计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(9):77−88 [doi: 10.15888/j.cnki.csa.009217]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于代码语句掩码注意力机制的源代码迁移模型①

徐明瑞, 李　征, 刘　勇, 吴永豪

(北京化工大学 信息科学与技术学院, 北京 100029)
通信作者: 刘　勇, E-mail: lyong@mail.buct.edu.cn

摘　要: 源代码迁移技术旨在将源代码从一种编程语言转换至另一种编程语言, 以减轻开发人员迁移软件项目的负担. 现有研究通常利用神经机器翻译 (NMT) 模型将源代码转换为目标代码, 但这些研究忽略了代码结构特征, 导致源代码迁移性能不佳. 为此, 本文提出了基于代码语句掩码注意力机制的源代码迁移模型 CSMAT (code-statement masked attention Transformer). 该模型利用 Transformer 的掩码注意力机制 (masked attention mechanism), 在编码时引导模型理解源代码语句的语法和语义以及语句间上下文特征, 在译码时引导模型关注并对齐源代码语句, 从而提升源代码迁移性能. 本文使用真实项目数据集 CodeTrans 进行实证研究, 并使用 4 个指标评估模型性能. 实验结果验证了 CSMAT 的有效性, 同时验证了代码语句掩码注意力机制在预训练模型的适用性.

关键词: 代码语句; 掩码; 代码迁移; 机器翻译; 注意力机制

引用格式: 徐明瑞,李征,刘勇,吴永豪.基于代码语句掩码注意力机制的源代码迁移模型.计算机系统应用,2023,32(9):77–88. http://www.c-s-a.org.cn/1003-3254/9217.html

Source Code Migration Model Based on Code-statement Masked Attention Mechanism

XU Ming-Rui, LI Zheng, LIU Yong, WU Yong-Hao

(College of Information Science and Technology, Beijing University of Chemical Technology, Beijing 100029, China)

Abstract: Source code migration techniques are designed to convert source code from one programming language to another, which helps reduce developers' burden in migrating software projects. Existing studies mainly use neural machine translation (NMT) models to convert source code to target code. However, these studies ignore the code structure features, resulting in poor source code migration performance. Therefore, this study proposes a source code migration model based on a code-statement masked attention Transformer (CSMAT). The model uses Transformer's masked attention mechanism to guide the model to understand the syntax and semantics of source code statements and inter-statement contextual features when encoding and make the model focus on and align the source code statements when decoding, so as to improve migration performance of source code. Empirical studies are conducted on the real project dataset, namely CodeTrans, and model performance is evaluated by using four metrics. The experimental results have validated the effectiveness of CSMAT and the applicability of the code-statement masked attention mechanism to pre-trained models.

Key words: code statement; mask; code migration; machine translation; attention mechanism

1　引言

随着计算机软件和硬件的发展, 编程语言也快速更新迭代. 为了让传统平台的软件项目适应新的业务场景, 开发人员需要用新的编程语言重写软件项目, 这

① 基金项目: 国家自然科学基金 (61902015, 61872026)
收稿时间: 2023-02-14; 修改时间: 2023-03-14; 采用时间: 2023-03-30; csa 在线出版时间: 2023-06-09
CNKI 网络首发时间: 2023-06-12

一过程称为代码迁移. 然而, 软件项目代码量十分庞大, 不同编程语言有着不同的功能特性, 这使得开发人员迁移源代码既耗时又容易出错. 例如澳大利亚联邦银行花了大约 7.5 亿美元和 5 年时间将其代码库从 COBOL 迁移至 Java[1]. 为此, 研究人员提出源代码迁移技术, 期望将源代码从一种编程语言转换至另一种编程语言, 以减轻开发人员迁移源代码的负担.

随着神经网络技术的发展, 神经机器翻译 (neural machine translation, NMT) 模型在自然语言翻译任务中取得了优异的成绩[2,3], 促使着研究人员探索使用神经机器翻译模型迁移源代码的可能性. 例如, Chen 等人[4]提出了基于语法树的 tree-to-tree 模型, 该模型的树编码器理解源代码抽象语法树 (abstract syntax tree, AST), 带有注意力机制的树译码器模型对齐并转换源代码 AST 节点为目标代码 AST 节点, 进而生成目标代码 AST. Shiv 等人[5]采用 Transformer 模型改进了 tree-to-tree 模型架构, 并提出了适用于 Transformer 模型的树节点位置编码方式. 然而此类模型受限于语法树: 一方面抽象语法树会省略部分语义节点, 例如有些编程语言的抽象语法树会把“+”“>”“<=”等符号抽象化为“BinOpSub”节点, 不能表现出源代码的语义特征[6]. 另一方面具体语法树 (concrete syntax tree, CST) 尽管能完整地表征源代码语法和语义, 但是其节点多且层数深, 模型难以有效地从中提取代码特征. Feng 等人[7]使用六种编程语言和自然语言数据集, 并采用掩码语言建模任务和替换词符检测任务预训练了 Transformer 编码器模型 CodeBERT, 让模型通过代码文本学习语法和语义. Lu 等人[8]提出了代码翻译任务, 并将代码文本作为输入, 让 Transformer 和 CodeBERT 模型学习代码文本的转换规则. 然而有研究表明, 此类模型忽略了代码结构特征, 导致模型难以理解并迁移源代码[9−11]. 为了解决此问题, Guo 等人[12]把数据流作为模型输入, 并采用数据流的边预测任务和数据流节点-代码变量对齐等任务预训练了 Transformer 编码器模型 GraphCodeBERT, 旨在用数据流引导模型更好地理解源代码. 从这些源代码迁移工作可以得知, 目前研究通常的做法是从语法树、数据流或者模型的角度提升模型迁移源代码的性能, 很少考虑到为源代码迁移任务设计理解代码结构特征的模型, 导致这些模型在源代码迁移任务中的性能不佳.

为了解决此问题, 本文提出模型可以在编码时理解源代码语句的语法和语义以及语句级别上下文特征, 并在译码时关注并对齐源代码语句, 进而提升迁移源代码的性能. 图 1 给出了代码迁移数据集中的一组 C#和 Java 代码语句关联性样例, 可以发现 C#的构造函数声明语句对应 Java 的构造函数声明语句和父类构造函数调用语句, C#的赋值语句对应 Java 的赋值语句. 该样例表明源代码和目标代码的语句有着语法和语义的联系. 一些研究也证明了神经网络模型理解编程语言的语法和语义上的可行性. 例如, Hindle 等人[13]验证了代码是一种比自然语言更具备规则的语言, 因此代码语句可以使用语言模型来建模; Zhang 等人[14]通过实验验证了代码语句的语法是有规则的, 并且代码语句序列同样具有规则及顺序依赖关系. 此外, 神经机器翻译研究中常用的 Transformer 模型[3]有着掩码注意力机制 (masked attention mechanism), 该机制能依据掩码矩阵来引导模型关注指定的词符特征[12]. 因此, 在模型迁移源代码至目标代码时, 可让掩码注意力机制引导模型对齐并转换源代码语句的语法和语义特征, 提升源代码迁移的性能.

