基于深度学习的代码生成方法研究进展*

杨泽洲[1], 陈思榕[1], 高翠芸[1], 李振昊[2], 李戈[3], 吕荣聪[4]

[1](哈尔滨工业大学(深圳) 计算机科学与技术学院, 广东 深圳 518055)

[2](华为技术有限公司, 广东 深圳 518129)

[3](北京大学信息科学技术学院, 北京 100871)

[4](香港中文大学 计算机与工程系, 香港 沙田)

通讯作者: 高翠芸, E-mail: gaocuiyun@hit.edu.cn

摘 要: 代码生成(Code Generation), 是指根据自然语言描述生成相关代码片段的任务. 在软件开发过程中, 常常会面临大量重复且技术含量较低的代码编写任务, 代码生成作为最直接辅助开发人员完成编码的工作受到学术界和工业界的广泛关注. 让机器理解用户需求, 自行完成程序编写也一直是软件工程领域重点关注的问题之一. 近年来, 随着深度学习在软件工程领域任务中的不断发展, 尤其是预训练模型的引入使得代码生成任务取得了十分优异的性能. 本文系统梳理了当前基于深度学习的代码生成的相关工作, 并将目前的基于深度学习的代码生成方法分为三类: 基于代码特征的方法, 结合检索的方法以及结合后处理的方法. 第一类是指使用深度学习算法利用代码特征进行代码生成的方法, 第二类和第三类方法依托于第一类方法进行改进. 本文依次对每一类方法的已有研究成果进行了系统的梳理, 总结与点评. 随后本文还汇总分析了已有的代码生成工作中经常使用的语料库与主要的评估方法, 以便于后续研究可以完成合理的实验设计. 最后, 本文对总体内容进行了总结, 并针对未来值得关注的研究方向进行了展望.

关键词: 代码生成; 深度学习; 代码检索; 后处理; 机器翻译

中图法分类号: TP311

中文引用格式: 杨泽洲, 陈思榕, 高翠芸, 李振昊, 李戈, 吕荣聪. 基于深度学习的代码生成技术研究进展. 软件学报, 2021, 32(7). http://www. jos. org. cn/1000-9825/0000. htm

英文引用格式: Yang ZZ, Chen SR, Gao CY, Li ZH, Li G, Lv RC. Deep Learning Based Code Generation Methods: A Literature Review. Ruan Jian Xue Bao/Journal of Software, 2021 (in Chinese). http://www. jos. org. cn/1000-9825/0000. htm

Deep Learning Based Code Generation Methods: A Literature Review

YANG Ze-Zhou[1], CHEN Si-Rong[1], GAO Cui-Yun[1], LI Zhen-Hao[2], LI Ge[3], LV Rong-Cong[4]

[1](School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, 518055, China)

[2](Huawei Technologies Co., Ltd., Shenzhen 518129, China)

[3](School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

[4](Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hongkong, China)

Abstract: Code Generation aims at generating relevant code fragments according to given natural language descriptions. In the process of software development, there exist a large number of repetitive and low-tech code writing tasks, so code generation has received a lot of attention among academia and industry for assisting developers in coding. In fact, it has also been one of the key concerns in the field of software engineering to make machines understand users' requirements and write programs on their own. The recent development of deep

* 基金项目:

收稿时间: 20xx-xx-xx; 修改时间: 20xx-xx-xx; 采用时间: 20xx-xx-xx

learning techniques especially pre-training models make the code generation task achieve promising performance. In this paper, we systematically review the current work on deep learning-based code generation and classify the current deep learning-based code generation methods into three categories: methods based on code features, methods incorporated with retrieval, and methods incorporated with post-processing. The first category refers to the methods that use deep learning algorithms for code generation based on code features, and the second and third categories of methods improve the performance of the methods in the first category. In this paper, the existing research results of each category of methods are systematically reviewed, summarized and commented. The paper then summarizes and analyzes the corpus and the popular evaluation metrics used in the existing code generation work. Finally, the paper summarizes the overall literature review and provides a prospect on future research directions worthy of attention.

Key words: code generation; deep learning; information retrieval; post-processing; machine translation

1 引言

从计算机诞生开始, 虽然编程的形式随着硬件及软件的不断进步而不停迭代, 但是从事计算机技术行业的人员始终与编写代码的任务紧密联系在一起. 因此如何提高软件开发的效率和质量, 一直是软件工程领域的重要问题之一. 这一方面是由于在不同软件开发过程中存在大量相似代码复用的情况, 多次编写重复代码会大大降低开发人员的开发效率以及创造热情; 另一方面, 结构清晰, 功能完备的高质量代码能够使得软件开发过程明晰, 并能够在后期有效降低维护成本.

除了互联网领域的工作人员, 计算思维在当今信息社会对于每一位从业者都必不可少. 政府也在制定相关专门文件推动和规范编程教育发展, 帮助学生理解并建立计算思维[1]. 但事实上, 对于大多数没有经过系统化学习的人而言, 从零开始上手编程并完整完成一段能够实现具体功能的程序是极具挑战的. 编程作为一种手段来完成人们设想的功能, 本质上是一种工具, 但这样的学习门槛使得想法与实际操作之间存在差异, 在一定程度上阻碍了许多具有创意性思维程序的诞生.

程序自动生成方法是一项机器根据用户需求自动生成相应代码的技术. 智能化代码生成具有多元形式, 一般地, 根据实际应用场景以及生成过程所需要的信息, 可以将智能化代码生成分为代码生成(Code Generation)和代码补全(Code Completion)两个任务. 前者是指根据开发人员利用自然语言编写代码的需求(部分会加入输入输出样例), 机器生成特定编程语言的代码片段(部分方法加入后处理环节以保证代码的可执行); 后者则是指在开发人员编写代码过程中, 代码补全算法模型根据已编写代码上文自动理解开发人员的编写意图并补全代码. 根据补全的代码粒度, 又可以将其分为词元级别(token-level)以及行级别(line-level)[2]. 简单来说, 代码生成接收的输入是自然语言描述, 输出是能够一定程度实现自然语言描述功能的代码片段; 而代码补全任务接收的输入是当前代码的上文, 输出的是当前代码的下文. 本文研究的智能化代码生成限定于前者, 即代码生成(Text-to-Code), 旨在根据自然语言描述生成特定编程语言的代码片段. 具体来说, 本任务关注软件实际开发过程中使用的高级编程语言, 如 C++, Java 和 Python 等.

这里给出代码生成问题的数学定义, 给定自然语言描述: $NL = \{nl_1, nl_2, \dots, nl_n\}$, 智能化代码生成模型$f_\theta$, 目的是生成代码片段: $PL = \{pl_1, pl_2, \dots, pl_m\}$, 有如下过程: $f_\theta(\{nl_1, nl_2, \dots, nl_n\}) = \{pl_1, pl_2, \dots, pl_m\}$.

