题提供了一种解决方案 [22]。这种方法不再收集历史提交，而是通过自动化流程合成可用于训练神经网络的缺陷/修复代码对，从而生成训练样本。自监督学习在 APR 研究中仍是一个相对少有人探索的领域，目前只有少数论文研究语法错误 [15, 17]、简单的 Python 错误 [16]，以及利用执行诊断信息进行修复 [22]。

本文提出 MUFIN，一种用于功能性程序修复的新型自监督方法。MUFIN 的主要目标是大幅提升给定神经程序修复模型的泛化能力。MUFIN 的核心创新在于采用反向翻译，并结合代码评估器从生成的样本中筛选高质量样本。反向翻译始终使用两个模型，在我们的案例中分别是修复器和破坏器。MUFIN 交替微调破坏器和修复器模型：使用修复器模型为破坏器生成训练样本，反之亦然。这种训练数据生成策略使模型能够接触到越来越多样化的训练数据，从而提升其对未见测试数据集的泛化能力。据我们所知，

允许出于个人或课堂用途，免费制作本作品全部或部分内容的数字或纸质副本，前提是这些副本不用于或分发以获取利润或商业利益，并且副本首页须附有本声明和完整引文。对于本作品中归他人而非 ACM 所有的部分，必须尊重其版权。注明出处后可进行摘要。若要以其他方式复制、再版、发布至服务器或分发至列表，则须事先获得明确许可和/或支付费用。请通过 permissions@acm.org 申请许可。
Conference'17，*2017* 年 7 月，美国华盛顿特区 © 2023 Association for Computing Machinery。ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

据我们所知，我们率先将这一概念应用于功能性程序修复。

我们在两个广受认可的 Java 程序修复基准上评估了 MUFIN：QuixBugs [20] 和 Defects4J [18]。结果表明，MUFIN 的表现优于基线模型，修复的缺陷数量比表现第二好的模型更多：在 Defects4J 中为 +12 个，在 QuixBugs 中为 +4 个。此外，我们研究了 MUFIN 中评论器设计的影响。实验结果揭示了评论器限制程度与训练样本数量之间的权衡，证实了评论器选择的重要性。表现最佳的评论器仅依据编译结果进行判断；限制更宽松或更严格的评论器效果都更差。

总而言之，本文的主要贡献 e:

- 我们设计并实现了 MUFIN：一种新颖的自监督功能性程序修复方法。与收集提交记录用于训练不同，MUFIN 能生成有价值的训练样本。
- 我们提出了三类用于自监督函数式程序修复的代码评判器。据我们所知，我们是首个研究用于 Java 程序修复的回译评判器的团队。
- 我们开展了一系列实验，以衡量 MUFIN 相较于强基线的改进程度。结果表明，MUFIN 有助于提升生成补丁的质量，并提高补丁的可编译性。
- 我们已在 https://github.com/andre15silva/ mufin 公开本研究的代码、数据集和实验结果。

2 MUFIN 方法

MUFIN 是一种原创的自监督自动程序修复方法，如图 1 所示。MUFIN 的核心创新在于其训练流程，该流程结合了三个训练阶段，后两个阶段构成一个循环。

MUFIN 需要一个已初始化的神经破坏器和神经修复器（*MUFIN* 初始化）。神经修复器旨在根据有缺陷的代码生成正确代码，而神经破坏器则旨在根据正确代码生成有缺陷的代码。若有现成模型，两者都可以采用先前工作中的模型；也可以从头开始训练，详见第 2.1 小节。

在第二阶段，MUFIN 使用反向翻译循环（*MUFIN* 反向翻译）交替微调 breaker 和 fixer 模型。反向翻译是指使用一个将文本从某个领域翻译到另一个领域的模型，为负责反向翻译的第二个模型生成新的训练样本，反之亦然，并通过迭代不断提升两个模型的性能 [23]。

MUFIN 反向翻译的首要目标是以完全自监督的方式，为 breaker 和 fixer 累积有价值的训练样本。生成的训练样本会被保存下来，并在每个优化步骤中统一使用。因此，随着迭代推进，每次微调所使用的训练样本数量都会增加，从而逐步提升模型性能。

整个训练过程的最终成果是一个高质量的神经修复器。自动生成训练样本对于让模型接触多样化的样本至关重要，从而提升其性能。其泛化能力。该训练流程的核心优势在于，它完全采用自监督方式，无需像监督式程序修复那样，繁琐且耗费高昂地收集成对的错误与修复样本。

现在，我们将详细介绍 MUFIN 的各个组成部分。

2.1 MUFIN 初始化

MUFIN 初始化阶段旨在初始化两个神经模型：神经修复器和神经破坏器。为启动后续的 MUFIN 反向翻译阶段，必须准备好这两个模型。它们需要具备合理的性能，因为其输出将在反向翻译循环中用作训练样本。

修复器初始化。神经修复器能够根据有缺陷的代码生成正确代码。合理的神经修复器可以通过多种方式获得：1）利用以往研究中公开发布、可复用且可微调的修复器模型；2）使用提交记录进行传统的监督训练；或 3）利用机械生成的样本进行自监督训练，样本的形式为从人工构造的有缺陷代码到正确代码。

破坏器初始化。神经破坏器能够根据正确代码生成有缺陷的代码。以往研究已经构建并分享了修复器模型，但破坏器模型十分罕见 [24]。因此，破坏器与修复器的可用性并不对称，这使得破坏器的初始化需要专门研究。事实上，目前要实现 MUFIN，就必须从头开始训练破坏器。

在 MUFIN 初始化过程中，用于初始化破坏器的训练数据由机械破坏器生成：机械破坏器由人工定义的破坏规则组成，这些规则会修改正确代码，使其变得有缺陷（它并非神经网络）。变异测试工具就是这类机械破坏器的一个例子。

给定一个机械式破坏器，MUFIN 从通过测试套件的正确程序开始。对于每个正确程序，机械式破坏器会将每条破坏规则应用于正确样本中的多个位置，从而生成多个错误。

清单 1 展示了机械式破坏器生成的一个错误示例。在此示例中，机械式破坏器交换了方法调用 *serializeFields* 的前两个参数。通过这样修改程序，机械式破坏器改变了程序的行为并引入了一个错误。

2.2 MUFIN 回译

MUFIN 回译阶段的目标是通过迭代使用一个模型的输出训练另一个模型，来改进已初始化的修复器和破坏器模型。MUFIN 通过使用未配对数据来实现这一目标（即，未与错误版本关联的正确样本，或未与修复补丁关联的错误样本）。评论器（第 2.3 节）会筛除不符合质量标准的输出，以尽可能提高用于训练的样本质量。

