论基于机器学习的程序翻译：风险与前景

Aniketh Malyala†Silver Creek High School 圣何塞，加利福尼亚州，美国
aniketh.malyala@gmail.com

Katelyn Zhou†Silver Creek High School 圣何塞，加利福尼亚州，美国
zhoukatelyn@gmail.com

Baishakhi Ray哥伦比亚大学 纽约，纽约州，美国
rayb@cs.columbia.edu

Saikat Chakraborty微软研究院 雷德蒙德，华盛顿州，美国
saikatc@microsoft.com

arXiv:2302.10812v1 [cs.PL] 21 Feb 2023

摘要——随着新型先进编程语言的出现，将遗留软件迁移到新的编程语言中变得势在必行。即使缺乏规模足够大且可靠的并行源代码语料库，无监督的基于机器学习的程序翻译也可能在此类迁移中发挥关键作用。然而，由于这类翻译器具有统计特性，其表现远非完美。本文研究了无监督程序翻译器及其失效的情形和原因。通过对这类失效进行深入的错误分析，我们发现，翻译器失效的情况遵循若干特定模式。基于这一发现，我们开发了一种基于规则的程序变异引擎：当输入代码符合特定模式时，该引擎会对其进行预处理；当输出符合某些模式时，则对其进行后处理。我们表明，我们的代码处理工具与程序翻译器相结合，可以构成一种混合程序翻译器，并显著提升当前最先进水平。展望未来，我们希望开发一种端到端的程序翻译工具，通过预处理和后处理步骤，将编程领域知识嵌入基于机器学习的翻译流水线。

索引词—**代码生成、代码翻译、程序转换**

I. 引言

在当今的软件开发生态中，编程语言（PL）正在快速演进，既有新语言的出现，也有现有语言新特性的加入。在过去几年中，Go、Rust、Swift、TypeScript、Python3 等许多语言逐渐流行起来。要跟上这样的发展变化往往颇具挑战——接受过一种编程语言训练的开发者很难适应新的范式 [1]。

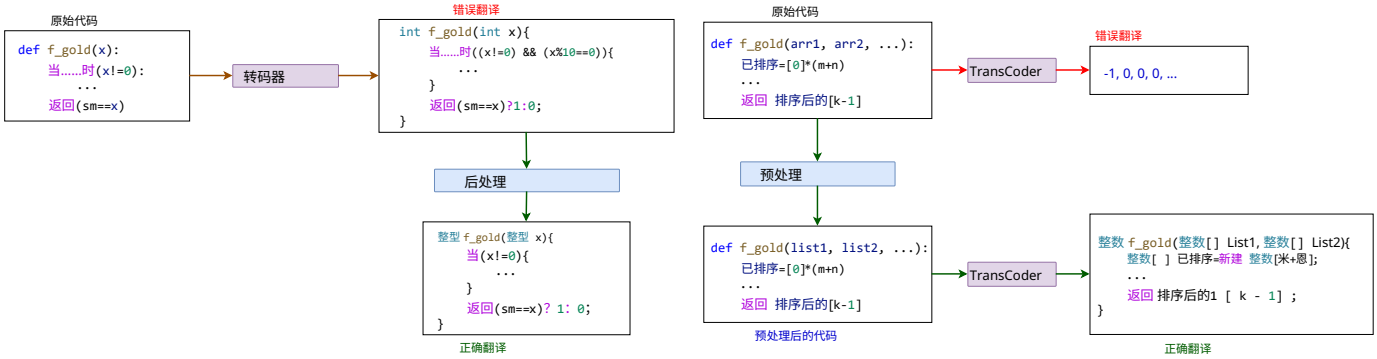
大量遗留软件是用 COBOL、Fortran 等旧语言编写的。维护这些软件颇具挑战，因为如今的开发人员需要深入了解这些过时的语言 [2]–[6]。各类组织一直投入大量资源，将其遗留代码迁移到较新的编程语言。例如，2012年，澳大利亚联邦银行在随后五年内投入10亿澳元，将其核心银行平台迁移[1]。瑞典北欧银行也于2020年启动了迁移工作。虽然迁移到较新的编程语言最终能够节省资金，但迁移所需的投入可能更高，因为不同编程语言遵循的编程理念截然不同（例如，面向对象与函数式）。