传统的代码生成方法主要依赖于高质量的词汇表, 手工构建的模板和特定领域的语言特性, 要求程序员人工编写逻辑规则, 以便生成程序能够根据其设定规则生成符合逻辑的代码片段. Little 等人[3]提出了一种将关键字(自然语言)转换为 Java 方法调用表达式的算法. Gvero 等人[4]为 Java 编程语言和 API 调用构建了一种概率上下文无关语法, 并提出算法可以将英文输入映射为声明, 最终生成有效的 Java 代码片段. 这样人工提取的方法具有很大的局限性, 不能够适应复杂多变的编程环境, 同时也增加了开发人员编写逻辑规则的开销. 因此随着人工智能和深度学习算法的进步, 自动提取词汇表及特征来生成可执行代码片段也得到长足发展.

事实上, 最近十年来, 利用机器学习和深度学习算法解决计算机各个领域相关问题的研究已成为一种趋势. Hindle 等人[5]发现代码与自然语言在统计学上呈现相似的分布, 代码同自然语言一样是重复的, 有规律的和可预测的. 这样的研究结果为人工智能领域的相关算法模型应用到代码领域相关的问题提供了理论基础, 即 AI(Artificial Intelligence) for SE(Software Engineer). 最近, Chen 等人[6]对代码的自然性及其应用的研究进展

进行了系统的综述, 有助于研究者更好地利用自然语言领域的思想来解决代码领域的问题. 对于代码生成任务而言, 借助机器学习和深度学习算法, 利用数据驱动构建模型, 完成自动代码生成已成为程序自动代码生成任务的解决范式, 称为智能化代码生成. 智能化代码生成能够有效应对不同的开发环境, 提高软件开发的效率和质量, 减轻开发人员的压力, 降低代码编写的门槛.

然而, 代码生成问题在研究时也面临诸多严峻挑战: 首先, 自然语言描述形式多种, 表达多样, 对于同样的函数描述实现可能一百个人就有一百种表达方式. 因此, 能否正确理解自然语言所描述的意图对于代码生成的质量具有决定性作用. 其次, 代码生成本质是生成类任务, 使用到的模型在解码过程中往往伴随着巨大的解空间, 针对复杂问题所生成的代码可能存在无法被执行或对于实际问题没有完全解决的情况, 如何在其中找到正确的符合用户需求的代码仍需要被探索. 因此, 如何更好地利用已有的外部知识库(例如 Stack Overflow 论坛)来提升代码生成模型的效果成为了一个可能的解决方案. 上述这些问题虽随着模型的一步步增大带来的性能提升有所缓解, 但仍未解决. 最后, 目前对于代码生成的质量评估主要采用了自动评估的方式, 评估的指标从机器翻译领域的指标转向基于测试样例的指标, 这一定程度上有助于实际模型性能的评价. 但对于实际场景中的代码生成, 缺乏人工评价的板块, 使得目前代码生成模型落地后的表现距开发人员的期待仍存在一定的差距.

2 研究框架

为了对智能化代码生成相关领域已有的研究工作和取得成果进行系统的梳理, 本文使用 code generation、generating source code、 generating program、 program synthesis 作为关键词在 ACM Digital Library、 IEEE Xplore Digital Library、 Elsevier ScienceDirect、 Springer Link Digital Library、 Google scholar 和 CNKI 在线数据库中进行检索. 基于上述论文数据库中检索出来的相关文献, 我们在人工筛选方法的基础上, 通过分析论文的标题、关键词和摘要去除与代码生成无关的文献. 接着我们递归地对每篇文献进行正向和反向滚雪球搜索[7], 最终选择出与主题直接相关的高质量论文共 53 篇(截止到 2022 年 11 月).

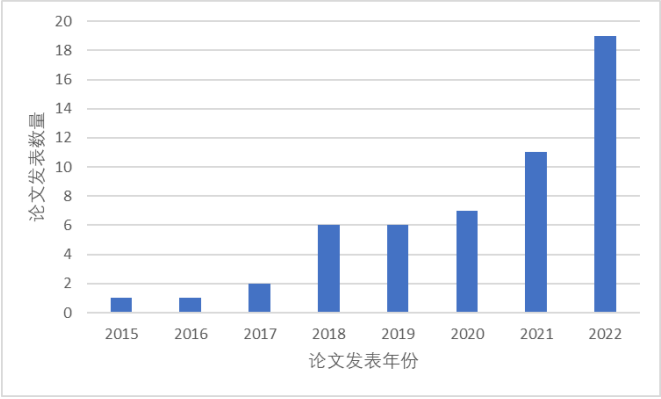

Fig. 1 Cumulative number of published papers for each year

图 1 相关论文每年累计发表的论文数

如图 1 所示, 从 2015 年到 2019 年, 智能化代码生成相关论文的数量呈上升趋势, 说明智能化代码生成任务逐渐被学界关注, 完成了部分研究并取得了一些成果. 随着 2020 年 CodeBERT[26]提出, 智能化代码领域也掀起了利用大型语言模型在代码语料进行预训练, 并在下游相关任务进行微调的热潮. 因此 2021 年和 2022 年相关的论文大幅增加. 尤其是 2022 年, 论文数量达到了 19 篇, 相关话题也在学界和工业界之外引起了社会的广泛讨论. 值得一提的是企业的论文(如 PANGUCoder 与 AlphaCode)常常直接在网上进行模型架构的公开, 为产品做技术支撑. 虽然这些论文不会在期刊或会议上发表, 但是其背后的思想及解决问题的工程方法具有很

强的借鉴意义, 不容忽视.

Hu 等人曾在 2019 年对基于深度学习的程序生成与补全技术进行了系统完备的中文综述[8], 但是随着后续智能化代码生成任务相关研究成果的急剧增加, 领域内亟需对于此任务进行回顾与总结, 方便后续研究者在此领域继续深耕. 本文将主要内容放在生成高级程序语言所编写的代码生成任务上, 主要讨论基于深度学习(即神经网络架构)的相关模型. 根据代码生成方法的主要思想和核心部件, 本文主要将代码生成方法分为三类, 基于代码特征的代码生成方法, 结合检索的代码生成方法和结合后处理的代码生成方法, 后两者可以视作基于第一类方法进行改进的代码生成模型, 而第一类又可以根据使用的范式不同分为基于监督学习的代码生成方法和基于预训练的代码生成方法. 表 1 展示了本文所收集论文分类的概况.