该过程首先通过应用初始破坏器生成正确程序作为种子，或者通过应用初始神经修复器生成有缺陷的程序作为种子。假设采用后一种方式，随后会根据正确代码评判器筛选修复器输出的暂定正确补丁：正确代码评判器会过滤样本，只保留那些

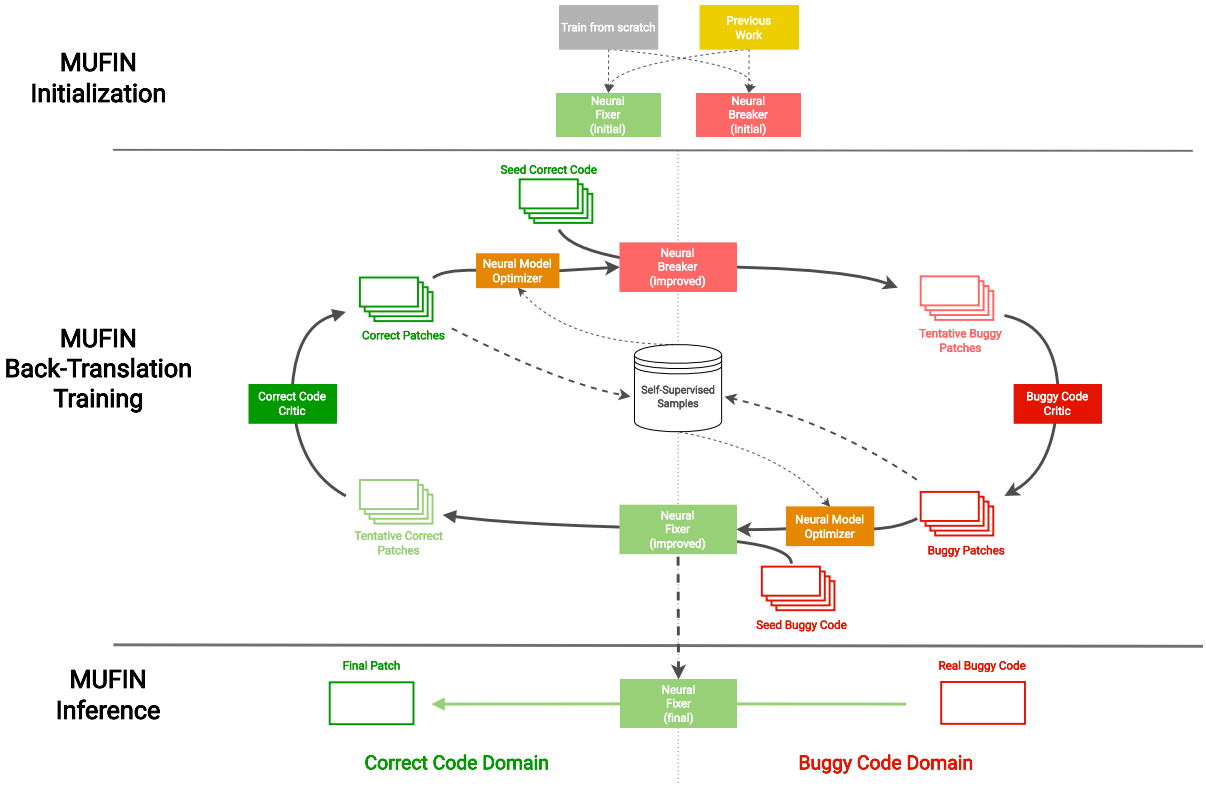


图 1：MUFIN 方法概览。

列表 1：由机械破坏器（SelfAPR [22]）生成的训练样本示例。机械破坏器交换方法调用 *serialize-Fields* 的前两个参数，从而改变程序的预期行为。

```
--- a/correct.java
+++ b/buggy.java
@@ -552,11 +551,11 @@
  } else {
-  serializeFields(value, gen, provider);
+  serializeFields(gen, value, provider);
  }
```

它认为是正确的。然后，使用自监督样本对神经缺陷生成器进行微调，使其能够将正确程序转换为有缺陷的程序。

利用改进后的神经缺陷生成器，可以从正确程序生成暂定的有缺陷补丁。随后，根据有缺陷代码判别器对这些暂定补丁进行筛选：该判别器会筛选样本，只保留其判定为有缺陷的样本。然后，使用这些自监督样本对神经修复器进行微调，使其能够将有缺陷的程序转换为正确的程序。

MUFIN 反向翻译可配置为运行 N 次迭代，并在每个生成步骤中使用束搜索生成 K 个候选补丁。在每次迭代中，两个神经模型都会通过反向传播得到改进。一次迭代生成的数据会在后续迭代中用作种子数据。

在回译循环结束时，如果只做程序修复，可以丢弃破坏器，但破坏器可能在其他任务中复用，见第5.1小节。最重要的成果是最终的神经修复器，随后用于推理。

2.3 回译中的评判器

在回译中，生成高质量训练样本是一项重大挑战，因为它可能削弱该过程的有效性 [15, 25]。为解决这一问题，人们使用“评判器”来评估所生成样本的质量。评判器是根据质量标准对样本进行判断的函数（定理2.1）。在面向程序修复的回译中，需要两个评判器：一个用于保留高质量样本以训练神经修复器，另一个用于保留高质量样本以训练神经破坏器。

定义 *2.1*（评判器）。评判器是一个谓词函数 $C : X \rightarrow \{0,1\}$，其中 X 是程序空间；若 x 被视为可接受，则 $C(x) = 1$，否则 $C(x) = 0$。

评判器可以配置不同程度的严格性。提高严格性可确保只选取高质量样本，但会减少保留下来的训练样本数量。下面介绍 MUFIN 中采用的两类评判器，它们的严格程度各不相同：

基于编译器的评判器。评判器可以仅依据编译结果进行判断。如果编译失败，程序就无法执行，也就无法评估是否存在功能性错误。各评判器的行为如下：

编译正确代码评判器：保留所有成功编译的程序。

编译错误代码评判器：保留所有成功编译的程序。其目的是让网络专注于能够编译但测试套件未通过的功能性错误，而不是编译器错误。

基于测试的评判器。评判器可以基于测试执行。单元测试失败是存在功能性错误的存在性证明。各评判器的行为如下：

测试通过代码评判器：保留所有成功编译且通过全部测试的程序。

有缺陷代码测试批评器：保留所有能够成功编译但至少有一个单元测试失败的程序。

2.4 MUFIN 推理

MUFIN 推理指的是将 MUFIN 应用于真实世界中的缺陷。在此阶段，MUFIN 使用最终的神经修复器修复有缺陷的代码。给定待修复的代码后，按照先前的工作 [11],，MUFIN 会采用故障定位方法（例如 GZoltar [26],、Flacoco [27]）来确定一系列可疑位置。随后，神经修复器会应用于这些位置。对于每个位置，MUFIN 使用束搜索生成并排序一系列 K 补丁。