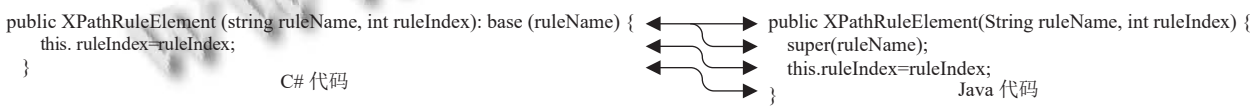


图 1 一组 C#和 Java 代码语句关联性样例

基于上述讨论和研究的启发, 本文在 Transformer 模型基础上提出了一种基于代码语句掩码注意力机制的源代码迁移模型 CSMAT (code-statement masked attention Transformer). 该模型的执行流程分为 3 个步骤: 首先预处理代码文本, 用自定义的语句词符<loc>标识代码语句; 随后, 编码器在掩码矩阵的引导下, 其自注意力 (self-attention) 机制能够关注到源代码语句的语法和语义以及语句间上下文特征; 最后, 译码器在掩码矩阵的引导下, 其跨层注意力 (cross-attention) 机制能够关注并对齐源代码语句, 进而生成目标代码.

为了验证模型性能, 本文基于真实项目数据集 CodeTrans 设计了源代码迁移任务, 并采用 BLEU、完全匹

配率、CodeBLEU 和词法正确率 4 个指标对模型输出结果进行评估. 对比实验结果显示 CSMAT 在 4 个指标的分值均优于对比模型, 并且消融实验验证了代码语句掩码注意力机制的有效性. 此外, 预训练模型的改进实验验证了代码语句掩码注意力机制的适用性. 本文的模型和实验结果公布在: https://github.com/Xmr-nxbx/CSMAT.

2 相关工作

传统的源代码迁移研究主要是采用基于规则的方法. Yasumatsu 等人[15] 从代码的基本类型、变量、字面量和表达式等角度提出了 Smalltalk 到 C 语言的翻译器. Bravenboer 等人[16] 设计了程序转换规则语言 Stratego 和程序转换工具集 XT. 石学林等人[17] 提出了数据流转换规则和控制流转换规则, 并通过 C2J 平台将 COBOL 转换至 Java. 刘静[18] 提出了词法语法分析、语法树存储结构和翻译处理等方式实现了 Verilog 至 MSVL 的转换. 然而, 以上方法需要开发人员提供准确且完备的代码迁移规则, 使得模型的构建低效且容易出错.

为了解决上述问题, 研究人员采用基于短语的统计机器翻译 (phrase-based statistical machine translation, PBSMT) 技术[19] 自动构建模型. 例如, Nguyen 等人[20] 用分词器将代码转换成词符序列, 然后基于 PBSMT 技术训练了 Java 至 C#的迁移模型. 但这些模型仅输入代码文本, 没有考虑代码结构特征, 导致模型迁移性能不佳. Karaivanov 等人[21] 利用短语表、编程语言前缀语法规则和 API 映射规则等特征结合 PBSMT 模型, 设计了 C#至 Java 的迁移模型. Nguyen 等人[22] 定义了粗粒度的语法单元模板 (syntaxemes) 和细粒度的语义单元模板 (sememes), 提出了分而治之的 Java 至 C#迁移模型. 该模型先迁移源代码为语法单元模板, 再依据该模板内的语义单元模板生成具体的目标代码. 从改进的研究可以发现, 基于 PBSMT 技术的源代码迁移模型需要通过人工定义规则或者构建模板的方式提升模型的迁移性能.

后来, 神经网络机器翻译模型取得了成功[2,3], 许多研究人员开始探索该技术用于源代码迁移任务的可能性. 例如, Chen 等人[4] 提出了 tree-to-tree 的抽象语法树翻译模型, 该模型能将源代码抽象语法树转换至目标代码抽象语法树. Feng 等人[7] 使用 6 种编程语言和自然语言, 并采用掩码语言建模任务和替换词符检测任务预训练了 Transformer 编码器模型 CodeBERT. Lu 等人[8] 收集了代码翻译数据集 CodeTrans, 并基于该数据集训练了 Transformer 和 CodeBERT 模型. 该数据集可以用于评估模型迁移代码的性能. Guo 等人[12] 在 CodeBERT 基础上采用掩码语言建模任务、数据流的边预测任务和数据流节点-代码变量对齐任务预训练代码文本、自然语言以及代码数据流, 提出了 GraphCodeBERT 模型.

从这些源代码迁移相关工作可知, 目前研究通常从语法树、数据流或者模型的角度提升模型性能, 很少考虑到为源代码迁移任务设计理解代码结构特征的模型. 为此, 本文在 Transformer 模型的基础上提出了一种基于代码语句掩码注意力机制的源代码迁移模型 CSMAT, 以提升 Transformer 模型迁移源代码的性能.

3 模型概述

本文把源代码迁移任务看作是代码序列到代码序列的转换任务. 该任务可以细分为两个阶段: 首先, 模型的编码器理解源代码语句的语法和语义以及语句间的上下文, 进而提取源代码的特征; 其次, 模型的译码器关注并对齐源代码语句, 生成符合目标编程语言语法规则的代码.

本文设计了代码语句的预处理方法, 用自定义的词符<loc>标识代码语句. 为了实现源代码迁移任务的两个阶段, 本文在 Transformer 模型基础上提出了基于代码语句掩码注意力机制的源代码迁移模型 CSMAT, 模型架构如图 2 所示. 本节将详细讲述预处理方法和 CSMAT 模型.

3.1 预处理方法

不同编程语言所要求的书写格式不同, 而神经网络的性能会受到不同格式代码的影响. 为了统一代码格式并标识代码语句, 本文提出预处理方法规范化代码文本格式, 该方法分为两步: (1) 格式化代码文本, 统一代码格式; (2) 加入语句词符, 标识代码语句.

为了统一代码格式, 本文使用了代码格式化工具 Astyle (https://astyle.sourceforge.net/). 该工具能对不同编程语言的代码片段采用统一格式进行格式化. 如表 1 给出了一组 Java 和 C#的代码片段, 表中“原始代码文

本”经格式化后如表中“格式化文本”所示, 可以发现格式化后的代码有着相同的换行样式和缩进样式.

因为 Java 和 C#的缩进样式不会改变代码语义, 本文删除了缩进以压缩代码文本长度. 随后, 本文在每行代码前加入了一个自定义的语句词符<loc>以标识该行代码语句. 由于 CSMAT 模型的译码器是依据文本顺序逐个词符进行预测的神经网络模型, 因此在语句前加入<loc>词符可以帮助译码器先关注并对齐源代码语句, 后依据对齐的源代码语句生成目标代码. 加入语句词符<loc>的代码如表 1 “加入语句词符的文本”所示.

代码语句完成标识以后, 本文删除换行符以进一步压缩代码文本长度, 并在代码文本首尾分别加入<s>和</s>词符以明确代码首尾位置. 最后, 代码文本经过 CSMAT 模型分词器转换为词符序列, 再依据分词器的单词表索引转换为索引序列. 该序列送入 CSMAT 模型后, 模型开始训练并更新参数.