本文主要关注智能化代码生成任务使用的模型以及算法的创新性, 算法的评估指标(及优劣), 使用数据集以及适用编程语言(即适用场景)四方面内容. 同时对之前工作中涉及到的数据集及评价指标单独进行了整理, 并对于不同数据集涉及的编程语言类型, 规模和相关论文使用的情况进行了统计. 除此之外, 本文还针对拥有检索增强模块以及后处理环节相关的论文进行单独讨论.

本文第 1 章为引言, 第 2 章介绍综述的整体研究框架. 第 3 - 5 章分别介绍基于代码特征, 结合检索以及基于后处理的代码生成方法的相关研究工作, 并进行讨论. 第 6 章汇总常用的代码生成数据集. 第 7 章介绍代码生成评估指标. 第 8 章对全文进行总结并对未来值得研究的领域进行展望.

Table 1 Summary of existing studies of code generation based on deep learning

表 1 基于深度学习的代码生成现有工作的方法总结

方法		相关文献
基于代码特征的代码生成方法	基于监督学习的代码生成方法	Ling 等人[14], Yin 等人[15][19, 20], Rabinovich 等人[16], Iyer 等人[17, 18], Sun 等人[21, 24], Wei 等人[22], Ye 等人[23]
	基于预训练的代码生成方法	Lu 等人[2], Kanade 等人[25], Feng 等人[26], Guo 等人[27], Clement 等人[29], Ahmad 等人[31], Wang 等人[32], Phan 等人[33], Nigkamp 等人[34], Fried 等人[35], Chen 等人[36], Li 等人[38], Christopoulou 等人[40], CodeGeeX[42]
结合检索的代码生成方法		Drain[45], Hayati 等人[46], Hashimoto 等人[47], Guo 等人[48], Xu 等人[49], Parvez 等人[51], Zhou 等人[53], Zan 等人[54]
结合后处理的代码生成方法		Jain 等人[56], Wang 等人[57], Chen 等人[58], Le 等人[60] Zhang 等人[52], Poesia 等人[55]

3 基于代码特征的代码生成方法

直观上, 代码生成任务可以简单抽象为机器翻译的问题, 即将自然语言描述翻译为代码表示. 在机器翻译问题中最常用的模型是序列到序列模型(Squence2Sequence Model), 其主要思想就是从训练数据中学习自然语言特征, 并利用代码特征进行生成. 在这个过程中需要用到大量的自然语言-代码对(<Text, Code>), 以便模型能够从训练数据中学习到双模态数据的对应关系, 因此在本文中被称作基于监督学习的代码生成方法.

自 2017 年 Transformer 模型[9]的提出, 大型预训练模型在自然语言处理领域不断发展, 对计算机视觉甚至通用人工智能领域都产生了巨大的影响. 同样, 代码生成任务也积极引入大型预训练模型, 试图在已有挖掘代码特征的基础上结合预训练范式进一步提升模型性能. 预训练模型主要存在两个特征, 第一个是在预训练阶段其所需数据往往是无标注的数据, 通过还原掩藏掉的部分词元或片段来进行训练[10], 这样的训练过程被称为自监督训练. 第二个是模型的规模以及所需数据量都非常大, 这是因为许多工作[11,12]都发现随着模型规模的增大, 数据量的增多和计算量的提升, 模型的性能就会不断提高. 直到本文完成的时刻, 尚未有工作发现预训练模型性能的瓶颈. 与其他领域任务一样, 基于预训练的代码生成方法显著提高了代码生成的下限, 并逐渐成为最近几年研究代码生成问题的主要解决方案. 因此, 本文将基于预训练模型进行代码生成任务的代码生成方法单独列出, 作为本章的第二部分.

3.1 基于监督学习的代码生成方法

3.1.1 简述

在基于监督学习的代码生成方法中, 最常用的模型是序列到序列模型[13], 其对应的是编码器-解码器范式, 主要包含编码器(Encoder)和解码器(Decoder)两部分, 被广泛应用于自然语言处理中的生成类任务, 同样也被代码生成类任务广泛使用. 对于代码生成任务而言, 编码器将输入自然语言描述转变为固定长度的向量表示, 解码器则将编码器所产生的向量表示转变为可运行程序输出. 下面提到的相关模型均基于这样的模型框架进行, 由于比起自然语言, 代码自身具有规律化和模块化的特点, 因此模型往往对解码的相关操作以及应用信息进行改进. 使用到的数据均为<Text, Code>的形式, 目的是从数据对中挖掘出文本和代码之间的对应关系, 从而完成生成任务.

3.1.2 已有工作的分析

高级程序设计语言的智能化代码生成流程首先由 2016 年 Ling 等人[14]的工作定义. Ling 等人希望在卡牌游戏万智牌(Magic the Gathering)和炉石传说(Hearth Stone)中, 可以通过自然语言描述某一张卡牌的能力或效果, 让模型自动地去生成对应的卡牌定义代码(高级程序语言, 即 Java 和 Python)来减少开发人员编写卡牌效果的时间成本. 作者利用序列到序列模型来实现自然语言到代码片段之间的转换. 在传统注意力机制模型中, 输入的注意力是由 RNN(Recurrent Neural Network, 循环神经网络)中每一个单元的输出经过计算加权得来. 作者将注意力机制模型进行改进, 在计算输入的注意力时考虑了每一个输入域并将各个输入域都用作计算注意力值. 但是由于每一个输入域的值域和输入规模都不一样, 因此作者利用线性投影层将不同输入投影到同一个维度和值域上. 为了统一自然语言和代码之中的实体(如变量名等)信息, 作者利用选择预测器(select predictor)来预测自然语言中需要保留的字段, 并且结合序列到序列模型来生成代码的主体框架, 然后填入自然语言中的保留字段来实现对应卡牌代码的生成.

2017 年, Yin 等人[15]在 Ling 的工作上[14]做出了进一步改进, 作者利用语法模型显式地对目标语言的语法进行建模, 并且作为先验知识融入模型训练之中. 模型的编码器和之前工作一样利用双向长短期记忆网络(Long Short-Term Memory, LSTM)对自然语言进行编码, 但是在解码器端, 模型的目标生成是抽象语法树(Abstract Syntax Tree, AST)的构建动作序列而非可执行代码. 得到构建动作序列之后再生成树, 并转化为相应的可执行代码. 经过实验, 模型在多个数据集中达到了最好的结果.