2.5 位置选择的破坏

在回译循环中，一个重要环节是选择破坏器将要操作并生成缺陷的位置。由于 MUFIN 需要处理包含多个源代码文件的项目，如何选择合适的位置进行破坏是一项挑战。

一种简单的做法是随机选择源代码行。然而，随机选择源代码行可能效果不佳，因为选中的位置未必相关。例如，可能会选中类声明之外的行，从而在回译过程中引入额外噪声。

为应对这一挑战，MUFIN 会遍历每个程序中的每个文件，并使用基于 AST 的分析 [28] 来识别位于代码块中的语句。对于每条语句，系统都会输出一对行号，分别表示该语句的起始行和结束行。根据识别出的位置，断裂模型接收这些位置的输入表示，并返回若干可能存在错误的代码片段。

2.6 输入表示

神经模型接收的是一个词元序列。为构建该输入表示，我们遵循现有研究（[11, 13,22]），并纳入待修改代码片段的上下文信息与故障定位信息。我们的输入表示由拼接而成的三部分组成。第一部分包含待修改代码段之前的 n 行（可配置）。第二部分由两个特殊词元 [START_BUGGY] 和 [END_BUGGY], 界定，用以标记待修改代码部分的开始与结束。最后，第三部分包含该代码部分之后的若干行待修改。神经修复模型与神经破坏模型使用相同的表示。

2.7 实现

我们复用了 He 等人在 SelfAPR Self- APR [22] 中使用的机械断裂器。我们采用 HuggingFace 的先进编码器-解码器 Transformer 架构 T5 [29], 实现 MUFIN 的神经模型。模型维度参照 t5-small 设置：d_{model} = 512、d_{ff} = 2048、d_{kv} = 64，采用 8 头注意力机制，编码器和解码器各包含 6 层，总参数量约为 6000 万。我们使用 HuggingFace 的 PLBART [30] Sentence-Piece 分词器。我们留出 2% 的训练数据用于验证。模型使用 AdamW [31, 32],进行优化，批量大小为 16，学习率为 1e-4，梯度衰减率为 0.01，并在单块 GPU（NVIDIA RTX A4000）上训练。在每个训练阶段，我们都会根据验证损失进行早停。

在 MUFIN 回译中生成正确代码时，我们将 K 设为 10，这与先前工作 [15],一致；由于可用代码位置的数量有限，生成有缺陷代码时则将其设为 1。在 MUFIN 推理期间，我们将K 设为 100，这一数值处于相关工作 [11, 22, 33]所采用的范围内。我们在补丁生成过程中采用早停策略：当所有波束序列都到达 EOS 标记时，生成即停止。

3 实验方法

3.1 研究问题

- **RQ1（回译的影响）**：与不使用回译的基线模型相比，MUFIN 的回译有何影响?
- **RQ2（评论器的影响）**：评论器设计在多大程度上影响 MUFIN 的有效性?

3.2 数据集构建

为了评估 MUFIN，我们构建了用于训练和测试的数据集。鉴于当前神经模型在修复多处位置的程序错误方面存在局限性 [34], 我们将训练集和测试集限定为仅包含单个代码块中的错误。

对于训练，我们从原始 Bears 数据集 [19]构建了两个输入数据集：一个包含正确程序，另一个包含有缺陷的程序。Bears 是一个可复现 Java 缺陷的数据集，包含从 72 个不同开源项目中收集的 251 个缺陷。我们选择 Bears，是因为其样本附带编译器和测试套件，而这正是 MUFIN 的评判器所需要的。

对于测试，我们采用两个广泛使用的基准：Defects4J [18] 和 QuixBugs [20]。Defects4J v2.0 包含从 17 个不同开源项目中收集的 835 个缺陷。QuixBugs 包含从 Quixey 挑战赛中收集的 40 个程序。

下面我们将详细说明每个数据集的构建方式。

3.2.1 训练数据集。

正确程序数据集。 我们采用 Ye 等人 [22] 的方法，提取每个项目最早的样本。对于每个样本，我们将修补后的版本作为正确程序。这样可以避免对任何项目过度采样，因为每个项目都由多个样本代表。无法在本地复现的程序会被过滤掉，以确保数据一致性。

从原始 Bears 数据集中，我们共获得 56 个正确程序。项目数量与正确程序数量不一致，是因为未能复现 16 个正确程序，其中 14 个无法编译，2 个未能通过测试。总体而言，该数据集包含 1,342,614 行代码和 70,160 个单元测试。

缺陷程序数据集。为构建缺陷程序数据集，我们尝试在本地复现所有可用样本，并移除无法复现的样本。对于每个样本，我们仅保留缺陷版本。最终收集的数据集包含来自 Bears 数据集的 61 个缺陷程序。

3.2.2 测试数据集。为创建测试数据集，我们对 Defects4J 和 QuixBugs 数据集应用筛选条件，仅选择满足以下两个标准的缺陷：(i) 可复现，以及 (ii) 未包含在训练数据集中。

我们从 Defects4J 收集了共计 428 个测试样本，从 QuixBugs 收集了 39 个测试样本。值得注意的是，我们排除了 *JacksonDatabind* 项目中的所有缺陷，以防止数据溢出。做出这一决定是因为 Bears（用于训练）和 Defects4J（用于测试）都包含从该项目收集的样本。我们已采取预防措施以避免进一步的跨数据集污染，并且未发现其他此类情况。

3.3 基线模型

MUFIN反向翻译需要合理的模型作为输入：修复器和破坏器。在实验中，我们需要控制模型配置（即训练数据集、输入表示、架构和超参数）。为此，我们从头开始训练神经修复器和破坏器。由此得到基线模型，以便证明反向翻译能够带来改进。

我们的基线模型使用机械生成的样本进行训练，这些样本由 SelfAPR [22] 的错误注入模型应用于正确程序数据集生成。神经破坏器经过训练，可将正确程序转换为有缺陷的程序；神经修复器则经过训练，可将有缺陷的程序转换为正确程序。随后，我们在RQ1（第3.4小节）和RQ2（第3.5小节）中使用这些模型来初始化MUFIN反向翻译。为了提供更多参考，我们还使用了一个按照相关工作BugLab [16], 的方法训练的基线模型，并以此命名。

3.4 RQ1 的方法

在 RQ1 中，我们将 MUFIN 与第 3.3 小节所述的基线模型进行比较。为此，我们根据配置空间上的超参数优化结果，选取表现最佳的 MUFIN 模型，其中包括 MUFIN 回译中的评论器。

为了评估神经修复模型的性能，我们使用广受认可的 Defects4J [18] 和 QuixBugs [20] 基准。我们计算 1）可能修复的缺陷数量（即人工编写的测试通过时）以及 2）正确修复的缺陷数量（即生成的补丁与人工编写的补丁等价，并经人工检查确认）。