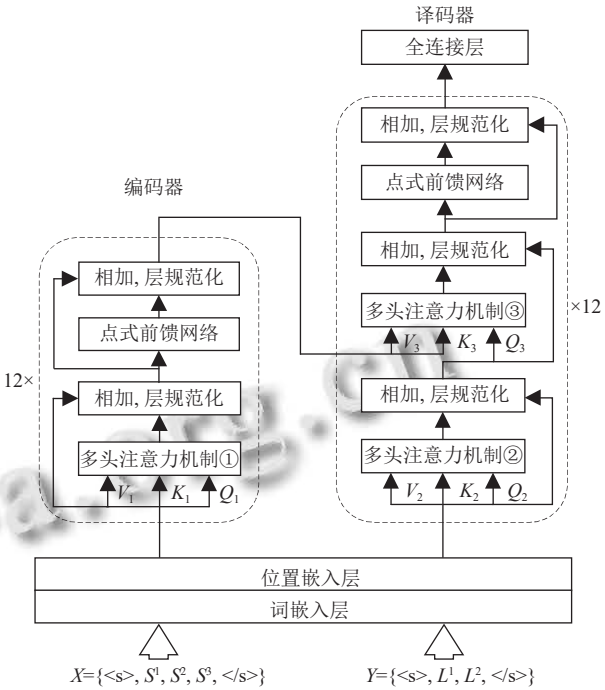


图 2 CSMAT 模型架构图

表 1 一组 Java 和 C#代码片段以及预处理后的文本格式

编程语言	原始代码文本	格式化文本	加入语句词符的文本

Java	public XPath(Parser parser, String path) {
	this.parser = parser;
	this.path = path;
	elements = split(path);
	}
		public XPath(Parser parser, String path) {
		this.parser = parser;
		this.path = path;
		elements = split(path);
		}
			<loc>public XPath(Parser parser, String path) {
			<loc>this.parser = parser;
			<loc>this.path = path;
			<loc>elements = split(path);
			<loc>}

C#	public XPath(Parser parser, string path) {
	this.parser = parser;
	this.path = path;
	elements =Split(path);
	}
		public XPath(Parser parser, string path) {
		this.parser = parser;
		this.path = path;
		elements = Split(path);
		}
			<loc>public XPath(Parser parser, string path) {
			<loc>this.parser = parser;
			<loc>this.path = path;
			<loc>elements = Split(path);
			<loc>}

3.2 多头注意力机制

本文提出的源代码迁移模型 CSMAT 使用了 Transformer 模型. 得益于 Transformer 模型多头注意力机制, 模型能关注并获取同一序列或不同序列中词符之间的特征.

多头注意力机制指的是多个注意力头对一组输入(查询 Q、主键 K、数值 V) 执行按比缩放的点积注意力 (scaled dot product attention) 运算[3]. 该注意力计算查询 Q 和主键 K 的相似度分数, 并把分数结合数值 V, 以表征查询 Q 从主键 K 和数值 V 关注并获取的上下文特征. 以长度为 x_1 的序列 X_1 关注并获取长度为 x_2 的序列 X_2 上下文特征为例, 多头注意力机制计算公式如下:

$$Q_i = X_1 W_i^Q, K_i = X_2 W_i^K, V_i = X_2 W_i^V \tag{1}$$

$$head_i = Softmax\left(\frac{Q_i K_i^{\mathrm{T}}}{\sqrt{d_k}} + M^i\right) V_i \tag{2}$$

$$Attention(Q,K,V,M) = [head_1, \cdots, head_n] W^o \tag{3}$$

其中, $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_h \times d_k}$分别是查询、主键、数值的序列投影矩阵, d_h 为序列表征向量维度, d_k 为注意力头向量维度, $W^o \in \mathbb{R}^{d_h \times d_h}$是多头注意力机制输出的投影矩阵. 此外, M^i是第 i 个注意力头 $x_1 \times x_2$ 大小的掩码矩阵, $M^i_{j,k}$为 0 时表示 X_1 序列的第 j 个词符能够关注到 X_2 序列的第 k 个词符, 为负无穷时则不能关注. 故多头注意力机制可通过掩码引导模型关注特定的词符特征. 为便于表示, 式 (1)–式 (3) 共同表示为$MultiAttn(Q,K,V,M)$

操作.

需要注意的是, 多头注意力机制依赖矩阵乘法操作, 该机制计算时没有考虑词符在序列中的顺序. 为了解决此问题, 序列 X_1 和序列 X_2 的索引序列先经词嵌入层映射至词向量 (word embedding), 再加上位置嵌入层的位置向量 (position embedding)[23] 以明确词符的顺序特征.

3.3 CSMAT 编码器

如图 2 所示, CSMAT 编码器由 12 层编码器层组成, 每层编码器包含多头注意力机制和点式前馈网络 (point wise feed forward network), 并且以残差和层规范化操作组合起来. 给出代码片段 X={<s>, S^1, S^2, S^3, </s>}输入编码器, 其中代码语句 S^i={x_1^i, ⋯, x_n^i}, x_1^i为语句词符<loc>, S^1、S^2 和 S^3 长度分别为 2、4、2, 编码器公式如下:

$$H'_n = LN(MultiAttn(H_{n-1}, H_{n-1}, H_{n-1}, M^{\text{enc}}) + H_{n-1}) \quad (4)$$

$$FFN(x) = Activation\left(xW^1 + b^1\right)W^2 + b^2 \quad (5)$$

$$H_n = LN(FFN(H'_n) + H'_n) \quad (6)$$

其中, LN 是层规范化 (layer normalization) 操作, FFN 是点式前馈网络, W^1, W^2, b^1, b^2是可训练的参数, H_{n-1} 是前一层编码器的输出 (H_0=X), $Activation$ 指的是激活函数, 本文为 GELU 函数. 式 (4) 中 $MultiAttn$ 的查询、主键和数值来源相同时, 可称为自注意力 (self-attention) 机制. 编码器的自注意力机制对应于图 2 的多头注意力机制①.

本文为式 (4) 的自注意力掩码矩阵 M^{enc} 设计了代码语句掩码规则, 该规则应用于所有注意力头. 编码器的自注意力机制能让代码片段 X 中的词符关注并获取代码片段 X 中的任意词符, 这使得注意力连接样式有着正反两个方向, 并且掩码矩阵有着对称矩阵样式. 在这里, 起止词符<s>和</s>均可关注并获取源代码每个词符的特征, 该掩码矩阵样式如图 3(b) 所示. 对于代码语句内注意力的掩码规则, 每条语句 S^i 的词符x_j^i可关注并获取词符x_k^i的特征 (即$x_j^i, x_k^i \in E^S$, E^S 为同一代码语句的词符关联集合), 该掩码矩阵样式如图 3(c) 所示. 该规则引导模型关注每条代码语句的语法和语义. 对于代码语句间注意力的掩码规则, 语句 S^i 的语句词符 x_1^i (即<loc>) 可关注并获取语句 S^j 的语句词符x_1^j (即<loc>) 的特征 (即$x_1^i, x_1^j \in E^{\text{loc}}$, E^{loc} 为不同语句词符<loc>关联集合), 该掩码矩阵如图 3(d) 所示. 该规则引导模型用语句词符<loc>表征代码语句, 同时引导模型表征代码语句之间的上下文特征. 以源代码 S^2 的语句词符为例, 以上 3 种注意力连接样式如图 3(a) 所示. 基于上述规则, 编码器自注意力掩码矩阵 M^{enc} 如图 3(e) 所示.