为了解决这些问题，研究人员提出了自动化工具，用于将以一种高级语言（例如，Java）编写的程序转换为另一种高级语言（例如，Python），这类工具通常称为转译器 或转译器 [7]，[8]。传统上，转译器采用基于规则的翻译方式 [9]–[11]。以源语言编写的程序会被表示为抽象语法树，然后通过手写规则（也称为模板）将其翻译为目标语言。这种手动驱动规则的翻译方式难以扩展，尤其是在涉及外部库和 API 的情况下。此外，当两种语言的结构差异很大时（例如，函数式语言 Haskell 与过程式面向对象语言 Java），编写转换规则并不总是可行。最后，使用此类手动规则生成的程序通常缺乏可读性。

为了解决这些问题，研究人员提出了基于机器学习（ML）的转译器。这类转译器通过学习两种高级编程语言之间的统计对应关系，让机器学习模型在这两种语言之间进行转换 [12]–[15]。然而，获取有意义且对齐的语言语料库颇具挑战 [16]，[17]。为此，Roziere 等人 [8] 提出了基于无监督学习的方法 TransCoder，通过回译学习编程语言之间的对应关系 [18]。程序的源语言首先由正向翻译器转换为目标语言。随后，生成的目标程序再由反向翻译器转换回源语言。通过联合优化，这些正向与反向翻译器对能够分别学习源语言和目标语言之间的对应关系，而无需显式对齐的语料库。

† 贡献相同。

[1]https://www.reuters.com/article/us-usa-banks-cobol/banks-scramble-to-fix-old-systems-as-it-cowboys-ride-into-sunset- idUSKBN17C0D8



(a) **后处理：** TransCoder 生成的代码有 (b) **预处理：** TransCoder 无法翻译 Python 多出的一个错误 `x%10==0` 条件；后处理正确地移除了 `array` 参数。预处理将 `arr` 变量转换为 `list` 后，TransCoder 就能正确翻译。

图1： **示例说明：Python 到 Java 的翻译。**

事实证明，无监督学习的表现可以优于此前所有方法。然而，由于基于 TransCoder 的模型完全受语言的统计特性驱动，因此无法保证生成代码的语法或语义准确性。图 1 展示了一个示例。虽然 TransCoder 模型几乎正确地转换了图 1a 中的输入代码，但转换后的 Java 方法多出一个条件子句，`x % 10 == 0`。知识渊博的开发者可以进一步修改几乎正确转换的代码，以提高准确性，尤其是在能够识别模型常犯的错误模式时。

假设1。尽管“无监督”翻译器并不完美，但如果我们了解模型常见的错误模式（即“盲点”），就可以对其结果进行后处理。

此外，由于这些模型是以临时、无监督的方式训练的，因此它们并未明确学习不同语言组件之间的语法和语义对齐。例如，`while` Java 和 Python 中的循环在语义上等价。然而，`for` 这两种语言中的循环在语义上有所不同——Java `for`循环结构通常包含用于更新循环控制变量的更新表达式；而在 Python 的 `for`循环中，这种功能则有所限制。因此，TransCoder 经常无法将 Java `for` 循环翻译成 Python 循环。

假设2。一旦确定模型的能力局限，我们就可以有系统地改写输入代码，绕过常见的错误生成模式。

在这项试点研究中，我们旨在了解 TransCoder 常见的不足之处，以及如何加以改进。为此，我们选择了 FaceBook AI 发布的一款大型开源无监督程序翻译模型 TransCoder。 [8]，该模型基于 1.28 亿个 GitHub 代码库进行训练，近期备受关注。随后，我们进行了严格的人工研究，以找出 TransCoder 翻译错误的常见情形。我们将这类错误分为两大类——（a）语义错误和（b）语法错误。进一步调查这两类错误后，我们发现，容易出现语义错误的翻译遵循特定且人类可观察的模式，并且可以通过简单的后处理加以修正，这印证了假设1（示例见图 1a）。相比之下，当模型生成语法无效的翻译时，我们发现输入遵循少数几种特定模式，并且可以通过预处理进行输入程序转换来修正（假设2）。图 1b 展示了一个示例。