为消除故障定位步骤所继承的偏差，我们在实验设置中考虑完美故障定位，如相关工作 [11, 12, 22] 中所做的那样。这种方法使我们能够仅专注于

表 1：MUFIN 相较于最先进的自监督程序修复方法的修复效果。与其他方法相比，MUFIN 在两个基准上正确修复和可能修复的缺陷都更多。

方法	# 训练样本数		测试	
	初始化	自监督	QuixBugs（39个缺陷）	D4J（428个缺陷）
BugLab [16]	821,311	-	2/4	16/31
基线模型	3,942,935	-	2/4	16/43
MUFIN	3,942,935	197,959	**6/7**	**28/62**

APR 流程的补丁生成步骤，并便于对不同方法的性能进行公平比较 [35]。

3.5 RQ2 的方法

在 RQ2 中，我们研究评审器设计对 MUFIN 整体有效性的影响程度。为此，我们使用 MUFIN 反向翻译，采用三种不同策略对第 3.3 小节所述的基线模型进行微调：1）不使用评审器；2）使用 编译器 评审器系列；3）使用 测试 评审器系列。第 2.3 小节介绍了这些评审器系列。模型按照 RQ1 的方法进行训练和评估。此外，我们还计算每个模型在每个测试数据集上生成的可编译补丁所占百分比，以衡量生成补丁的语法质量。

4 实验结果

4.1 RQ1 结果（反向翻译）

在 RQ1 中，我们将 MUFIN 的修复效果与第 3.3 小节所述的基线模型以及 BugLab [16] 进行比较。评估结果见表 1，其中展示了各方法在两个测试数据集上的表现。该表的结构如下：第一列显示用于训练各模型的方法，第二列和第三列分别表示各训练阶段使用的训练样本数量，其中第 3 列表示 MUFIN 反向翻译中通过自监督生成的样本数量。最后两列给出了在 QuixBugs 和 Defects4J 上的测试结果。对于每个 X/Y 单元格，X 表示成功修复的缺陷数量（即人工编写的测试通过），Y 表示可能已修复的缺陷数量（即人工编写的测试通过）。

在 MUFIN 初始化阶段使用 821,311 个样本进行优化的 BugLab [16],，分别能修复 QuixBugs 和 Defects4J 中的 2 个和 16 个缺陷。此外，它分别能合理地修复 QuixBugs 和 Defects4J 中的 4 个和 31 个缺陷。

基线模型在 MUFIN 初始化阶段使用 3,942,935 个样本进行优化，可以分别修复 QuixBugs 和 Defects4J 中的 2 个和 16 个错误。此外，它有望分别修复 QuixBugs 和 Defects4J 中的 4 个和 43 个错误。

MUFIN 的黄金模型正确修复了 Defects4J 中的 28 个错误和 QuixBugs 中的 6 个错误。MUFIN 明显优于基线模型：MUFIN 正确修复了 +12 个 Defects4J 错误，在 QuixBugs 上也观察到相同趋势。由于 MUFIN 基于完全相同的基线模型，这意味着性能提升来自 MUFIN 的核心贡献：反向翻译循环。通过迭代

使用一个模型的输出训练另一个模型，从而改进神经修复器和破坏器模型，MUFIN 生成了有价值的训练样本，提升了模型对未见测试数据集的泛化能力。同样，在生成合理补丁方面也观察到了更好的性能：MUFIN 为 Defects4J 中的 62 个错误和 QuixBugs 中的 7 个错误生成了合理修复，优于基线模型。

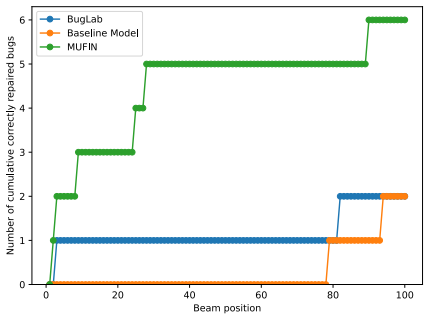
泛化示例。现在来考虑一个只有 MUFIN 的模型才能修复的错误。例如，代码清单 2 展示了 MUFIN 对 Defects4J 中 Cli-5 错误给出的正确补丁。在此补丁中，赋值语句右侧的值从算术运算修改为字面量。基线模型和 BugLab 都无法正确修复此错误。这表明 MUFIN 生成了有价值的训练样本，用于修改算术表达式。

代码清单 2：MUFIN 对 Defects4J 中 Cli-5 错误给出的正确补丁。基线模型和 BugLab 模型都无法正确修复此错误。

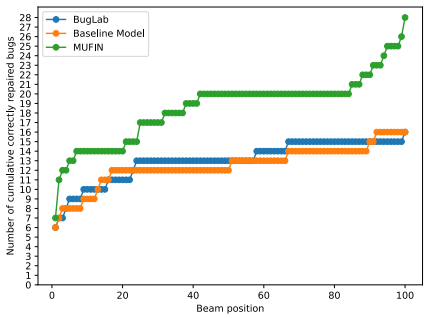
```
--- a/buggy.java+++
b/correct.java@@ -552,11 +551,11
@@// 停止发生的无限循环-
nextLineTabStop =width-1;+
nextLineTabStop = 1;}
```

束上的补丁正确性。 图 2 绘制了各模型在两个测试基准上沿推理束累计正确修复的缺陷数量。这有助于观察正确补丁位于束中的何处。从每个基准的两个子图中可以看出，MUFIN 能在束中更早识别出正确补丁。这表明 MUFIN 改善了束内的补丁优先级排序。这意味着，得益于 MUFIN，驱动束搜索的底层似然能更好地刻画正确性。

与最先进方法的性能比较。 根据我们的方法，我们使用的模型比相关工作中的模型更小（例如，我们的模型有 60M 个参数，而 SelfAPR [22] 有 220M 个参数），因为我们的目标是精确衡量回译带来的改进。需要指出的是，这些实验结果并不代表最先进的修复效果 [36]。我们认为，将原始回译循环应用于大型语言模型，也会像我们的实验所展示的那样带来显著益处。然而，这需要超出我们实验室能力范围的计算资源。



(a) QuixBugs



(b) Defects4J

图 2：各模型在两个测试基准上沿束累计正确修复的缺陷数量。MUFIN 不仅总共修复了更多缺陷，而且在整个束上表现一致。

RQ1 的回答： RQ1 基于精心设计的方案，旨在衡量回译的影响。MUFIN 的回译训练使模型能够正确修复 +12 （Defects4J）和 +4 （QuixBugs）中的更多错误，优于基线模型。由于最终模型与基线模型相同，其有效性的提升可归因于回译训练。回译为模型提供了更多训练样本，从而提升了模型在测试数据集上的泛化能力。