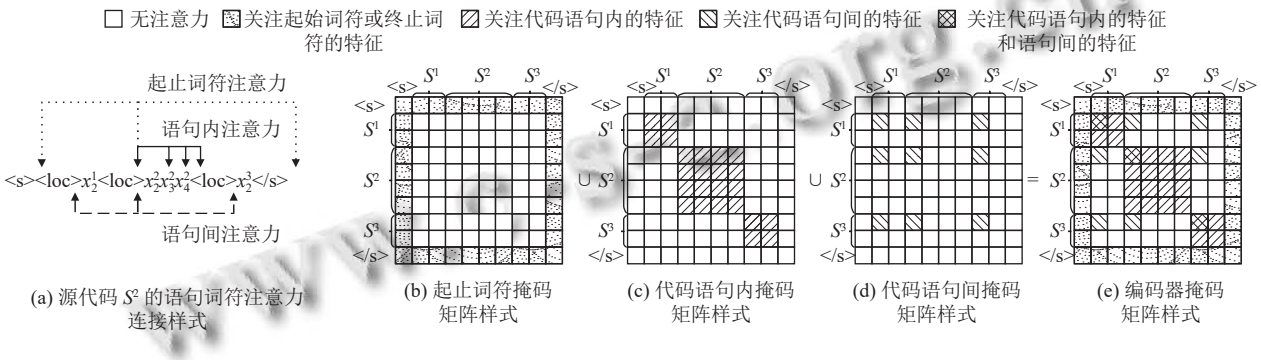


图 3 CSMAT 编码器自注意力连接样式和掩码矩阵样式

基于此, 给出代码片段 X 的索引 i 和 j, 式 (4) 掩码矩阵 M^{enc} 如下表示:

$$M_{i,j}^{\text{enc}} = \begin{cases} 0, & x_i, x_j \in \{<\text{s}>, </\text{s}>\} \text{或} x_i, x_j \in E^S \cup E^{\text{loc}} \\ -\infty, & \text{其他} \end{cases} \quad (7)$$

3.4 CSMAT 译码器

如图 2 所示, CSMAT 译码器由 12 层译码器层组成, 每层译码器层包含两个多头注意力机制和点式前馈网络, 也以残差和层规范化操作组合起来. 给出代码片段 Y={<s>, L^1, L^2, </s>}输入译码器, 其中代码语句 L^i={y_1^i, ⋯, y_n^i}, y_1^i为语句词符<loc>, L^1 和 L^2 长度分别为 5 和 3, 译码器公式如下:

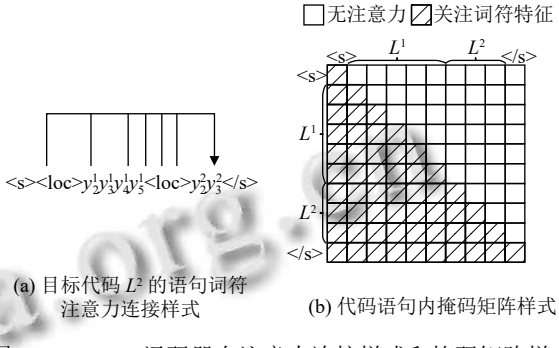
$$G'_n = LN\left(MultiAttn\left(G_{n-1}, G_{n-1}, G_{n-1}, M^{dec1}\right) + G_{n-1}\right) \quad (8)$$

$$G''_n = LN\left(MultiAttn\left(G'_n, H_{\text{enc}}, H_{\text{enc}}, M^{dec2}\right) + G'_n\right) \quad (9)$$

$$G_n = LN\left(FFN\left(G''_n\right) + G''_n\right) \quad (10)$$

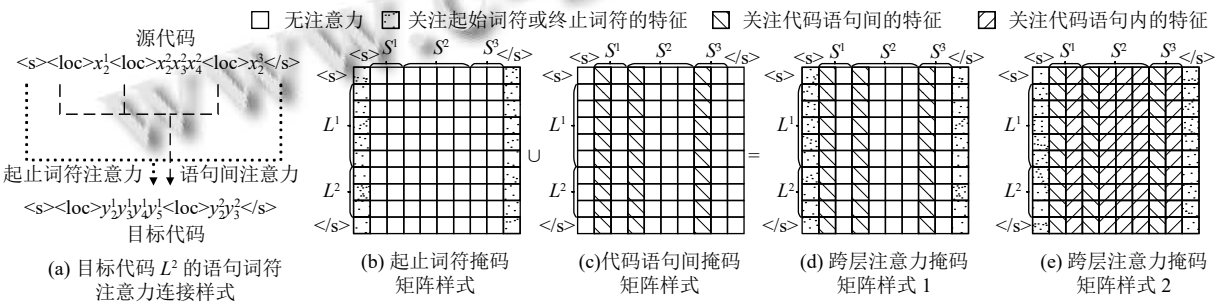
其中, G_{n-1} 是前一层译码器的输出 (G_0=Y), H_{enc} 是编码器最后一层的输出. 式 (8) 为译码器自注意力机制, 对应于图 2 的多头注意力机制②. 该注意力机制在掩码 M^{dec1} 的引导下关注并获取前缀词符的上下文特征, 使译码器预测出未来的词符, 进而生成完整的目标序列. 本文使用了译码器原始的自注意力机制, 该注意力连接样式如图 4(a) 所示, M^{dec1} 掩码矩阵如图 4(b) 所示.

式 (9) 多头注意力机制中查询来自译码器, 主键和数值来源于编码器, 本文称之为跨层注意力 (cross-attention) 机制, 并为其设计了代码语句掩码规则. 跨层注意力机制对应于图 2 的多头注意力机制③. 考虑到跨层注意力需关注源代码具体词符特征, 故该掩码规则只应用于部分注意力头, 原始注意力头掩码矩阵 $M^{dec2,\,style2}$ 如图 5(e) 所示. CSMAT 译码器采用代码语句掩码规则的注意力头数占比为 1/2.



(a) 目标代码 L^2 的语句词符注意力连接样式　　(b) 代码语句内掩码矩阵样式

图 4　CSMAT 译码器自注意力连接样式和掩码矩阵样式



(a) 目标代码 L^2 的语句词符注意力连接样式　(b) 起止词符掩码矩阵样式　(c)代码语句间掩码矩阵样式　(d) 跨层注意力掩码矩阵样式 1　(e) 跨层注意力掩码矩阵样式 2

图 5　CSMAT 译码器跨层注意力连接样式和掩码矩阵样式

跨层注意力机制是目标代码 Y 关注并获取源代码 X 的特征, 这使得注意力连接样式从源代码 X 到目标代码 Y, 且掩码矩阵不再是对称矩阵样式. 在代码语句掩码规则下, 目标代码的每个词符均可关注并获取源代码起止词符<s>和</s>的特征, 该掩码矩阵样式如图 5(b) 所示. 为了对齐源代码语句, 目标代码的每个词符也均可关注并获取源代码 S^i 语句词符 x_1^i (即<loc>) 特征, 该掩码矩阵样式如图 5(c) 所示. 该规则能使译码器通过源代码语句词符<loc>的特征对齐源代码语句, 更好地理解并转换源代码语句的上下文特征. 以上两种注意力连接样式如图 5(a) 所示. 基于上述规则, 代码语句掩码的译码器跨层注意力掩码矩阵 $M^{dec2,\,style1}$ 如图 5(d) 所示.

基于此, 给出目标代码片段 Y 的索引 i 和源代码片段 X 的索引 j, 代码语句掩码的跨层注意力机制掩码矩阵 $M^{dec2,\,style1}$ 如下表示:

$$M_{i,j}^{dec2,style1} = \begin{cases} 0, & x_j \in \{<\text{s}>, </\text{s}>, <\text{loc}>\} \\ -\infty, & \text{其他} \end{cases} \quad (11)$$

4　实验设置

本节将介绍源代码迁移任务的实验配置, 并给出 3 个研究问题评估本文提出方法的有效性.