基于机器学习的代码翻译模型前景广阔。然而，若没有语法或语义指导，我们就无法充分发挥其潜力。作为一项概念验证，我们将此类指导融入基于规则的转换器中，该转换器可以对源代码进行预处理和／或后处理；这些转换器可以与 TransCoder 相结合，构建混合程序翻译器，也称为转译器。我们的初始原型在 Java 到 Python 的翻译任务中，相较于原版基于机器学习的 TransCoder，性能提升了 86%；在 Python 到 Java 的翻译任务中，性能提升了 50%。这表明，利用感知程序属性的技术为机器学习模型提供指导，在程序翻译方面具有巨大潜力。

II. 研究设计

TransCoder 是一种先进且广受欢迎的模型，它利用 GeeksForGeeks 未标注数据集，通过无监督学习实现编程语言翻译。它是一个庞大的基于 Transformer 的模型，使用包含约 280 万个开源仓库的公开 GitHub 代码库进行训练。然而，其报告的准确率仍不理想；TransCoder 的性能通过一种称为计算准确率的指标进行评估，即运行翻译后的程序时，它能否产生与源代码相同的输出。TransCoder 的 Python 到 Java 翻译的计算准确率为 68.7%，而反向翻译的准确率为 56.1%。

为了了解 TransCoder 经常会犯哪类错误，我们选取了 100 个示例，深入分析 TransCoder 生成的翻译结果。两位作者审阅了代码示例并记录发现，随后由另外两位作者进行核实。对于每个案例，所有作者都就潜在错误的类型达成一致。为此，我们总结了 TransCoder 常犯的一些常见错误模式。基于这些发现，我们提出一种结合机器学习与传统基于规则的解决方案的混合技术，以端到端地解决代码翻译问题。

数据集。 Facebook AI 的 GitHub 页面 [19]提供了大量用于测试 TransCoder 模型的数据，这些数据取自 GeeksForGeeks 数据集。提供的测试数据集包含约 280 个文件，分别使用 Python、Java 和 C++ 编写。每个文件都包含一个待翻译的方法 f gold()，以及一个包含测试用例的主方法。我们随机抽取了 50 个测试用例，分别用于分析 Java 到 Python 和 Python 到 Java 的翻译。对于每个测试用例，我们使用 TransCoder 翻译每个文件，分析每次翻译的进展，并标记在多个文件翻译中出现的相似错误及其潜在解决方案。

III. 初步结果

表 I：**TransCoder 中发现的常见错误模式**

	Java 到 Python（Java 到 Python）	**Python 到 Java**（Python 到 Java）
1. 额外上下文	18%	38%
2. 循环转换	12%	0%
3. 类型敏感性	38%	4%
4. 额外约束	0%	50%
5. 其他错误	14%	16%
（大多）正确	22%	18%

根据这项研究，我们确定了 4 种不同的错误类别。表 I 展示了它们的分布。与 Java 到 Python 的翻译（Java 到 Python）相比，Python 到 Java 的翻译（Python 到 Java）成功率略高，为 22%，而非 18%。本节将讨论常见错误模式，以及利用基于模板的预处理和后处理方法修复这些错误的潜在途径。这些百分比是通过计算 Java 到 Python 和 Python 到 Java 中 50 个测试用例里出现所述错误的比例得出的。图 2 展示了这些错误及其可能的解决方案，下文将更详细地描述这些错误。

1. 额外上下文。 该模型的目标是准确翻译一个方法，通常称为焦点方法。然而，焦点方法通常被一个 `main` 方法和测试用例包围。我们将这些额外的周边内容称为“额外上下文”。TransCoder 往往会混淆方法内部和外部的参数，有时还会翻译额外上下文，从而导致代码不正确或难以阅读。9 50 个 Java 到 Python（J2P）示例中有 9 个，以及 50 个 Python 到 Java（P2J）示例中有 19 个受到这一问题的影响。