Rabinovich 等人[16]则是引入抽象语法网络(Abstract Syntax Networks, ASNs)作为标准编码器-解码器范式的扩展. 与常规的序列到序列模型相比, 抽象语法网络的输出为抽象语法树, 核心在于改变解码器, 使其具有与输出树结构平行的动态确定的模块化结构. 作者在文中使用了抽象语言描述框架(Abstract Syntax Description Language, ASDL)将代码片段表示为具有类型化节点的树. 模块化解码器本质为相互递归的模块集合, 其包含的四个模块根据给定输入分别以自顶向下的方式递归生成可执行代码的抽象语法树. 经过这样流程所产生树的语法结构就反映了生成过程的调用图, 即完成了代码生成任务.

2018 年, 在上述模型的基础上, Iyer 等人[17]在编码器端构造了类型矩阵(Type Matrix)来标识方法名和变量名数据类型, 并将矩阵拼接到编码器输入之后, 来一起计算输入的嵌入向量; 在解码器端, 作者利用两步注意力计算每一个 token 的注意力以及每一种类型和变量名的注意力, 最终利用第二步的注意力结果作为最后生成代码的先验. 除此之外, 由于当下模型生成的类中有可能包含没有被学习过的域信息, 所以作者利用监督复制机制来决定哪一个标识符是需要复制到代码中生成. Iyer 等人[18]在一年后对此方法进行改进, 提出了一种迭代方法, 通过反复整理大型源代码体的最常出现的深度 -2 子树, 从大型源代码语料库中提取代码习语的语法树, 并训练语义分析器在解码过程中应用这些习语. 这里的代码习语是指代码片段中最常见的结构(比如嵌套循环, 条件判断等), 文中用语法解析树中经常出现的子树表示. 相比于利用语法树解析代码习语并得到一颗很深的语法树, 直接将代码习语应用到解码过程中可以提升训练速度和性能. 作者将所有的代码习语压缩到一颗深度为 2 的简化语法树之中, 使得模型可以更精准地输出对应的代码.

同样是在 2018 年, Yin 等人[19]则是在之前自己工作的基础上[14]提出了 TranX, 一个基于 transition 的代码

生成模型. TranX 的核心在于 transition 系统, 其能将输入的自然语言描述根据一系列树构造动作映射到抽象语法树上. 模型以抽象语法树作为中间表示, 对目标表示的特定领域结构进行抽象. 最后再利用一个由神经网络参数化的概率模型对每一个假定的抽象语法树进行评分, 从而得到目标代码的抽象语法树, 进而完成代码生成的目的. 一年后, 作者又在此基础上加以重排序(Reranking)技术[20]进行优化, 主要使用了重排序模型对代码生成结果进行重排序来提升模型性能. 重排序主要由两个方面内容完成, 一方面是根据生成的代码来重构真实输入文本的概率进行评估; 二是利用匹配模型直接评估代码与输入文本的相关程度. 重排序模型也同样分为两个部分来完成上面两方面的评估, 一是生成重构模块(Generative Reconstruction Feature), 使用一个带注意力机制的序列到序列模型, 通过生成的代码极大似然估计出输入的文本; 二是利用判别匹配模块(Discriminative Matching Feature), 将生成代码与输入文本的 token 借助注意力机制的神经网络计算出结果.

2019 年, Sun 等人[21]发现了程序代码比起自然语言描述包含有更多有意义的 token, 因此之前工作使用 RNN 来捕获长距离序列可能并不恰当. 作者在文中提出了基于语法的结构化卷积神经网络(Convolution Neural Network, CNN), 根据抽象语法树中的语法构造规则来完成代码生成任务. 详细来讲, 文章主要针对解码器部分进行了部分抽象语法树(即已生成的代码所构成的抽象语法树)信息的补充, 并为提取抽象语法树中不同粒度的信息分别设计了卷积层, 最终将多种信息聚合达到增强模型的效果.

同年, Wei 等人[22]提出代码生成任务和代码总结任务两者是对偶任务(dual task), 于是作者利用对偶任务学习的方法来同时提高两者的性能, 即代码生成(Code Generation)模型和代码摘要(Code Summarization)模型被同步地训练, 其思想也被运用到后续的研究当中[23].

2020 年, Sun 等人[24]使用 Transformer 架构来解决代码元素之间存在的长依赖问题, 并对模型进行修改提出 TreeGen, 使其能够结合代码的结构信息. 具体来说, TreeGen 共分为三个部分: NL Reader, AST Reader 和 Decoder. 其中, NL Reader 用于对自然语言描述进行编码, AST Reader 用于对已生成的部分代码的语法树进行编码, Decoder 则是用于聚合带有自然语言描述的生成代码的信息并预测接下来的语法规则. 本文将基于 AST 的代码生成任务视作通过语法规则来扩展非终结符节点, 直至所有的叶子结点均为终结符停止的过程.

3.2 预训练模型

3.2.1 简述

近些年来, 预训练模型在自然语言处理领域取得了巨大的成功. 从预训练模型架构来看, 可以分为编码器-解码器架构(Encoder-Decoder), 仅编码器架构(Encoder-only)和仅解码器架构(Decoder-only). 虽然模型总体架构并无形式创新, 但是其核心思想与之前的基于监督学习的模型不同. 预训练模型从大规模无标注的数据中通过自监督的训练策略获取知识, 且绝大多基于 Transformer 架构进行. 首先在大规模的无标注的数据集上对模型进行预训练, 然后利用预训练得到的表征在下游的有标注数据集上进行微调. 实验证明这种方式可以极大地提高模型的泛化性, 在多项任务上取得了很好的结果. 类似地, 研究人员在代码领域也提出了对应的预训练模型, 并且在代码生成任务上取得了优异的效果.

3.2.2 已有工作的分析

CuBERT[25]是首个提出代码预训练模型的工作, CuBERT 继承 BERT[10]的架构, 利用代码语料进行训练, 但和 CodeBERT[26]相比, 其影响力较小. CodeBERT 是第一个多编程语言的大型双模态(指自然语言描述 NL 与编程语言 PL)预训练模型, 后续在各个下游任务上被广泛使用, 影响力较大. CodeBERT 的预训练的任务为掩蔽语言建模(Mask Language Modeling, MLM)[10], 通过随机掩盖模型中的某些词并让模型去预测被掩盖的词, 在预训练阶段提高模型的理解能力. 在实际场景当中, CodeBERT 被用来作为编码器对输入的文本或代码进行编码, 然后应用到各式各样的下游任务中, 例如代码生成, 代码摘要以及代码检索等.

由于 CodeBERT 仅仅包含文本的语义信息, 所以 Guo 等人[27]在 2021 年提出将代码的结构信息数据流纳入预训练的过程之中并提出 GraphCodeBERT, 同样采用 MLM 对模型进行预训练. 但不仅仅掩盖文本信息中的一些单词, 而是也会在代码的数据流图之中随机掩盖某些数据节点然后让模型去预测. 实验证明在预训练过程中显式地去考虑代码的结构信息可以极大地提高代码对模型的理解能力, 并提高在下游任务当中的性能.