4.2 RQ2 结果（批评者）

在 RQ2 中，我们研究 MUFIN 中评论器设计的影响。表 2 展示了各评论器在两个数据集 QuixBugs 和 Defects4J 上的效果。表格内容如下：第一组元列提供用于训练模型的方法和评论器的信息；第二组元列给出各阶段使用的训练样本数量，其中第 3 列为 MUFIN 回译过程中通过自监督生成的样本数量。第三组元列包含修复效果结果。对于每个单元格 X/Y，X 表示正确修复的缺陷数量（即人工编写的测试通过时），

其中，Y 表示可能已修复的错误数量（即人工编写的测试通过时）。最后一个元列展示了补丁的可编译性结果。每个单元格中的 X% 表示给定数据集中可编译补丁占所有生成补丁的百分比。

修复效果。总体而言，所有 MUFIN 模型的修复效果都优于基线模型。与基线模型相比，MUFIN 模型能够正确修复更多 Defects4J 错误：+8（无评审器）、+12（评审器 = 编译器）和 +7（评审器 = 测试），并为更多错误找到合理的补丁。QuixBugs 上也观察到了相同的情况：与初始模型相比，MUFIN 模型能够正确修复更多错误：+3（无评审器）、+4（评审器 = 编译器）和 +3（评审器 = 测试）。无论是否使用评审器，MUFIN 反向翻译中的微调过程都很有帮助。即使不使用评审器，模型性能也有所提升，这一点充分证明了这一点。

同时，我们观察到评审器会影响模型最终的修复效果。使用编译器 评审器训练的模型效果最好，共正确修复了 28 个 Defects4J 错误。

现在来看评论器导致的训练样本减少情况。如表 2 所示，不使用评论器的 MUFIN 会生成 679,140 个用于微调的自监督样本。编译器 评论器只保留三分之一的数据，因此得到 197,959 个训练样本。最后可以看到，测试 评论器的筛选极为严格，最终仅得到 21,196 个训练样本。由于使用测试 评论器的 MUFIN 表现不如不使用评论器的版本，这说明筛选过于严苛，导致自监督样本数量不足，无法发挥作用。这与我们最初的预期相反：测试 评论器筛选出的数据质量很高，因为在我们考虑的测试套件完善的项目中，缺陷可以百分之百确定是缺陷，而修复也很可能确实是修复。

我们注意到，尽管编译器等评论器的筛选条件较为宽松，但这并不妨碍破坏器生成有价值、适合用于模型微调的有趣缺陷。使用编译器 评论器训练的破坏器并非只会引入无关紧要的缺陷。例如，来看清单 3 中 MUFIN（编译器）的破坏器引入的一个缺陷。在这个例子中，有问题的补丁引入了一条全新的语句，其中包含一个在输入上下文中不可用的方法调用。这个缺陷能够通过编译。即使没有通过失败的测试用例得到正式验证，由于它在程序中引入了非预期行为，仍可将其视为功能性缺陷。

总体而言，评审器的严格程度（即评审器在执行样本质量要求时有多严格）与为反向翻译生成的训练样本数量之间存在明显的权衡。*compiler* 评审器在这方面表现最佳。

补丁可编译性。表 2 最后一列显示，所有 MUFIN 模型的补丁可编译率都高于初始基线模型。例如，在所有错误中，MUFIN（编译器）和 MUFIN（无评审器）生成的可编译补丁均占 17%，波束大小为 100。显然，尽管将 MUFIN（测试）用作评审器提高了相对于基线模型的修复效果，但并未改善补丁可编译性。在 QuixBugs 中，MUFIN（测试）的补丁可编译率甚至低于基线模型。

这些结果进一步凸显了评审器严格程度与可用于微调的样本数量之间的权衡关系；评审器严格程度可作为样本质量的代理指标。MUFIN（无评审器）和 MUFIN（编译器）生成的样本数量更多，能够提升生成补丁的语法质量。同样的道理也解释了为什么无评审器方案在可编译性方面略胜一筹，因为它产生的训练样本多出 3 倍。

评审器执行时间。不同的评审器筛选样本所需的时间各不相同。我们注意到，与使用*tests*评审器相比，使用*compiler*评审器的计算开销要低得多，因为前者不仅会编译，还会执行测试套件。为解决这一问题，未来的工作可以优先采用轻量、快速的纯静态评审器。

RQ2 的回答： 我们的结果表明，评审器设计与最终修复模型的有效性密切相关。对于 MUFIN，最佳评审器是*compiler* 评审器；它筛选出生成数据中质量最高的三分之一，从而使神经修复器能够正确修复 28 个 Defects4J 缺陷，比未经微调的基线多修复 12 个缺陷。我们清楚地观察到并详细讨论了评审器严格程度与可用训练样本数量之间的权衡。

5 讨论

5.1 破坏器的再利用

尽管本文旨在修复有缺陷的程序，但缺陷也是多项其他软件工程任务的核心组成部分。许多此类任务的方法都需要大量可执行的缺陷样本，用于构建或执行。例如，故障定位 [37]和缺陷检测 [38]的实验研究都需要大量缺陷样本，才能获得可靠的结果。

收集如此大量的可执行缺陷样本难以规模化，因为这需要投入大量人力。因此，能够生成缺陷的神经模型是解决这一问题的一种可能方案：将破坏器模型应用于少数正确项目中的多个位置，就能获得大量可执行的缺陷。就此而言，MUFIN 的破坏器模型具有内在价值。按照设计，MUFIN 的破坏器模型经过训练，能够根据评判器的评价生成高质量缺陷。

我们的实验结果证明了 MUFIN 破坏器生成缺陷的高产能力。在最宽松的设置下，MUFIN（无评判器）仅凭破坏器就生成了 678,540 个缺陷，平均每个项目植入约 12,117 个缺陷。此外，使用仅保留能够成功编译且单元测试失败的样本的评判器（MUFIN（测试））时，我们得到的缺陷均可保证能够构建，且至少有一项单元测试失败。我们的数据集包含 21,180 个此类缺陷，其数量比文献中人工整理的数据集中可执行缺陷的数量高出一个数量级。

代码清单 4 展示了 MUFIN 破坏模型生成的一个错误示例。这里，破坏模型引入了一个功能性错误，导致至少一个单元测试失败。具体来说，破坏模型过早关闭了输出流，从而释放了所有

表 2：MUFIN 在两个测试数据集上搭配三种不同评审器时的效果。使用 MUFIN 回译进行微调始终能提升修复器的效果。评审器的严格程度与效果之间存在权衡。编译器 评审器效果最佳。