修复方法。 在这些焦点方法脱离上下文单独翻译后（不含额外上下文），TransCoder 会生成正确的输出。图 2 第 1 行展示了一个示例。虽然焦点方法 `f gold` 被调用，且 `main` 方法仍在上下文中，但 TransCoder 无法生成任何有意义的翻译。不过，移除额外上下文后，翻译准确率显著提高。

在论文的其余部分中，我们将 TransCoder 视为函数翻译器。因此，以下观察到的翻译错误主要是指使用 TransCoder 单独翻译函数时产生的错误。**2. 循环转换**。原版 TransCoder 在翻译复杂的 `for` 循环为 `while` 循环时表现不佳，尤其是在将 Java 翻译为 Python 时。由于 Java `for`循环通常比 Python `for` 循环支持更多功能（例如，循环变量的增量方式不同、变量更多、条件更多），TransCoder 模型难以将 Java 中复杂的 `for` 循环翻译为 Python。50 个样本中有 6 个包含复杂的 for 循环，这些循环都未能生成正确的输出，其中 6 个里有 4 个生成了毫无意义的翻译。

修复方法。我们推测，在将输入传递给 TransCoder 之前，先将 `for` 循环转换为 `while` 循环会有所帮助，因为后者在 Python 和 Java 中语法等价。因此，作为预处理步骤，我们进行了语义保持转换，将 `for` 转换为 `while` 。这一预处理显著改善了全部 6 个错误案例的翻译效果。图 2 第二行展示了一个示例。

3. 类型敏感性。我们发现，TransCoder 对某些类型比较敏感。例如，50 个 Java 到 Python 示例中有 19 个案例将数组用作参数。如图 2 第三行所示，TransCoder 无法翻译这些案例。在 Python 到 Java 的转换中也有类似情况（见图 1b）：当输入焦点方法包含两个或更多名称为 `arr` 的参数时，TransCoder 无法翻译它们。请注意，由于 Python 是一种动态类型语言，我们必须依赖变量名来推断其类型。然而，对应的真实标注 Java 代码证实，预期类型确实是数组。

修复方法。我们探索了一种预处理步骤：在不改变代码语义的前提下，尝试使用等价类型或类。例如，在上述案例中，我们将 Java 代码中所有数组参数引用都改为等价数据类型的 List，因为 Java 数组和 Java List 翻译成 Python 后是相同的。请注意，将数组转换为 List 时，我们不能使用完全相同的数据类型，而必须使用包装类数据类型（int 转为 Integer，double 转为 Double，等等）。这种预处理中的类型转换帮助我们提升了 TransCoder 在全部 19 个案例中的性能。