上述预训练模型仅仅包含编码器端, 这种架构的预训练模型在理解任务上的效果较好, 但无法很好地完成生成式任务. 为了更好地完成根据自然语言描述进行代码生成(Text-to-Code)的任务, CodeXGLUE[2]中提出了 CodeGPT 模型, 这是一个由代码语料进行训练, 与 GPT-2[28]完全同架构的 12 层 Transformer 解码器模型. 与仅编码器架构的模型相比, 仅解码器架构能够更好地完成生成代码的任务.

除了单独使用 Transformer 的编码器或解码器结构, 后续也有相关工作同时使用了 Transformer 的两端.

Clement 等人[29]对于代码和自然语言描述采用了同一个词汇表, 基于 T5(text-to-text transfer transformer)[30]提出了多模态的翻译模型 PYMT5, 通过单个模型既可以同时学习到代码/自然语言生成并且理解二者之间的关系. PYMT5 使用 T5 利用相似子序列掩藏目标(similar span-masking objective)进行预训练, 其本质是一个基于编码器-解码器架构的 Transformer 模型. 子序列掩藏目标是指随机采样一些连续的 3 个 token 的子序列使用特殊标记(例如[MASK 0])对其进行掩藏, 然后训练序列到序列模型来补全这些掩藏的 token, 训练目标包含了被隐藏 token 及其序号. 作者将 PYMT5 与在同样数据集上进行预训练的 GPT-2 相比取得了很大提升.

Ahmad 等人[31]提出的 PLBART 也是一个编码器-解码器模型, 在预训练过程当中, 与 CodeBERT 做法一样, 作者随机掩盖某些单词, 但是 PLBART 输出的是一个完整的文本或单词, 其中包括了被掩盖的单词. 通过这种训练方式, 作为一个序列到序列模型的 PLBART 就可以在预训练阶段在编码器和解码器端同时学习到很好的初始化点, 让预训练好的模型可以更快更好地应用到下游任务当中. PLBART 提高了模型在生成任务上的能力, 也提高了模型在代码生成任务上的表现.

更进一步地, Wang 等人[32]在 2021 年提出了 CodeT5, CodeT5 在预训练阶段充分考虑到了代码的特点, 作者从代码片段中抽取标识符, 并重点让模型去预测这些在代码中具有实际意义的单词, 实验证明 CodeT5 在包括生成任务在内的多项软件工程领域任务均取得了更好的效果. 同一时间, Phan 等人提出了 CoTexT[33], 同样使用了编码器-解码器架构, 模型初始化也使用了 T5. 唯一的区别在于为了缩小预训练和调优之间的差异, CodeT5 利用了<Text, Code>双模态数据去训练模型完成一个双向生成的转换. 具体来说, CodeT5 将 NL→PL 的生成与 PL→NL 的生成视作对偶任务(dual task), 并同时对模型进行优化. 根据两篇文章中实验结果的对比, CodeT5 的效果要好于 CoTexT, 但是如果只是在 T5 的基础上利用多任务学习来提升代码生成任务的性能, CodeT5 和 CoTexT 效果相当. 简单来说, CodeT5 和 CoTexT 两个模型在完成代码生成任务的具体实现上最大的差别在于是否使用代码生成与代码摘要作为对偶任务进行训练. 这样的实验结果侧面证明了将代码生成和代码摘要任务作为对偶任务能够提升模型的生成能力.

Nijkamp 等人[34]开源了大型预训练模型 CodeGEN, 模型参数高达 16.1B, 依次在 THEPILE, BIGQUERY, BIGPYTHON 三个数据集上进行训练, 数据量超过 800G. 与之前直接将自然语言输入给预训练模型不同, 作者提出利用大型预训练模型进行对话式程序生成的方法: 作者将编写规范和程序的过程描述为用户和系统之间的多轮对话. 用户分多次为系统提供自然语言, 同时以合成子程序的形式接收来自系统的响应, 这样用户与系统一起在多轮对话后完成代码生成. 分步提供自然语言规范的方式可以将较长且复杂的意图分解为多个简单的意图, 减少每一轮对话中模型的搜索空间. 除此之外, 作者还开发了一个多轮编程基准来衡量模型的多轮程序综合能力, 实验结果表明以多轮方式提供给 CodeGEN 的相同意图与单轮提供的相比显着改进了代码生成的性能, 验证了对话式代码生成范式的有效性.

之前的代码生成模型都是从左到右生成代码序列, 然而在实际开发过程中, 代码很少直接以从左到右的方式编写, 而是完成部分代码编写后反复编辑和完善. InCoder[35]打破了先前从左至右的代码生成预训练模型范式. 这是一种统一的生成模型, 可以执行程序合成(通过从左到右生成)以及编辑(通过掩蔽和填充). InCoder 的模型架构继承自 GPT 架构, 不同点在于其对训练语料进行顺序打乱预测. 该方法随机选择一个跨度并将其替换为掩码 token, 并将跨度放置在序列之后作为目标. 利用这样处理后的语料进行训练, 使得模型具有填充双向上下文的能力. 这样 InCoder 不仅可以从左到右预测 tokens, 而且可以根据两端的 tokens 预测中间的 tokens, 实现了填充式的代码生成技术. 这是第一个能够填充任意代码区域的大型生成代码模型, 这种以双向上下文为条件的能力大大提高了代码生成任务的性能.

Table 2 Statistics of code generation pretraining model

表 2 代码生成预训练模型概况

预训练模型	backbone	模型规模	预训练数据集	数据量	训练语言
CuBERT	BERT	-	Python from Github	6.6M	Python
CodeBERT	BERT	125M	CodeSearchNet	3.5G	Ruby/JavaScript/Java/ Python/GO/PHP/English
GraphCodeBERT	BERT	125M	CodeSearchNet	3.5G	Ruby/JavaScript/Java/ Python/GO/PHP/English
CodeGPT	GPT-2	124M	Python and Java from CodeSearchNet	Python 1.1M Java 1.6M	Java/Python
CoTexT	T5	-	CodeSearchNet AND Java and Python from BigQuery	-	Java/Python
CodeT5	T5	60M/223M/ 770M	CodeSearchNet and C/C# datasets	8.35G	Ruby/JavaScript/GO/ Python/Java/PHP/C/C#
PLBART	BART	140M	Java and Python from BigQuery AND SO posts	655G	Java/Python/English
CodeGen	-	350M/2.7B/ 6. 1B/16.1B	THEPILE/BIGQUERY/ BIGPYTHON	THEPILE 825G	C/C++/Go/Java/ JavaScript/Python/English
InCoder	-	1. 3B/6.7B	content from StackOverflow	159G	PTYHON and 28 other languages
CodeX	GPT-3	300M/2.5B/ 12B	Python from GitHub	159G	Python
AlphaCode	-	300M/1B/ 3B/9B/41B	a snapshot of github	715.1G	C++/C#/Java/JavaScript/ Lua/PHP/Python/Ruby/ Go/Rust/Scala/TypeScript
PanGu-Coder	PanGu-alpha	317M/2.6B	Python from GitHub	147G	Python
CodeGeeX	-	13B	open-sourced code datasets, The Pile and CodeParrot	-	C++/Python/C/Java/ JavaScript/Go/HTML/PHP/ Shell/CSS/Others
aiXcoder L	GPT-2	1.3B	Java from GitHub	-	Java
aiXcoder XL	-	13B	Open-sourced code from GitHub	-	Java