方法	# 训练样本数		修复		补丁可编译性	
	初始化	自监督	QuixBugs（39 个缺陷）	D4J（428 个缺陷）	QuixBugs（39 个错误）	D4J（422 个错误）
基线模型	3,942,935	-	2/4	16/43	17.64%	12.73%
MUFIN(无评论者)	3,942,935	679,140	5/7	24/57	**21.33%**	**17.07%**
MUFIN (编译器)	3,942,935	197,959	**6/7**	**28/62**	**21.33%**	17.00%
MUFIN（测试）	3,942,935	21,196	5/8	23/51	16.54%	13.32%

代码清单 3：MUFIN（编译器）破坏器生成的错误示例。这里，破坏模型合成并引入了一条全新的语句，改变了程序的预期行为。

```
--- a/correct.java+++ b/buggy.java如果 (variableType == 变量_任务) {+
response.setUrl(urlBuilder.buildUrl(RestUrls.URL_任务,
task.getId()));restVar.setValueUrl(urlBuilder.buildUrl(RestUrls.URL_任务_变量_数据, id, na
me));
```

清单 4：MUFIN（测试）破坏器生成的错误示例。该破坏器过早关闭输出流，导致后续调用 flush 操作时抛出 IOException。

```
--- a/correct.java
+++ b/buggy.java
@@ -866,12 +866,11 @@
  _flushBuffer();
  if (flushStream) {
+  out.close();
   _out.flush();
  }
```

通过引入 *close()* 方法调用，在 *flush()* 方法调用之前关闭与其相关的系统资源。当执行 *flush()* 方法调用时，由于流已关闭，会抛出 *IOException* 。

此外，也可以将变异测试 [39] 视为依赖于某个破坏器。为此，该破坏器可以作为即插即用组件用于变异测试基础设施，这与神经变异相关研究的思路一致 [24, 40]。

综上所述，MUFIN 的用途不止于程序修复；最初作为副产品的破坏器也可以成为未来研究的宝贵资源。我们向科学界公开破坏器模型和生成的缺陷。所有缺陷均可构建并运行，且至少包含一个失败测试，可用于自动化软件工程实验。

5.2 有效性威胁

我们将实现中可能存在的错误以及人工补丁分析中的错误，视为对结果有效性的内部威胁。

为应对这些问题，我们公开提供实现代码和实验材料。

外部威胁之一是研究仅关注一种编程语言（Java）。为减轻外部威胁，我们在两个成熟的 Java 程序修复基准上进行了评估。原则上，我们的实验结果应可推广至任意编程语言和修复基准。

5.3 未来工作

在 RQ2 中，我们研究了评审器设计对修复器模型有效性的影响。不过，仍有一些相关问题尚未解答。首先，评审器设计在多大程度上取决于初始修复器／破坏器模型的有效性，仍不得而知。其次，MUFIN 回译中种子数据集如何影响修复器模型的泛化能力，也尚未得到解答。我们的直觉是，种子数据集越大，同时越多样、越贴近实际，使用回译进行微调的效果就越显著。第三，仍需研究 MUFIN 回译的进一步迭代是否以及如何改进神经修复器模型，以及这一过程何时、如何收敛。

6 相关工作

6.1 自动化程序修复

基于启发式的方法 [3–6] 通过先根据预设启发式规则计算修改的搜索空间，再使用给定的测试集运行暂定修补后的程序，从而生成并验证候选补丁。能使修改后的程序通过所有测试的候选补丁被视为正确。

基于约束的方法 [7–9] 采用了不同的策略。首先，它们会找出修复缺陷必须满足的约束。然后，在这些已识别约束的指导下，程序合成技术会生成补丁。

基于学习的方法 [10–17, 22] 利用有缺陷样本与正确样本对来学习模型，并据此生成补丁。通常会采用深度学习技术来构建补丁生成模型。MUFIN 与传统的监督式学习修复方法 [10–14, 41] 不同之处在于，它采用自监督学习。

6.2 代码上的自监督学习

自监督学习是一种机器学习范式，其做法是将未配对数据转换为配对数据，以监督方式训练机器学习模型。这是解决监督训练数据获取成本高这一问题的方法。近年来，自监督学习在自然语言处理（NLP）领域日益流行，并已成功应用于学习词语表示 [42, 43], 、预训练语言模型 [44–46] 以及预训练代码语言模型 [30, 47, 48]。

自监督学习较少应用于编码任务。Kommrusch 等人 [49] 通过对完整和不完整的证明进行自监督，在迭代训练过程中学习证明程序等价性。Ni 等人 [50] 采用类似方法，为数学推理问题生成并采样正确和部分正确的代码解答。

自监督学习可以应用于自动程序修复（APR）。Loriot 等人 [51]通过在人工注入的违规项上训练模型，学习修复 Checkstyle 违规。Yasunaga 等人 [17]提出了一种基于人工缺陷的编译错误自监督处理流程。这两种技术的扰动流程作用于字符和词元层级，且并不旨在引入功能性缺陷，这与 MUFIN 的目标有关键区别。Vasic 等人 [52] 通过合成替换变量名，为用于定位和修复变量误用错误的联合模型生成训练数据集。Allamanis 等人 [16]使用人工生成的四类不同缺陷训练缺陷检测与修复模型。Ye 等人 [22] 提出 SelfAPR：一种利用执行诊断信息训练神经模型的自监督框架。这些工作的关键区别在于，它们都没有像 MUFIN 那样使用回译。

6.3 代码回译

回译是一种用于神经机器翻译的自监督学习技术，它通过将文本从源语言翻译成目标语言，再译回源语言来生成合成训练数据 [23, 53]。Rozière 等人 [25, 54] 将回译应用于代码翻译任务，而非程序修复。Wang 等人 [55] 在其数据增强策略中使用回译，以发现输入源（代码和注释）与已修复错误之间的因果关系。他们的目标是提高 APR 模型预测的可解释性。Ahmad 等人 [56]也在代码翻译任务中使用回译，其目标是从目标语言译至自然语言，再译至源语言，而不是从目标语言译至源语言，再译回目标语言。

Drain 等人 [57] 使用回译训练修复模型。这项工作与 MUFIN 的主要区别在于，他们没有使用评论器，也没有考虑 Java 中复杂的功能性错误。

Yasunaga 和 Liang [15] 提出了用于修复编译错误的 Break-It-Fix-It（BIFI），并通过引入评论器对其进行扩展。BIFI 与 MUFIN 的主要区别如下。首先最重要的是，BIFI 专注于简单的编译错误，其补丁大多只涉及单个字符的改动，例如缺少分号。相比之下，MUFIN 修复的是导致测试套件失败的功能性错误，其补丁会修改多个词元并改变 AST 结构。我们的实验首次证明了回译可用于字符改动不止几个的场景。