4. 生成额外约束。Python 到 Java 的翻译中最突出的问题是生成额外的逻辑运算符以

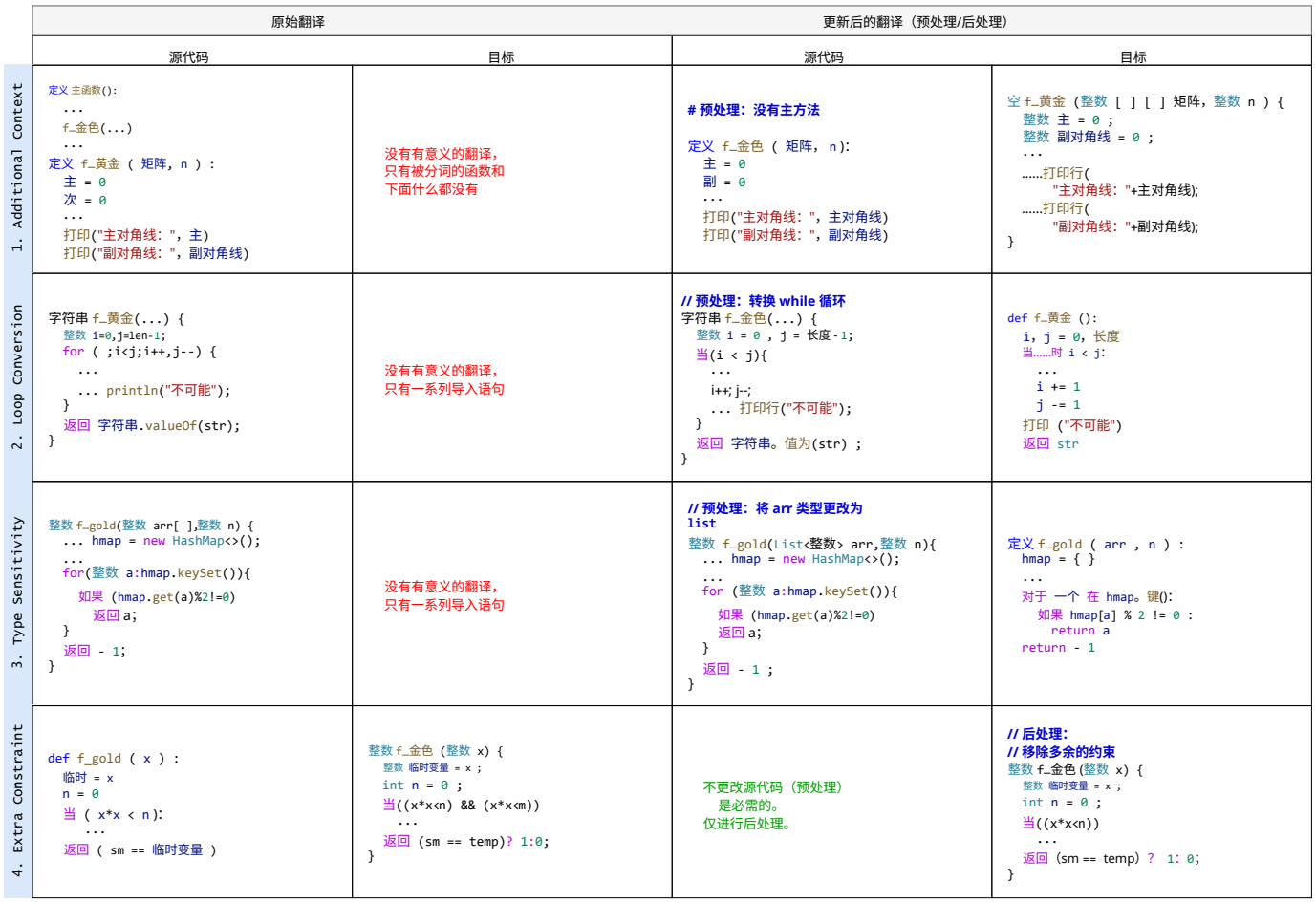


图 2： **检测到的错误模式及其对应的基于规则的解决方案**

及 `if` 和 `else if` 和 `while` 语句。在 50 个示例中，有 25 个存在此类问题。尽管这些额外的逻辑运算符在语法上是有效的，但它们可能会改变代码语义。图 2 的最后一行就是一个示例。

修复。 作为后处理步骤，我们会丢弃源版本中未出现的所有逻辑约束。这是因为我们观察到，尽管模型会添加逻辑约束，但它从不修改原始条件。

总体结果。每种变异的性能均以成功率衡量。我们将“成功”分为两种情况：

1. 如果翻译后的程序无法编译，那么在应用变异后，程序的翻译版本能够编译即为成功。

2. 如果翻译后的程序确实能够编译，但会产生错误，那么当对其应用变异后，程序的运行结果与原程序更加相似时，我们将其视为一次成功。更具体地说，如果能够更容易地将翻译后的代码解释为具有与源代码相同的功能，我们就会将该变异归类为成功。

为了评估每种变异的有效性，我们首先确定每种变异适用于所抽取的哪些测试用例。在分别翻译原始源代码和变异后的源代码后，我们针对每个测试用例将该变异归类为成功或失败。如果某个测试用例适用于多种变异，我们会应用所有可能的组合，以确保每种翻译都取得成功。某种特定变异或规则的成功率，计算方式为成功次数除以其适用的用例数量。我们所识别出的每种变异都具有 100% 的成功率，但仍有一些错误尚未找到可行的变异（表 1 中的其他错误）。