4.1　研究问题

RQ1: 本文提出的模型在源代码迁移任务中表现如何?

此 RQ 是为了评估现有基于神经网络的源代码迁移模型在两个迁移任务 (即 Java 到 C#和 C#到 Java) 中的性能. 为了减少预训练模型使用外部数据集训练的干扰, 实验将初始化 CSMAT 模型参数和对比模型参数, 并使用相同的训练集进行训练. 其中数据集在第 4.2 节中进行了介绍, 对比模型在第 4.4 节中进行了介绍.

RQ2: 本文方法的不同设置分别对模型的性能有怎样的影响?

此 RQ 是为了从预处理方法、编码器注意力机制和译码器注意力机制 3 个角度验证代码语句掩码注意力机制的有效性. 因此, 实验将对语句词符、编码器注意力机制和译码器注意力机制 3 个部分进行消融实验.

RQ3: 本文提出的方法应用于现有预训练模型后, 对其性能有怎样的影响?

此 RQ 目的是为了评估代码语句掩码注意力机制在预训练模型的适用性. 为了更好地评估注意力机制对模型性能的影响, 本文对比了原始预训练模型、改进了编码器的模型以及同时改进了编码器和译码器的模型. 预训练模型在第 4.4 节中进行了介绍.

4.2 数据集

本文采用的数据集来自文献 [8] 中的 CodeTrans 数据集[12]. 该数据集收集了来自 Lucene、POI、JGit 和 Antlr 等开源项目中的代码函数片段. 这些项目最初基于 Java 语言开发, 随后迁移至 C#语言. 该数据集通过两个编程语言版本中相似的代码函数片段进行收集, 经过抽象语法树和数据流清除冗余的代码片段获得. 该数据集的训练集、验证集和测试集大小分别是 10.3k、0.5k 和 1k. 该数据集具体语法树词符、代码词符和代码语句统计如表 2 所示, 其中具体语法树是通过 tree-sitter (https://tree-sitter.github.io/tree-sitter/) 解析工具生成. 代码分词器的配置以及模型输入长度将在第 4.5 节中介绍.

表 2 具体语法树词符、代码词符和代码语句统计

编程语言	每个具体语法树内的词符			每个代码片段内的词符			每个代码片段内的语句		
	平均数量	分布		平均数量	分布		平均数量	分布	
		<400	≥400		<200	≥200		<20	≥20
Java	99.19	11 529 (97.7%)	271 (2.3%)	51.84	11 488 (97.4%)	312 (2.6%)	5.55	11 508 (97.5%)	292 (2.5%)
C#	122.71	11 417 (96.8%)	383 (3.2%)	66.71	11 445 (97.0%)	355 (3.0%)	6.38	11 411 (96.7%)	389 (3.3%)

4.3 评价指标

本文的实验不仅采用了现有研究广泛使用的 BLEU[3,24–28]、完全匹配率[4,8,12] 和 CodeBLEU[8,12] 评价指标, 还引入了词法正确率来评价模型性能. 指标的介绍如下.

(1) BLEU: 该指标通过预测序列和参考序列的 n-gram 重叠率, 给出 0–100% 之间的分值. 分值越高表示预测序列越接近参考序列.

(2) 完全匹配率: 该指标统计预测代码和参考代码在文本上是否完全匹配, 给出 0–100% 之间的分值. 分值越高表示预测代码和参考代码相同的比例越高.

(3) CodeBLEU: 该指标基于 BLEU 的 n-gram 重叠率计算方法, 计算预测代码的普通 BLEU 成绩、代码关键词分数权重高的 BLEU 成绩、语法树节点序列 BLEU 成绩以及数据流节点序列 BLEU 成绩. 该指标从文本、语法和语义 3 个维度对预测代码评分, 给出 0–100% 之间的分值. 分值越高表示预测代码的质量越接近参考代码.

(4) 词法正确率: 该指标通过分析预测代码的语法树是否存在词法错误节点的方式, 判断预测代码的词法是否正确. 该指标借助 tree-sitter 工具做预测代码的词法分析, 并给出 0–100% 之间的分值. 分值越高表示预测代码越满足编程语言的词法格式.

4.4 对比模型和预训练模型

本节将给出对比模型和预训练模型的介绍. 现有的源代码迁移研究通常采用了基于 Transformer 的外部数据集预训练模型, 并取得了较好的性能. 为了消除此类预训练模型使用外部数据集的干扰, 本文将非预训练的模型作为对比模型, 并把预训练代码数据集的模型作为实验采用的预训练模型.

对比模型有: (1) Naive 模型是常用的对比模型[8,12], 模型直接把输入作为输出; (2) PBSMT 模型是基于短语的统计机器翻译模型[19], 该模型实验结果来自文献 [8]; (3) Pointer-Generator 模型[29] 指的是指针生成网络, 是基于循环神经网络的序列到序列模型, 模型能自动复制源代码词符作为输出的一部分; (4) tree-to-tree 模型[4] 是树到树的转换模型; (5) Transformer 模型[3] 是序列到序列模型, 是目前源代码迁移任务中最佳的非预训练模型.

预训练模型有: (1) CodeBERT 模型[7] 是预训练编程语言和自然语言文本的 Transformer 编码器模型. (2) GraphCodeBERT 模型[12] 是预训练编程语言文本、自然语言文本以及数据流的 Transformer 编码器模型.

4.5 参数设置

参照 CodeBERT 和 GraphCodeBERT 模型的设置, 实验中基于 Transformer 模型均设置为 12 层, 隐藏层设置为 768, 多头注意力数量为 12, 词符位置表征方式设置为位置嵌入层, 并采用添加了<loc>词符的 RoBERTa 分词器对代码分词. 依据第 4.2 节的统计, 基于 Transformer 的模型输入长度和输出长度设置为 320. 此外, 模

型训练阶段的批处理大小设置为 16, 梯度累积步数设置为 2, 损失函数设置为标签平滑参数是 0.1 的交叉熵函数[3], 并采用带有学习率为 1E–5 线性调度函数的 AdamW 优化器[30] 优化模型参数. 实验中模型运行环境为带有 Python 3.9、PyTorch 1.13.1 和 Transformers 4.25.1 的 Linux 服务器 (Ubuntu 20.04.3 LTS, Intel Core i9-9900KF @ 3.60 GHz, Nvidia RTX2080Ti 11 GB).

5 实验结果

5.1 RQ1: 本文提出的模型在源代码迁移任务中表现如何?

为了回答此 RQ, 本节将对 CSMAT 和对比模型在测试集的评价指标结果进行比较. 实验对比结果如表 3 所示. 可以发现 CSMAT 的源代码迁移性能均优于其他模型. PBSMT 受限于模型架构, 难以通过代码文本理解源代码, 导致模型性能不及 Transformer. Pointer-Generator 模型尽管可以复制源代码的词符作为输出, 但是该模型受限于循环神经网络的长序列理解和生成难题[31], 导致模型性能同样不及 Transformer. Tree-to-tree 模型能够通过语法树较好地理解代码的结构特征, 并且 C#到 Java 任务中词法正确率优于 Pointer-Generator, 但其性能依然不及 Transformer. 从第 4.2 节对数据集词符数量的统计可知, 该模型输入的具体语法树词符数量多于代码文本词符数量, 这表明该模型难以理解并生成具有复杂结构特征的代码. 本文提出的模型 CSMAT 优于 Transformer. 从模型架构上分析, 该模型得益于代码语句掩码注意力机制, 能够在迁移目标代码时, 有效地理解并对齐源代码语句的语法和语义, 提升源代码迁移性能.