7 结论

本文介绍了 MUFIN，这是一种用于 Java 自动化程序修复的新型自监督方法。MUFIN 引入了一个专为功能性缺陷设计的新型回译循环。为展示 MUFIN 的实用性，我们在两个广泛认可的 Java 程序修复基准上进行了实验。结果表明，与基线模型相比，MUFIN 提高了补丁质量，无论是在修复效果还是补丁可编译性方面。此外，我们还证明，并非所有评判器都同样有效。基于编译结果的编译器评判器带来了最佳且最稳定的结果：它能够为神经网络提供高价值的训练样本，同时又不会过于严格，因此回译循环中仍可使用数量可观的自监督训练样本。

我们注意到，自监督神经程序修复仍是一个相对缺乏探索的研究方向。然而，我们的结果表明，这一范式在改进各类模型方面具有巨大潜力。为此，我们公开了实现代码和实验材料，以推动该领域的后续研究。

参考文献

[1] I. Vessey，“计算机程序调试中的专业知识：过程分析”，国际人机研究期刊，第23卷，第5期，第459–494页，1985年。[2] C. L. Goues、M. Pradel 和 A. Roychoudhury，“自动化程序修复”，*ACM* 通讯，第62卷，第12期，第56–65页，2019年。[3] S. Forrest、T. Nguyen、W. Weimer 和 C. Le Goues，“一种自动化软件修复的遗传编程方法”，载于第*11*届遗传与进化计算年度会议论文集，2009年，第947–954页。[4] D. Kim、J. Nam、J. Song 和 S. Kim，“从人工编写的补丁中学习自动补丁生成”，载于*2013*年第*35*届国际软件工程会议（*ICSE*）。IEEE，2013年，第802–811页。[5] K. Liu、A. Koyuncu、D. Kim 和 T. F. Bissyandé，“Tbar：重新审视基于模板的自动化程序修复”，载于第*28*届 *ACM SIGSOFT* 国际软件测试与分析研讨会论文集，2019年，第31–42页。[6] B. Cornu、T. Durieux、L. Seinturier 和 M. Monperrus，“Npefix：Java 中空指针异常的自动运行时修复”，*arXiv* 预印本 *arXiv:1512 07423*，2015年。[7] H. D. T. Nguyen、D. Qi、A. Roychoudhury 和 S. Chandra，“Semfix：通过语义分析进行程序修复”，载于*2013*年第*35*届国际软件工程会议（*ICSE*）。IEEE，2013年，第772–781页。[8] J. Xuan、M. Martinez、F. Demarco、M. Clement、S. L. Marcote、T. Durieux、D. Le Berre 和 M. Monperrus，“Nopol：自动修复 Java 程序中的条件语句错误”，*IEEE* 软件工程汇刊，第43卷，第1期，第34–55页，2016年。[9] S. Mechtaev、J. Yi 和 A. Roychoudhury，“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，2016年，第691–701页。[10] R. Gupta、S. Pal、A. Kanade 和 S. Shevade，“Deepfix：利用深度学习修复常见的 C 语言错误”，载于第三十一届 *AAAI* 人工智能会议，2017年。[11] Z. Chen、S. J. Kommrusch、M. Tufano、L.-N. Pouchet、D. Poshyvanyk 和 M. Monperrus，“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年。[12] T. Lutellier、H. V. Pham、L. Pang、Y. Li、M. Wei 和 L. Tan，“Coconut：结合上下文感知神经机器翻译模型并使用集成方法进行程序修复”，载于第*29*届 *ACM SIGSOFT* 国际软件测试与分析研讨会论文集，2020年，第101–114页。[13] H. Ye 、 M. Martinez 和 M. Monperrus，“ 基于执行反向传播的神经程序修复 ”，载于第*44*届国际会议论文集，关于

软件工程，2022，第 1506–1518 页。[14] Z. Chen、S. J. Kommrusch 和 M. Monperrus，《利用神经迁移学习修复 C 代码中的安全漏洞》，IEEE《软件工程汇刊》，2022。[15] M. Yasunaga 和 P. Liang，《Break-it-fix-it：面向程序修复的无监督学习》，发表于国际机器学习会议（ICML），2021。[16] M. Allamanis、H. Jackson-Flux 和 M. Brockschmidt，《自监督缺陷检测与修复》，发表于 NeurIPS，2021。[17] M. Yasunaga 和 P. Liang，《基于图的自监督程序修复：利用诊断反馈》，发表于国际机器学习会议。PMLR，2020，第 10 799–10 808 页。[18] R. Just、D. Jalali 和 M. D. Ernst，《Defects4j：用于对 Java 程序开展受控测试研究的现有故障数据库》，发表于《2014 年国际软件测试与分析研讨会论文集》，2014，第 437–440 页。[19] F. Madeiral、S. Urli、M. Maia 和 M. Monperrus，《Bears：用于自动程序修复研究的可扩展 Java 缺陷基准》，发表于《2019 年 IEEE 第 26 届国际软件分析、演化与再工程会议（SANER）》。IEEE，2019，第 468–478 页。[20] D. Lin、J. Koppel、A. Chen 和 A. Solar-Lezama，《Quixbugs：基于 Quixey 挑战的多语言程序修复基准集》，发表于《2017 年 ACM SIGPLAN 系统、编程语言与应用：软件服务于人类国际会议论文集》，2017，第 55–56 页。[21] R. K. Saha、Y. Lyu、W. Lam、H. Yoshida 和 M. R. Prasad，《Bugs.jar：大规模、多样化的真实世界 Java 缺陷数据集》，发表于《第 15 届国际软件仓库挖掘会议论文集》，2018，第 10–13 页。[22] H. Ye、M. Martinez、X. Luo、T. Zhang 和 M. Monperrus，《Selfapr：利用测试执行诊断进行自监督程序修复》，发表于《第 37 届 IEEE/ACM 自动化软件工程国际会议》，2022，第 1–13 页。[23] G. Lample、M. Ott、A. Conneau、L. Denoyer 和 M. Ranzato，《基于短语与神经网络的无监督机器翻译》，发表于 EMNLP，2018。[24] M. Tufano、C. Watson、G. Bavota、M. Di Penta、M. White 和 D. Poshyvanyk，《从缺陷修复中学习如何变异源代码》，发表于《2019 年 IEEE 国际软件维护与演化会议（ICSME）》。IEEE，2019，第 301–312 页。[25] B. Roziere、J. Zhang、F. Charton、M. Harman、G. Synnaeve 和 G. Lample，《利用自动化单元测试进行无监督代码翻译》，发表于国际学习表征会议，2021。[26] J. Campos、A. Riboira、A. Perez 和 R. Abreu，《Gzoltar：用于测试与调试的 Eclipse 插件》，发表于《第 27 届 IEEE/ACM 自动化软件工程国际会议论文集》，2012，第 378–381 页。[27] A. Silva、M. Martinez、B. Danglot、D. Ginelli 和 M. Monperrus，《Flacoco：基于工业级覆盖率的 Java 故障定位》，arXiv 预印本 arXiv:2111.12513，2021。[28] R. Pawlak、M. Monperrus、N. Petitprez、C. Noguera 和 L. Seinturier，《Spoon：用于实现 Java 源代码分析与转换的库》，《软件：实践与经验》，第 46 卷，第 9 期，第 1155–1179 页，2016。[29] C. Raffel、N. Shazeer、A. Roberts、K. Lee、S. Narang、M. Matena、Y. Zhou、W. Li、P. J. Liu 等，《探索统一文本到文本转换器的迁移学习极限》，《机器学习研究期刊》，第 21 卷，第 140 期，第 1–67 页，2020。[30] W. Ahmad、S. Chakraborty、B. Ray 和 K.-W. Chang，《面向程序理解与生成的统一预训练》，发表于《2021 年北美计算语言学协会分会会议：人类语言技术论文集》。计算语言学协会，2021 年 6 月，第 2655–2668 页。[31] I. Loshchilov 和 F. Hutter，《解耦权重衰减正则化》，发表于国际学习表征会议，2019。[32] D. P. Kingma 和 J. Ba，《Adam：一种随机优化方法》，发表于 ICLR（海报），2015。[33] Q. Zhu、Z. Sun、Y.-a. Xiao、W. Zhang、K. Yuan、Y. Xiong 和 L. Zhang，《用于神经程序修复的语法引导编辑解码器》，发表于《第 29 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》，2021，第 341–353 页。[34] S. Saha 等，《利用演化实现多代码块程序修复》，发表于《2019 年 IEEE/ACM 第 41 届国际软件工程会议（ICSE）》。IEEE，2019，第 13–24 页。[35] K. Liu、A. Koyuncu、T. F. Bissyandé、D. Kim、J. Klein 和 Y. Le Traon，《无法修复未能发现的问题！对自动化程序修复系统基准测试中故障定位偏差的研究》，发表于《2019 年第 12 届 IEEE 软件测试、验证与确认会议（ICST）》。IEEE，2019，第 102–113 页。