IV. 相关工作

此前已有多项研究探讨了通过机器学习进行编程语言翻译的可能性。然而，几乎所有研究都依赖监督学习 [17], [20]–[24]。尽管这种方法准确，但并不现实，因为很难积累大量带标签且翻译正确的数据集 [17]。

尽管很难找到带标签的数据集，一些研究人员发现，基于一种称为回译的技术来训练模型是有效的 [8],

[16]。由于采用无监督学习，这些模型的能力不受带注释的平行数据数量的限制，因此在程序翻译领域达到了最先进的水平。在这项工作中，我们对这样一种模型 TransCoder 进行了案例研究 [8]。其他研究则探讨了 SMT（统计机器翻译）模型在编程语言翻译中的应用 [12], [14], [25] 。这些研究也得出了与本项目相似的结论：大多数测试用例都存在错误，但只需进行少量修正即可生成正确翻译。这些模型还可以通过更加注重程序分析的方法加以改进，我们的技术也证明了这一点 [14]。

研究人员还提出了用于同语言代码转换的翻译模型，可用于语法修复 [26],[27],语义程序修复 [28]–[30], 重构 [31],[32], 等。最近，研究人员开始提出通用代码转换模型，这些模型基于从 GitHub [33], 收集的开发者编写的代码转换，或基于规则的转换 [22]进行“预训练”。未来，我们计划研究语法和语义修复模型，将其作为预处理和后处理组件。

V. 结论与未来工作

论文摘要。 本文讨论了无监督程序翻译器的陷阱，并阐述了程序属性感知规则的潜力，这些规则可作为预处理/后处理步骤来指导基于机器学习的翻译。我们开发了一种概念验证的同语言程序转换器，用于对 TransCoder 的输入进行预处理，并对其输出进行后处理。我们表明，简单的基于规则的同语言程序转换器能够显著提升程序翻译性能。我们的初步结果以及复现每种变异的详细说明已公开发布于 https://github.com/kzh23/Replication-Package-ICSE-NIER-2023-Unsupervised-ML。基于机器学习的翻译器依赖于“大数据”中蕴含的统计知识，而我们提出将编程领域知识融入翻译流水线。

未来工作。 本文是将基于机器学习的程序翻译与基于程序分析的程序变异相结合的初步尝试。未来，我们旨在构建更复杂、自动化程度更高的程序转换技术。初步结果表明，利用能够感知程序属性的规则来指导机器学习工具，在程序翻译方面具有巨大潜力。未来，我们将研究如何在机器学习流水线中更巧妙地融入此类指导。例如，目前，原版 TransCoder 只能单独翻译方法。这些局限性会妨碍所提技术在现实中的适用性，因为现实中需要翻译的是以遗留语言编写的整个项目。我们还将进一步研究所提技术在低资源语言中的适用性；在这类语言中，我们无法获得足够的样本数据来训练机器学习模型；在这种情况下，基于规则的方法可能需要提供更多指导。

为此，我们设想构建一个可扩展、模块化的端到端系统，将预处理、翻译和后处理步骤结合起来。我们还计划研究将代码编辑模型 [22], [33] 用作预处理工具，以及将程序修复工具 [29], [34], [35]用作后处理步骤，以实现更好的泛化。

致谢

本研究部分由美国国家科学基金会资助（资助编号：SHF- 2107405、SHF-1845893、IIS-2040961），并得到 IBM 和 VMware 的支持。本文所表达的任何观点、研究发现、结论或建议均属于作者，不一定反映美国政府、美国国家科学基金会、IBM 或 VMware 的立场。