表 4 给出了 CSMAT 和部分对比模型的输出样例. 可以发现 Pointer-Generator 生成的函数声明语句中, 返回值类型和代码语法错误. 本文认为这是因循环神经网络的长序列理解和生成难题所导致[31]. Tree-to-tree 生成的函数有两个相同的形参, 并且生成了上下文无关的“invocation_expression”变量. 联系表 2 具体语法树词符数量可推测, 该模型难以理解并迁移词符较多的源代码具体语法树. Transformer 模型生成了语法错误的函数调用语句, 而 CSMAT 输出结果和参考代码相同. 该结果表明, CSMAT 在代码语句掩码的引导下, 能有效地理解并对齐源代码语句的语法和语义, 具有最佳的迁移性能.

RQ1 总结: 在源代码迁移任务中, 本文提出的基于代码语句掩码注意力机制的源代码迁移模型 CSMAT 相比于对比模型具有最佳的迁移性能.

表 3 CSMAT 和对比模型在两个源代码迁移任务的结果 (%)

模型	Java到C#				C#到Java			
	BLEU	完全匹配率	CodeBLEU	词法正确率	BLEU	完全匹配率	CodeBLEU	词法正确率
Naive	18.54	0	—	—	18.69	0	—	—
PBSMT	43.53	12.5	42.71	—	40.06	16.1	43.48	—
Pointer-Generator	26.18	13.8	43.87	48.5	27.84	20.5	44.88	48.6
Tree-to-tree	36.34	3.4	42.13	45.6	32.09	4.4	43.86	65.2
Transformer	60.99	37.9	66.88	89.0	55.41	40.6	62.20	89.2
CSMAT	**62.74**	**39.5**	**67.82**	**90.2**	**59.09**	**41.5**	**65.14**	**89.4**

表 4 CSMAT 和部分对比模型在 Java 到 C#任务的输出样例

来源	代码文本
源代码	public StringBuilder insert(int offset, int i) { insert0(offset, Integer.toString(i)); return this; }
参考代码	public java.lang.StringBuilder insert(int offset, int i) { insert0(offset, System.Convert.ToString(i)); return this; }
Pointer-Generator	public boolean insertBuilder offset, long l) { insert0(offset, Convert.ToString(l)); return this; }
Tree-to-tree	public java.lang.StringBuilder insert(int offset, int offset) { insert0(offset , System.Convert.ToString(l, invocation_expression)); return this; }
Transformer	public java.lang.StringBuilder insert(int offset, int i) { insert0(offset, java.lang.StringBuilder return this; }
CSMAT	public java.lang.StringBuilder insert(int offset, int i) { insert0(offset, System.Convert.ToString(i)); return this; }

5.2 RQ2: 本文方法的不同设置分别对模型的性能有怎样的影响?

表 5 展示了本文模型 CSMAT 对语句词符、编码器注意力机制和译码器注意力机制的消融实验结果. 为了方便表示, 模型名称前加入“LOC-”表示模型的输入包含代码语句<loc>词符, 模型名称后的下标“enc”表

示模型的编码器采用了代码语句掩码的自注意力机制, “-Ndec”表示模型的译码器采用了代码语句掩码的跨层注意力机制, “-Ndec”中的“N”表示译码器跨层注意力机制采用了代码语句掩码规则的注意力头数占比.

表 5 CSMAT 和消融实验的模型在两个源代码迁移任务的结果 (%)

模型	Java到C#				C#到Java			
	BLEU	完全匹配率	CodeBLEU	词法正确率	BLEU	完全匹配率	CodeBLEU	词法正确率
Transformer	60.99	37.9	66.88	89.0	55.41	40.6	62.20	89.2
LOC-Transformer	60.39	37.5	66.49	88.7	55.84	40.8	62.31	89.0
LOC-Transformer$_{enc}$	62.53	38.8	**68.42**	89.2	58.80	**42.4**	64.64	89.3
LOC-Transformer$_{enc\text{-}1/4dec}$	62.22	38.9	68.09	90.1	58.64	41.4	64.78	88.4
CSMAT (LOC-Transformer$_{enc\text{-}1/2dec}$)	**62.74**	**39.5**	67.82	**90.2**	**59.09**	41.5	**65.14**	**89.4**
LOC-Transformer$_{enc\text{-}3/4dec}$	61.81	38.1	67.86	90.1	57.59	39.4	63.71	88.4

从 LOC-Transformer 和 Transformer 结果对比可知, 将语句词符<loc>添加至 Transformer 模型的输入后, Java 到 C#任务所有指标结果分别降低了 0.6、0.4、0.4 和 0.3 个分值, C#到 Java 任务词法正确率降低了 0.2 个分值, BLEU、完全匹配率和 CodeBLEU 分别提升了 0.43、0.2 和 0.11 个分值. 从该结果可以推测输入加入语句词符<loc>的模型不仅要理解代码的语法和语义, 还要理解语句词符的语法规则, 从而影响了模型的迁移性能. 当模型的编码器引入代码语句掩码注意力机制后, LOC-Transformer$_{enc}$ 在两个源代码迁移任务中所有指标成绩均优于 LOC-Transformer 和 Transformer. 该结果表明编码器中的代码语句掩码注意力机制, 能引导模型更好地理解源代码语句的语法和语义以及语句间的上下文特征, 同时降低了模型理解语句词符<loc>语法规则的训练成本.

为了验证本文译码器注意力机制的性能, 本文还从该机制中采用代码语句掩码规则的注意力头数占比进行了消融实验. 对比表 5 中译码器 3 种不同注意力头数占比的模型 (LOC-Transformer$_{enc\text{-}1/4dec}$、CSMAT、LOC-Transformer$_{enc\text{-}3/4dec}$), 可以发现 CSMAT 在两个代码迁移任务的多个指标上取得了最佳的成绩. 而另两个模型的性能较差, 甚至 C#到 Java 任务中 Transformer$_{enc\text{-}1/4dec}$ 和 LOC-Transformer$_{enc\text{-}3/4dec}$ 在词法正确率指标上低于 Transformer. 从模型架构的角度来分析两个性能较差的模型, 可以发现译码器中代码语句掩码注意力头数占比较大 (或较小), 模型难以获取更加丰富的源代码词符 (语句) 特征, 导致模型迁移性能较差.

实验针对译码器中采用代码语句掩码规则不同注意力头数占比的模型, 给出了输出样例, 如表 6 所示. 其中 CSMAT 生成了正确的代码文本. Transformer$_{enc\text{-}1/4dec}$ 和 LOC-Transformer$_{enc\text{-}3/4dec}$ 生成了重复的代码语句, 本文认为这是因为模型的译码器对源代码语句特征或者源代码词符特征理解性能不佳所致. 此外, LOC-Transformer$_{enc}$ 生成的最后一条赋值语句存在偏差. 本文推测该模型的译码器没有掩码引导模型对齐源代码的语句, 使得模型输出了源代码对应位置上相邻且出现频率最多的“path”变量作为语句的赋值对象.