[36] N. Jiang、K. Liu、T. Lutellier与L. Tan，“代码语言模型对自动化程序修复的影响”，arXiv预印本 arXiv:2302.05020，2023。[37] Y. Li、S. Wang与T. Nguyen，“基于代码覆盖率表示学习的故障定位”，载于2021年IEEE/ACM第43届软件工程国际会议（ICSE）。IEEE，2021，第661–673页。[38] M. Pradel与K. Sen，“DeepBugs：一种基于名称的缺陷检测学习方法”，《ACM编程语言学报》，第2卷，第OOPSLA期，第1–25页，2018。[39] Y. Jia与M. Harman，“变异测试发展的分析与综述”，《IEEE软件工程汇刊》，第37卷，678，2010 5，649no. pp. – 。[40] M. Tufano、J. Kimko、S. Wang、C. Watson、G. Bavota、M. Di Penta与D. Poshyvanyk，“DeepMutation：一种神经变异工具”，载于ACM/IEEE第42届软件工程国际会议论文集：配套论文集，2020，第29–32页。[41] N. Jiang、T. Lutellier与L. Tan，“CURE：面向自动程序修复的代码感知神经机器翻译”，载于2021年IEEE/ACM第43届软件工程国际会议（ICSE）。IEEE，2021，第1161–1173页。[42] T. Mikolov、K. Chen、G. Corrado与J. Dean，“向量空间中词表示的高效估计”，载于第一届学习表征国际会议，ICLR 2013，美国亚利桑那州斯科茨代尔，2013年5月2–4日，研讨会论文集，2013。[43] J. Pennington、R. Socher与C. D. Manning，“GloVe：用于词表示的全局向量”，载于2014年自然语言处理经验方法会议（EMNLP）论文集，2014，第1532–1543页。[44] W. L. Taylor，““完形填空程序”：一种衡量可读性的新工具”，《新闻学季刊》，第30卷，第4期，第415–433页，1953。[45] J. Devlin、M.-W. Chang、K. Lee与K. Toutanova，“BERT：面向语言理解的深度双向Transformer预训练”，载于2019年计算语言学协会北美分会会议论文集：人类语言技术，第1卷（长文与短文），2019，第4171–4186页。[46] Y. Liu、M. Ott、N. Goyal、J. Du、M. Joshi、D. Chen、O. Levy、M. Lewis、L. Zettlemoyer与V. Stoyanov，“RoBERTa：一种经过稳健优化的BERT预训练方法”，arXiv预印本 arXiv:1907.11692，2019。[47] Y. Wang、W. Wang、S. Joty与S. C. Hoi，“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，2021，第8696–8708页。[48] Z. Feng、D. Guo、D. Tang、N. Duan、X. Feng、M. Gong、L. Shou、B. Qin、T. Liu、D. Jiang等，“CodeBERT：面向编程语言与自然语言的预训练模型”，载于计算语言学协会研究成果：EMNLP 2020，2020，第1536–1547页。[49] S. Kommrusch、M. Monperrus与L.-N. Pouchet，“通过语义保持重写规则证明程序等价性的自监督学习”，arXiv预印本 arXiv:2109.10476，2021。[50] A. Ni、J. P. Inala、C. Wang、A. Polozov、C. Meek、D. Radev与J. Gao，“从自采样的正确与部分正确解答中学习数学推理”，载于学习表征国际会议。[51] B. Loriot、F. Madeiral与M. Monperrus，“Styler：学习格式规范以修复Checkstyle违规”，《实证软件工程》，第27卷，第6期，第1–36页，2022。[52] M. Vasic、A. Kanade、P. Maniatis、D. Bieber与R. Singh，“通过联合学习定位与修复实现神经程序修复”，载于学习表征国际会议，2018。[53] S. Edunov、M. Ott、M. Auli与D. Grangier，“大规模理解回译”，载于2018年自然语言处理经验方法会议论文集，2018，第489–500页。[54] B. Roziere、M.-A. Lachaux、L. Chanussot与G. Lample，“编程语言的无监督翻译”，《神经信息处理系统进展》，第33卷，第20601–20611页，2020。[55] J. Wang、S. Si、Z. Zhu、X. Qu、Z. Hong与J. Xiao，“利用因果推断实现可解释的自动程序修复”，载于2022年神经网络国际联合会议（IJCNN）。IEEE，2022，第1–6页。[56] W. U. Ahmad、S. Chakraborty、B. Ray与K.-W. Chang，“先摘要再生成以进行回译：编程语言的无监督翻译”，arXiv预印本 arXiv:2205.11116，2022。[57] D. Drain、C. B. Clement、G. Serrato与N. Sundaresan，“DeepDebug：利用栈跟踪、回译与代码骨架修复Python缺陷”，arXiv预印本 arXiv:2105.09352，2021。