参考文献

[1] L. A. Meyerovich 和 A. S. Rabkin，“编程语言采用的实证分析”，载于*2013*年 *ACM SIGPLAN* 面向对象编程系统、语言与应用国际会议论文集，2013年，第 1–18 页。[2] R. J. Kizior、D. Carr 和 P. Halpern，“COBOL 有未来吗？”，载于信息系统教育会议论文集，第 17 卷，第 126 期，2000年。[3] N. Stern，*21* 世纪的 *COBOL*。John Wiley & Sons, Inc.，2007年。[4] H. M. Sneed，“从 COBOL 迁移到 Java”，载于*2010 IEEE*软件维护国际会议。IEEE，2010年，第 1–7 页。[5] J. Pu、Z. Zhang、J. Kang、Y. Xu 和 H. Yang，“利用面向方面理解遗留 COBOL代码”，载于第 *31* 届年度国际计算机软件与应用会议（*COMPSAC 2007*），第 2 卷。IEEE，2007年，第 385–390 页。[6] N. Wilde、M. Buckellew、H. Page 和 V. Rajlich，“非结构化遗留 Fortran代码中特征定位的案例研究”，载于第五届欧洲软件维护与再工程会议论文集。IEEE，2001年，第 68–76 页。[7] R. Kulkarni、A. Chavan 和 A. Hardikar，“转译器及其优势”，《国际计算机科学与信息技术期刊》，第 6 卷，第 2 期，第 1629–1631 页，2015年。[8] B. Roziere、M.-A. Lachaux、L. Chanussot 和 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 20 601–20 611 页，2020年。[9] “Babel 是一个 JavaScript 编译器”，https://babeljs.io/，访问日期：2010-10-12。[10] K. Kimura、A. Sekiguchi、S. Choudhary 和 T. Uehara，“用于摆脱云服务和 API 复杂用法的 JavaScript转译器”，载于*2018* 年第 *25* 届亚太软件工程会议（*APSEC*）。IEEE，2018年，第 69–78 页。[11] “2to3 — 自动将 Python 2 代码翻译为 Python 3”，https:// docs.python.org/3/library/2to3.html，访问日期：2010-10-12。[12] K. Aggarwal、M. Salameh 和 A. Hindle，“使用机器翻译将 Python 2 代码转换为 Python 3 代码”，PeerJ PrePrints，技术报告，2015年。[13] G. Lample、M. Ott、A. Conneau、L. Denoyer 和 M. Ranzato，“基于短语和神经网络的无监督机器翻译”，*arXiv* 预印本 *arXiv:1804.07755*，2018年。[14] A. T. Nguyen、T. T. Nguyen 和 T. N. Nguyen，“用于语言迁移的词法统计机器翻译”，载于 *2013* 年第 *9* 届软件工程基础联合会议论文集，2013年，第 651–654 页。[15] Y. Oda、H. Fudaba、G. Neubig、H. Hata、S. Sakti、T. Toda 和 S. Nakamura，“使用统计机器翻译学习从源代码生成伪代码”，载于 *2015* 年第 *30* 届 *IEEE/ACM* 自动化软件工程国际会议（*ASE*）。IEEE，2015年，第 574–584 页。