表 6 CSMAT 和译码器消融实验的模型在 C#到 Java 任务的输出样例

来源	代码文本
源代码	public XPath(Parser parser, string path) { this.parser = parser; this.path = path; elements = Split(path); }
参考代码	public XPath(Parser parser, String path) { this.parser = parser; this.path = path; elements = split(path); }
LOC-Transformer$_{enc}$	public XPath(Parser parser, String path) { this.parser = parser; this.path = path; path = split(path); }
LOC-Transformer$_{enc\text{-}1/4dec}$	public XPath(Parser parser, String path) { this.path = path; this.path = path; elements = split(path); }
CSMAT (LOC-Transformer$_{enc\text{-}1/2dec}$)	public XPath(Parser parser, String path) { this.parser = parser; this.path = path; elements = split(path); }
LOC-Transformer$_{enc\text{-}3/4dec}$	public XPath(Parser parser, String path) { this.path = path; this.path = path; elements = split(path); }

RQ2 总结: 本文提出的编码器代码语句掩码注意力机制能提升模型迁移源代码的性能. 此外, 本文提出的译码器的代码语句掩码规则的注意力头数占比为 1/2 时, 模型迁移源代码的性能最佳.

5.3 RQ3: 本文提出的方法应用于现有预训练模型后, 对其性能有怎样的影响?

本节实验采用预训练模型如第 4.4 节所描述. 为了便于对比, 这些模型微调阶段的参数设置与第 4.5 节的

描述保持一致. 其中 GraphCodeBERT 模型的输入仅为代码文本以减少数据流特征对实验结果的影响. 为了方便表示, 预训练模型命名规则如第 5.2 节所描述.

表 7 给出了预训练模型在两个任务的改进结果. 从 CodeBERT 改进模型结果可以发现, 改进了编码器和译码器的 LOC-CodeBERT$_{\text{enc-1/2dec}}$ 在 Java 到 C#任务中完全匹配率和词法正确率相比于 CodeBERT 提升了 0.5 和 0.7 个分值, 而 BLEU 和 CodeBLEU 降低了 0.09 和 0.18 个分值; 在 C#到 Java 任务中词法正确率指标与 CodeBERT 相当, 而在 BLEU、完全匹配率和 CodeBLEU 的分值均有提升. 结合 CodeBERT 预训练方式分析可知, 模型在微调阶段引入了代码语句特征能提升源代码迁移任务的性能, 而语句词符<loc>的加入使模型需重新理解代码文本的语法形式, 以至于模型在部分指标有较小的性能损失.

对于改进的 GraphCodeBERT 模型, LOC-GraphCodeBERT$_{\text{enc-1/2dec}}$ 在 Java 到 C#任务中词法正确率相比于原始模型 GraphCodeBERT 提升了 0.8 个分值, 而 BLEU、完全匹配率和 CodeBLEU 分别低于原始模型 0.94、1.0 和 0.41 个分值; 而在 C#到 Java 任务中所有指标成绩均低于原始模型. LOC-GraphCodeBERT$_{\text{enc}}$ 在 Java 到 C#任务中完全匹配率低于原始模型 0.1 个分值, 而 BLEU、CodeBLEU 和词法正确率相比原始模型均有提升; 在 C#到 Java 任务中所有指标均优于原始模型. 以上结果表明本文提出的编码器注意力机制能提升预训练模型源代码迁移性能. 考虑到 GraphCodeBERT 在预训练阶段训练了代码数据流特征, 本文分析该模型译码器理解源代码时, 相比于对齐源代码语句, 其更倾向于理解源代码中变量的声明和调用方式.

表 8 给出了改进后的两个预训练模型 (LOC-CodeBERT$_{\text{enc-1/2dec}}$ 和 LOC-GraphCodeBERT$_{\text{enc}}$) 及其原始预训练模型的输出样例. 可以发现 CodeBERT 生成的返回值语句中存在重复的表达式, GraphCodeBERT 生成了错误的函数名称, 而 LOC-CodeBERT$_{\text{enc-1/2dec}}$ 和 LOC-GraphCodeBERT$_{\text{enc}}$ 输出结果和参考代码相同. 本文认为, CodeBERT 和 GraphCodeBERT 难以准确地对齐源代码, 导致 CodeBERT 重复生成相同的表达式, GraphCodeBERT 迁移了源代码“readUByte”函数名称. 改进后的两个模型在代码语句掩码的引导下, 能更好地理解和对齐源代码语句, 具有更好的性能.

表 7 本文提出的方法应用于预训练模型在两个源代码迁移任务的结果 (%)

模型	Java到C#				C#到Java			
	BLEU	完全匹配率	CodeBLEU	词法正确率	BLEU	完全匹配率	CodeBLEU	词法正确率
CodeBERT	**77.55**	52.7	**80.69**	94.7	73.57	55.5	77.67	**95.4**
LOC-CodeBERT$_{\text{enc}}$	76.73	**54.2**	80.22	93.5	73.25	**57.7**	77.28	94.9
LOC-CodeBERT$_{\text{enc-1/2dec}}$	77.46	53.2	80.51	**95.4**	**74.47**	57.3	**78.48**	**95.4**
GraphCodeBERT	78.84	**55.1**	81.16	94.0	75.25	58.3	78.42	94.4
LOC-GraphCodeBERT$_{\text{enc}}$	**78.85**	55.0	**81.84**	**95.7**	**75.83**	**60.0**	**79.58**	**94.9**
LOC-GraphCodeBERT$_{\text{enc-1/2dec}}$	77.90	54.1	80.75	94.8	73.32	57.1	77.45	94.0

表 8 本文提出的方法应用于预训练模型在 C#到 Java 任务的输出样例

来源	代码文本
源代码	public int readUShort() { int ch1 = readUByte(); int ch2 = readUByte(); return (ch2 << 8) + (ch1 << 0); }
参考代码	public int ReadUShort() { int ch1 = ReadUByte(); int ch2 = ReadUByte(); return (ch2 << 8) + (ch1 << 0); }
CodeBERT	public override int ReadUShort() { int ch1 = ReadUByte(); int ch2 = ReadUByte(); return (ch2 << 8) + (ch2 << 8) + (ch1 << 0); }
LOC-CodeBERT$_{\text{enc-1/2dec}}$	public int ReadUShort() { int ch1 = ReadUByte(); int ch2 = ReadUByte(); return (ch2 << 8) + (ch1 << 0); }
GraphCodeBERT	public int ReadUByte() { int ch1 = ReadUByte(); int ch2 = ReadUByte(); return (ch2 << 8) + (ch1 << 0); }
LOC-GraphCodeBERT$_{\text{enc}}$	public int ReadUShort() { int ch1 = ReadUByte(); int ch2 = ReadUByte(); return (ch2 << 8) + (ch1 << 0); }

RQ3 总结: 本文提出的方法可以提升预训练模型迁移源代码的性能, 具有较好的适用性. 对于预训练代码数据流的模型, 在微调阶段引入编码器代码语句掩码注意力机制即可提升模型的性能.