越来越多基于 Transformer 体系结构的大型预训练模型被提出并在代码生成任务上取得最优结果. 因此, 一些企业机构着手于将大型预训练代码生成模型落地, 试图为广大开发人员提供便利, 并在此过程中为业界提供了大量优质的代码生成模型.

2021 年年末, OpenAI 最早发布的 CodeX[36], 是基于 GPT-3 在公开数据集上预训练得到的大规模模型, 基于 CodeX 的 Copolit[37]插件也已成为代码生成辅助工具的标杆, 在 CodeX 论文中提出的 HumanEval 数据集也成为后续代码生成的常用基准数据集之一.

2022 年年初, DeepMind 公司研发的展现出强大编程能力的 AlphaCode[38]在新闻上号称打败了一半的程序员, 本质上也是基于公开代码仓库进行预训练的大规模模型. 与 CodeX 不同, AlphaCode 更专注于竞赛题目的编写, 因此选用了完整的 Transformer 架构的模型, 便于更好地理解较长的由自然语言描述的题目, 同时在调优时也选择了 CodeForces[39]的竞赛题目.

国内, 华为推出的 PanGu-Coder[40]是基于 PanGu-alpha 模型在公开代码数据集上进行预训练. 之后基于此

开发的 CodeArts 插件也已在实际开发场景中拥有不错的表现, 对标基于 CodeX 的 Copoilt.

2022 年, aiXcoder[41]团队陆续推出了用于 Java 代码补全的 13 亿参数量的 aiXcoder L 和 130 亿参数量的 aiXcoder XL 服务. aiXcoder L 基于 GPT-2, 在开源 Java 代码上训练得到. aiXcoder XL 基于自研的 masked language model 框架, 能做到单行、多行以及函数级代码补全和生成.

最近, 清华大学联合鹏城实验室共同推出的大规模代码生成预训练模型 CodeGeeX[42], 采用了标准的 Transformer 架构, 在公开代码仓库超过 20 多种编程语言上进行预训练, 能够支持较其他开源基线更高精度的代码生成性能, 能够支持代码片段在不同编程语言间进行自动翻译转换.

如表 2 所示, 汇总了本节涉及到的代码生成预训练模型相关内容.

除以上专门用于进行代码生成任务的相关模型, ChatGPT[43]虽作为问答模型, 但也被证实具有代码编写的能力, 自 2022 年 12 月发布以来就引起了学术界和工业界的广泛讨论. ChatGPT 能够适应不同的问题情境给出接近甚至超过人类的回答, 并具有一定推理和代码编写能力. ChatGPT 与 InstructGPT[44]使用了相似的训练方式. 主要流程分为三个步骤, 第一步是搜集带标记的数据使用监督学习策略对已有的 GPT 模型进行调优; 第二步是搜集来自人类反馈的比较数据来训练一个分类器(称为 reward model)用于比较 GPT 模型生成若干答案的好坏; 第三步是借助第二步训练好的 reward model, 利用强化学习策略对模型进行进一步优化. 原有的提示学习范式是通过调整模型的输入使得下游任务更好地适配模型, 而通过 InstructGPT 的训练过程及 ChatGPT 的出色表现可以发现, 模型的性能尚未被充分挖掘, 且可以通过人为标注反馈的方式, 让模型更好地去理解用户的意图, 达到让模型适配用户的目的.

3.3 小结

本章主要对基于代码特征的代码生成方法相关工作进行了叙述与梳理, 并按照监督学习与预训练-调优范式分为两节进行概述.

在基于监督学习的代码生成方法中, 主要依托于编码器-解码器架构进行自然语言描述与代码特征的挖掘, 试图从训练数据中学习到自然语言与代码之间的对应关系. 除此之外, 一些工作针对解码器生成代码的部分进行了改动, 试图将更多代码相关的规则性信息融入其中, 取得了比仅使用 token 进行代码生成更好地性能.

在基于预训练的代码生成方法中, 主要对引入代码生成任务的预训练模型进行了简单介绍. 这一小节的方法的骨干模型(backbone)均来自于自然语言处理领域的大型语言预训练模型. 将其应用于代码生成任务的一个大致流程为: 收集大量无标注的代码语料, 其中包含代码注释, 利用试图还原掩藏掉的部分代码片段对模型进行预训练, 最后再使用<Text, Code>对数据对预训练模型进行调优.

最近, ChatGPT 所具备强大的语言理解能力和一定程度的推理能力引起学术界和工业界的广泛关注, 其背后通过反馈调整使模型来匹配用户的思想, 为进一步挖掘大规模语言预训练模型提供了新的解决思路.

4 结合检索的代码生成模型

4.1 简述

对于第三章中的代码生成模型而言, 其代码生成过程的解空间过大, 这种现象在预训练模型中的表现愈发明显[36]. 互联网上存在的代码片段数量非常庞大, 对于用户的绝大部分要求而言, 其他开发人员可能存在过类似的需求, 并已经与他人协作或自行完成相关代码的设计与编写. 因此, 检索出类似已存在的代码模板这个任务本身就是有利于用户更深刻理解任务的. 同时, 基于前人相关任务的代码模板进行的改动也比直接生成的代码更具有实用价值.

编写代码是一个开放域(open-domain)的问题, 即在编写代码过程中不可避免需要参考前人工作与他人的编程思路, 而在第二章中介绍的大部分工作则是直接将其视作封闭性质的任务(closed-book)[45], 即只是根据训练数据的模式来完成代码生成的任务. 从这个角度来看, 引入外部知识库对原有训练的模型进行补充是有意

义有价值的. 事实上, 利用自然语言描述的编码作为编码过程中的先验可能是不足的. 为了让已有代码生成模型与外部代码知识数据库相结合, 有相关工作利用检索操作对代码生成过程进行增强. 通过检索相似代码帮助解码器进行代码生成, 以减小解码空间, 最终提升生成代码的质量.