[16] W. U. Ahmad、S. Chakraborty、B. Ray 和 K.-W. Chang，“通过摘要和生成进行回译：编程语言的无监督翻译”，*arXiv* 预印本 *arXiv:2205.11116*，2022年。[17] X. Chen、C. Liu 和 D. Song，“用于程序翻译的树到树神经网络”，神经信息处理系统进展，第 31 卷，2018年。[18] S. Edunov、M. Ott、M. Auli 和 D. Grangier，“大规模理解回译”，*arXiv* 预印本 *arXiv:1808 09381*，2018年。[19] Facebookresearch，“Facebookresearch/transcoder：TransCoder 研究项目的公开发布 https://arxiv.org/pdf/2006.03511.pdf。”[在线]。可用地址：https://github.com/facebookresearch/TransCoder [20] W. Ahmad、S. Chakraborty、B. Ray 和 K.-W. Chang，“用于程序理解和生成的统一预训练”，载于*2021* 年北美分会计算语言学协会会议：人类语言技术论文集。在线版：计算语言学协会，2021年 6 月，第 2655–2668 页。 [在线]。可用地址：https://www.aclweb.org/anthology/ 2021 211 . [21] naacl-main。S. Lu、D. Guo、S. Ren、J. Huang、A. Svyatkovskiy、A. Blanco、C. B. Clement、D. Drain、D. Jiang、D. Tang、G. Li、L. Zhou、L. Shou、L. Zhou、M. Tufano、M. Gong、M. Zhou、N. Duan、N. Sundaresan、S. K. Deng、S. Fu 和 S. Liu，“CodeXGLUE：用于代码理解和生成的机器学习基准数据集”，$CoRR$，第 abs/2102.04664 卷，2021年。[22] S. Chakraborty、T. Ahmed、Y. Ding、P. Devanbu 和 B. Ray，“NatGen：通过‘自然化’源代码进行生成式预训练”，载于*2022* 年 *ACM*欧洲软件工程联合会议和软件工程基础研讨会（*ESEC/FSE*）。ACM，2022年。[23] Z. Feng、D. Guo、D. Tang、N. Duan、X. Feng、M. Gong、L. Shou、B. Qin、T. Liu、D. Jiang 和 M. Zhou，“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论文集：*EMNLP 2020*。在线版：计算语言学协会，2020年 11 月，第 1536–1547 页。[24] D. Guo、S. Ren、S. Lu、Z. Feng、D. Tang、S. LIU、L. Zhou、N. Duan、A. Svyatkovskiy、S. Fu、M. Tufano、S. K. Deng、C. Clement、D. Drain、N. Sundaresan、J. Yin、D. Jiang 和 M. Zhou，“Graphcode{bert}：利用数据流进行代码表示的预训练”，载于学习表示国际会议，2021年。[25] S. Karaivanov、V. Raychev 和 M. Vechev，《基于短语的编程语言统计翻译》，载于*2014*年*ACM*新思想、新范式与编程及软件反思国际研讨会论文集，2014年，第173–184页。[26] T. Ahmed、N. R. Ledesma 和 P. Devanbu，《Synfix：使用编译器诊断自动修复语法错误》，*arXiv*预印本 *arXiv:2104.14671*，2021年。[27] ——，《Synshine：改进语法错误修复》，*IEEE*软件工程汇刊，2022年。[28] S. Chakraborty、Y. Ding、M. Allamanis 和 B. Ray，《Codit：使用基于树的神经模型进行代码编辑》，*IEEE*软件工程汇刊，第1–1页，2020年。[29] Z. Chen、S. 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，第47卷第9期，第1943–1959页，2019年。 [在线]。可访问：https://www.cs.wm.edu/∼denys/pubs/seq2seq4repair TSE camerarady.pdf [30] M. Tufano、C. Watson、G. Bavota、M. D. Penta、M. White 和 D. Poshyvanyk，《通过神经机器翻译在真实环境中学习缺陷修复补丁的实证研究》，*ACM*软件工程与方法学汇刊（*TOSEM*），第28卷第4期，第1–29页，2019年。[31] M. Aniche、E. Maziero、R. Durelli 和 V. Durelli，《监督式机器学习算法在预测软件重构方面的有效性》，*IEEE*软件工程汇刊，2020年。[32] A. M. Sheneamer，《一种基于机器学习的软件克隆重构自动顾问》，$IEEE\,Access$，第8卷，第124 978–124 988页，2020年。[33] J. Zhang、S. Panthaplackel、P. Nie、J. J. Li 和 M. Gligoric，《Coditt5：源代码与自然语言编辑的预训练》，*arXiv*预印本 *arXiv:2208.05446*，2022年。[34] H. Ye、M. Martinez 和 M. Monperrus，《基于执行的反向传播神经程序修复》，载于第*44*届国际软件工程会议论文集，2022年，第1506–1518页。[35] M. Yasunaga 和 P. Liang，《Break-it-fix-it：用于程序修复的无监督学习》，载于国际机器学习会议。PMLR，2021年，第11 941–11 952页。 [在线]。可访问：https://arxiv.org/pdf/2106.06600.pdf