6 结论与展望

本文提出了基于代码语句掩码注意力机制的源代码迁移模型 CSMAT. 该模型在代码语句掩码注意力机制的引导下, 编码器能够理解源代码语句的语法和语

义以及语句间的上下文特征, 译码器能够对齐源代码语句和上下文特征, 从而提升模型迁移源代码的性能. 本文采用了基于真实项目收集的 CodeTrans 数据集, 设计了 Java 到 C#和 C#到 Java 两个源代码迁移任务. 实验结果显示, 本文模型 CSMAT 的性能均优于对比模型, 并且通过消融实验验证了代码语句掩码注意力机制的有效性. 此外, 本文的方法还提升了预训练模型的代码迁移性能, 验证了代码语句掩码注意力机制在预训练模型的适用性.

在未来的研究中, 我们还将使用代码分析工具提取更多有效的代码特征, 并尝试扩展代码函数级别迁移任务至程序级别迁移任务, 进一步提升源代码迁移模型的性能.

参考文献

1 Roziere B, Lachaux MA, Chanussot L, *et al*. Unsupervised translation of programming languages. Proceedings of the 34th International Conference on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2020. 20601–20611.

2 Cho K, Van Merriënboer B, Gulcehre C, *et al*. Learning phrase representations using RNN encoder-decoder for statistical machine translation. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing. Doha: Association for Computational Linguistics, 2014. 1724–1734. [doi: 10.3115/v1/D14-1179]

3 Vaswani A, Shazeer N, Parmar N, *et al*. Attention is all you need. Proceedings of the 31st International Conference on Neural Information Processing Systems. Long Beach: Curran Associates Inc., 2017. 6000–6010.

4 Chen XY, Liu C, Song D. Tree-to-tree neural networks for program translation. Proceedings of the 32nd International Conference on Neural Information Processing Systems. Montreal: Curran Associates Inc., 2018. 2552–2562.

5 Shiv VL, Quirk C. Novel positional encodings to enable tree-based transformers. Proceedings of the 33rd International Conference on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2019. 1082.

6 Jiang X, Zheng ZR, Lyu C, *et al*. TreeBERT: A tree-based pre-trained model for programming language. Proceedings of the 37th Conference on Uncertainty in Artificial Intelligence. PMLR, 2021. 54–63.

7 Feng ZY, Guo DY, Tang DY, *et al*. CodeBERT: A pre-trained model for programming and natural languages. Findings of the Association for Computational Linguistics: EMNLP 2020. Association for Computational Linguistics, 2020.1536-1547.[doi:10.18653/v1/2020.findings-emnlp.139]

8 Lu S, Guo DY, Ren S, *et al*. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. arXiv:2102.04664, 2021.

9 Hu X, Li G, Xia X, *et al*. Deep code comment generation. Proceedings of the 26th Conference on Program Comprehension. Gothenburg: Association for Computing Machinery, 2018. 200–210. [doi: 10.1145/3196321.3196334]

10 Allamanis M, Peng H, Sutton C. A convolutional attention network for extreme summarization of source code. Proceedings of the 33rd International Conference on Machine Learning. New York: PMLR, 2016. 2091–2100.

11 Allamanis M, Barr ET, Devanbu P, *et al*. A survey of machine learning for big code and naturalness. ACM Computing Surveys, 2019, 51(4): 81. [doi: 10.1145/3212695]

12 Guo DY, Ren S, Lu S, *et al*. GraphCodeBERT: Pre-training code representations with data flow. Proceedings of the 9th International Conference on Learning Representations. ICLR, 2021.

13 Hindle A, Barr ET, Gabel M, *et al*. On the naturalness of software. Communications of the ACM, 2016, 59(5): 122–131. [doi: 10.1145/2902362]

14 Zhang J, Wang X, Zhang HY, *et al*. A novel neural source code representation based on abstract syntax tree. Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE). Montreal: IEEE. 2019. 783–794. [doi: 10.1109/ICSE.2019.00086]

15 Yasumatsu K, Doi N. SPiCE: A system for translating Smalltalk programs into a C environment. IEEE Transactions on Software Engineering, 1995, 21(11): 902–912. [doi: 10.1109/32.473219]

16 Bravenboer M, Kalleberg KT, Vermaas R, *et al*. Stratego/XT 0.17. A language and toolset for program transformation. Science of Computer Programming, 2008, 72(1–2): 52–70.

17 石学林, 张兆庆, 武成岗. 自动化的 Cobol 2 Java 遗产代码迁移技术. 计算机工程, 2005, 31(12): 67–69.

18 刘静. Verilog-to-MSVL 程序翻译软件的实现 [硕士学位论文]. 西安: 西安电子科技大学, 2014.

19 Koehn P, Och FJ, Marcu D. Statistical phrase-based translation. Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics. 2003. 127–133.

20 Nguyen AT, Nguyen TT, Nguyen TN. Lexical statistical

machine translation for language migration. Proceedings of the 9th Joint Meeting on Foundations of Software Engineering. Saint Petersburg: Association for Computing Machinery, 2013. 651–654. [doi: 10.1145/2491411.2494584]

21 Karaivanov S, Raychev V, Vechev M. Phrase-based statistical translation of programming languages. Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. Portland: Association for Computing Machinery, 2014. 173–184. [doi: 10.1145/2661136.2661148]

22 Nguyen AT, Nguyen TT, Nguyen TN. Divide-and-conquer approach for multi-phase statistical migration for source code (T). Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). Lincoln: IEEE, 2015. 585–596. [doi: 10.1109/ASE.2015.74]

23 Devlin J, Chang MW, Lee K, *et al*. BERT: Pre-training of deep bidirectional transformers for language understanding. Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers). Minneapolis: Association for Computational Linguistics, 2019. 4171–4186. [doi: 10.18653/v1/N19-1423]

24 Cho K, Van Merriënboer B, Bahdanau D, *et al*. On the properties of neural machine translation: Encoder-decoder approaches. Proceedings of the 8th Workshop on Syntax, Semantics and Structure in Statistical Translation. Doha: Association for Computational Linguistics, 2014. 103–111. [doi: 10.3115/v1/W14-4012]

25 Papineni K, Roukos S, Ward T, *et al*. BLEU: A method for automatic evaluation of machine translation. Proceedings of the 40th Annual Meeting on Association for Computational Linguistics. Philadelphia: Association for Computational Linguistics, 2002. 311–318. [doi: 10.3115/1073083.1073135]

26 Li Z, Wu YH, Peng B, *et al*. SeCNN: A semantic CNN parser for code comment generation. Journal of Systems and Software, 2021, 181: 111036. [doi: 10.1016/j.jss.2021.111036]

27 Yang G, Liu K, Chen X, *et al*. CCGIR: Information retrieval-based code comment generation method for smart contracts. Knowledge-based Systems, 2022, 237: 107858. [doi: 10.1016/j.knosys.2021.107858]

28 Li Z, Wu YH, Peng B, *et al*. SeTransformer: A Transformer-based code semantic parser for code comment generation. IEEE Transactions on Reliability, 2023, 72(1): 258–273.

29 See A, Liu PJ, Manning CD. Get to the point: Summarization with pointer-generator networks. Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Vancouver: Association for Computational Linguistics, 2017. 1073–1083. [doi: 10.18653/v1/P17-1099]

30 Loshchilov I, Hutter F. Decoupled weight decay regularization. Proceedings of the 7th International Conference on Learning Representations. New Orleans: ICLR, 2019.

31 Bengio Y, Simard P, Frasconi P. Learning long-term dependencies with gradient descent is difficult. IEEE Transactions on Neural Networks, 1994, 5(2): 157–166. [doi: 10.1109/72.279181]

(校对责编: 牛欣悦)