4.2 已有工作的分析

Hayati 等人[46]首次将检索技术引入代码生成任务, 提出 RECODE 模型, 针对 Yin 等人[15]提出的模型不能正确生成复杂代码的问题进行改进. 文中使用 TranX[19]中 transition 的构造规则来创建抽象语法树. 具体来说, RECODE 首先从训练集中检索与输入句子最相似的自然语言描述; 然后根据对应的代码片段抽取语法树的生成行为序列; 通过启发式方法, 利用前面得到的生成行为序列来改变最终解码过程中语法树构建特定行为的概率, 从而提升模型性能.

Hashimoto 等人[47]同样在代码智能化任务中使用到了检索技术, 但是与增强或改进已有生成算法不同, 作者认为编辑并更改已有的代码比直接生成代码更简单有效, 并基于此想法设计了一种检索并编辑的框架(retrieve-and-edit framework). 该框架下的模型需要首先根据自然语言描述的输入与训练集中的自然语言进行相似度计算, 并检索到相关的代码, 然后再编辑检索到的代码使其能够更加符合自然语言描述的任务. 详细来说, 此框架包含检索器(Retriever)和编辑器(Editor)两部分内容. 对于检索器而言, 本质是一个序列到序列模型. 首先利用训练数据对模型进行训练, 目的是将自然语言输入映射到能够重构为代码输出的向量; 训练完成后利用检索器中的编码器对输入的自然语言进行编码得到向量表示, 与训练集中其他自然语言描述计算相似度得到检索结果. 对于编辑器而言, 作者使用了一个标准的带有注意力和复制机制的序列到序列模型完成对代码的编辑.

Guo 等人将代码检索与元学习技术相结合, 用来提高模型代码生成的能力[48]. 作者提出了一种上下文感知的编码器-解码器模型来作为检索器, 将上下文环境(文中是指 Java 中的类环境信息)以及自然语言描述输入编码作为隐变量用于检索; 此外将检索出的数据点利用模型无关元学习(model agnostic meta-learning, MAML)训练范式快速地将检索数据适应到对应的任务之中. 作者沿用了先前工作[47]中的检索器, 本质上是利用编码器计算出上下文以及自然语言的隐变量进行检索. 在实际训练过程中会在训练数据中采样一部分作为测试集, 使用剩下数据经过检索器得到训练集, 以此来定义一个任务. 通过不同任务对模型进行的参数更新及选择, 这就是元学习的训练过程. 经过这种训练方式, 模型可以快速适应到检索器所提供的数据中. 作者在两种范式上均进行了实验, 分别是 MAML 和检索并编辑(即上文提到的 retrieve-and-edit framework[47]), 结果表明 MAML 可以更精准地生成代码.

Xu 等人[49]针对自然语言注释与代码数据的缺乏的问题, 基于当时表现最优的 TranX 模型[19], 引入了两个外部知识库来进行数据增强, 帮助模型进行预训练从而提升模型的性能. 两个外部知识库分别是从 Stack Overflow 上和 API Documentation 上提取出来的 NL-code 数据对. 考虑到 API Documentation 上处理得到的数据虽然没有噪音但是可能与开发人员的实际开发情境不符, 作者根据 CoNaLa[50]以及直接爬取 Stack Overflow 的数据(同样采用了 Yin[50]的方法)构建了真实用户产生的 NL-PL 分布, 并对 API Documentation 得来的数据进行重采样以消除文档与实际用户使用情况之间的差异. 这种方法与模型无关, 文中作者利用引入外部知识库的数据对模型进行预训练, 并在人工注释的数据集 CoNaLa 上进行调优, 使得模型能够很好地完成代码生成任务.

Parvez 等人[51]认为之前的生成模型只是借助高质量的代码和文档信息对模型进行训练, 而在生成过程中并没有直接利用这些额外信息. 因此作者提出 REDCODER(Retrieval augmentED CODe gEneration and summaRization)框架, 利用检索对生成模型的输入进行增强, 以此来达到最大效用利用额外信息的效果. 同时, 检索增强的方法是双面的, 一方面可以用来检索代码来完成代码生成任务的增强, 另一方面也可以通过检索到的文本信息对代码摘要任务进行增强. 详细来讲, REDCODER 主要分为两部分, 第一部分是检索器(retriever), 其中包含两个编码器, 分别对代码和自然语言描述进行编码, 根据编码后的向量相似度检索出相关的源码(用于代码生成)或摘要(用于代码摘要); 第二部分为生成器(generator), 将检索到的源码/文本内容与

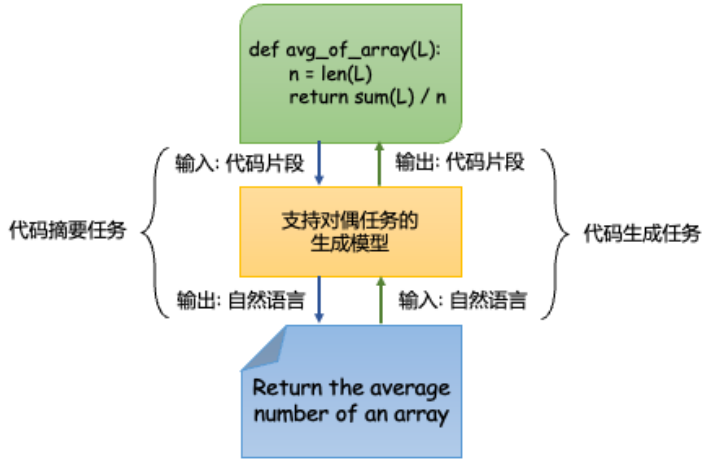

Fig. 2 Input and output of the dual task including code generation and code summarization

图 2 代码生成与代码摘要任务作为对偶任务的输入与输出

原始的模型的输入相结合来产生目标输出, 通过检索来增强了输入从而提升了 REDCODER 的代码生成能力. 对于代码生成任务而言, 通过交换输入输出可以得到代码摘要任务, 反之亦然, 这样的任务被称为对偶任务(dual task), 其两个任务的输入输出及模型示意入图 2 所示. 将代码生成任务与代码摘要任务视作对偶任务在其他工作中也有涉及, 用以提升代码生成任务的性能[22, 23, 51, 52].

Drain 等人[45]对于提供的 docstring(即描述需求的自然语言)与数据库中的文档分别创建编码器, 并针对 CodeSeachNet 问题中提出的 NBoW 算法进行了改进. NBoW 算法在使用 BPE 分词后直接对序列进行编码. 序列表示取自词元平均(mean-pooling), 最大(max-pooling)或基于注意力的加权和(attention-like weighted sum)三者之一. 作者在本文中则是直接将序列表示的三个来源直接进行线性组合, 称为混合表示, 最后再通过相似度计算得到前 K 个样板方法的代码. 该工作通过实验比较证明了基于这样混合表示的检索算法要优于之前的 ElasticSearch, NBoW 以及 DPR(Dense Passage Retrieval)三种常用的检索算法. 为了得到更高的信息密度, 检索到更多有意义的 NL-PL, 作者没有直接使用 Stack Overflow 上问答的原始数据, 而是使用 CoNaLa[50]作为 Stack Overflow 的替代数据集进行检索. 作者在文中使用的生成模型是 BART-large, 并在掩藏过的 Python 代码和 Stack Overflow 问答对的混合数据上继续训练. 在实际生成过程中, 生成模型将文本描述, 函数声明以及检索到的前 K 个样板方法的代码作为输入, 目的是生成方法的具体实现.

上述基于检索的工作都是在训练集中检索相似数据, 这些方法假定所有库和函数调用都在训练数据中存在, 因此在本质上无法借助训练数据中没有的代码对模型进行增强, 同时, 生成的代码往往可能会调用看不见的函数和库, 如果只是从训练数据中进行检索, 模型是不具备泛化能力的. 实际上, 训练数据不可能包含所有的库. 并且, 公开可用的 API 库不断增长变化, 代码模型不可能通过简单地在现有代码库上训练来与所有可用的 API 保持同步. DocPrompting[53]同样使用检索的方式进行代码生成, 不同的是, 其维护一个文档库, 该文档库是独立于模型的外部数据存储, 并且允许进行频繁的更新. 无需重新训练模型, 模型可检索并利用动态更新的文档库, 从而生成模型训练时从未见过的代码. 实验表明, 这种检索的方式可以有效提升代码生成的性能. 同时, 作者在文中提到的对外部文档库进行更新在一定程度上能够缓代码生成作为开放域任务的普遍问题.

在实际开发过程中, 公司或团队内部的软件开发者共享工程代码与文件是普遍的. 考虑到安全性, 这些代码往往不会进行公开处理, 而是作为私有仓库在公司内部进行共享流转与使用. 由于私有仓库中的代码往往不会作为训练数据用于模型的训练, 因此希望针对私有仓库生成代码对于已有模型是极具挑战的. 为解决

这个问题, Zan 等人[54]设计了一个框架使得预训练模型拥有生成使用私有仓库代码的能力. 受到开发人员学习使用私有库进行代码编写过程的启发, 作者试图通过构建检索并生成的框架来模拟学习私有 API 文档以及调用所学 API 来完成所需功能的过程. 作者在框架中创建了两个模块来完成这两项功能. 第一个模块是 APIRetriever, 其作用是基于编程问题和 API 文档来检索到有用的 API, 并设计了方便的交互方式来了解用户的需求, 本质是将自然语言描述与 API 信息利用 BERT 进行编码并进行相似度计算. 第二个模块是 APICoder, 其作用是直接使用已有的语言模型去调用私有的 API 完成描述的任务. 为了能够让语言模型知道怎样更好地调用 API, 作者针对 APIRetriever 检索到的代码文件进行分块, 并为每个代码块设置相应的提示(prompt), 基于 CodeGen 继续在包含了公开库 API 信息的代码库上进行训练, 称为 CodeGenAPI.

4.3 小结

本章主要对结合检索的代码生成方法相关工作进行了简单介绍.本章中提到的方法并没有设计新的模型架构, 而是通过结合检索的方式生成代码片段. 如图 3 所示,结合检索的代码生成方法大致可以分为两种形式, 第一种是图 3 的上半部分(即图中①部分): 从训练数据或是外部知识库检索出与目标代码相关或相似的代码片段, 将其作为模板进行编辑并返回结果; 第二种是图 3 的下半部分(即图中②部分): 将检索得来的借过作为原有模型的输入, 试图通过为自然语言描述补充代码信息的方式提升模型的性能. 结合检索的代码生成方法能够大大降低生成模型解空间的规模, 同时也能在一定程度上利用外部知识库使模型跳脱出自己训练的固有规则, 对于代码生成任务是有帮助的.

在检索的过程中, 往往将自然语言描述作为查询的键(key), 并经模型处理后借助向量进行表示, 在训练数据或外部数据库查找与查询键(key)相关的代码片段进行返回. 在这个过程中, 模型向量表示的能力决定了检索结果的质量. 因此, 尝试多种自然语言表示形式与不同的模型表征方法可能是检索增强的一条未来之路.

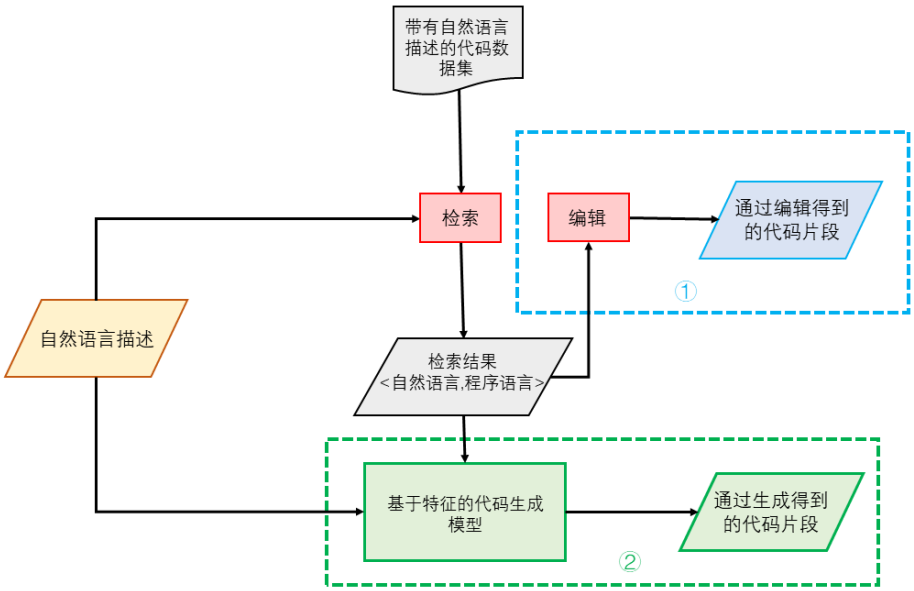

Fig. 3 The overview of the methods with retrieval

图 3 结合检索的代码生成方法概览图