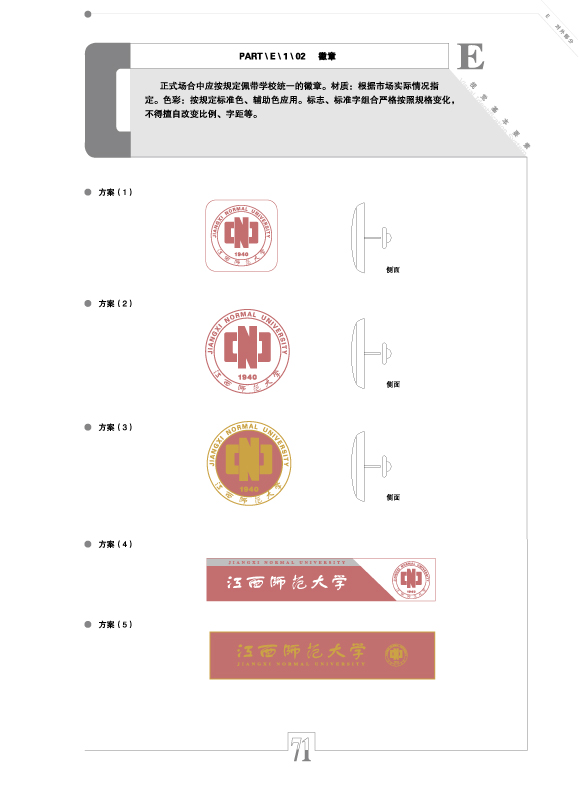
分 类 号： 密 级：

学校代码： 10414 学 号： 202041600066



**硕士研究生学位论文**

**基于预训练掩码语言模型的**

**程序类型推理研究**

**Program Type Inference Based on Pre-trained Mask Language Model**

**袁志强**

院 所：计算机信息工程学院 导师姓名：左正康、黄箐

学科专业：软件工程 研究方向：智能化软件

二〇二三年五月

**独 创 性 声 明**

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名： 签字日期： 年 月 日

**学位论文版权使用授权书**

本学位论文作者完全了解江西师范大学研究生院有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权江西师范大学研究生院可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后适用本授权书）

学位论文作者签名： 导师签名：

签字日期： 年 月 日 签字日期： 年 月 日

摘 要

代码片段当中常涉及未声明接收对象和非完全限定名。将这些未声明接收对象和非完全限定名解析为对应的完整数据类型名（即：类型推理），是有效利用代码片段当中知识的先决条件。为了推理出代码片段中的完整数据类型名（Fully-qualified name），现有工作均基于符号知识库，采用关键字匹配式的“查字典”策略。然而，符号知识库的构建依赖解析可编译的代码文件。这种可编译的开销会限制符号知识库中所存储的完整数据类型名和代码上下文的数量。当使用关键字匹配式的策略进行类型推理时，会由于符号知识库中知识的有限性而出现Out-of-vocabulary问题，即：因查找的完整数据类型名未存储在符号知识库中而返回空值。为了解决这一问题，本文提出将经过提示调优的预训练掩码语言模型作为类型推理的神经知识库（类型推理模型），并采用“完型填空”式的策略以进行类型推理。相比符号知识库的构建存在可编译的开销，神经知识库的构建因基于代码自然性将代码视作为文本而不存在可编译的开销。此外，为了提高开发人员在实际开发环境中复用代码片段的效率，本文设计了两款基于类型推理模型的载体插件，分别是Integrated development environments（IDE）插件和WEB插件。

实验部分，本文从有效性、实用性和能力探究三个方面对本文所提出的类型推理模型进行系统化的评估。有效性实验结果表明，本文所提出的类型推理模型具有低资源学习能力，即：仅使用10%数据对掩码语言模型进行微调便取得了优异的类型推理性能。实用性实验结果表明，类型推理模型的性能优于现有最新的类型推理工具，并且能够很好的处理现有工作中所存在的Out-of-vocabulary问题。能力边界探究实验结果表明，类型推理模型具备泛化能力（可在不同的编程语言中进行类型推理）以及混合能力（可为不同的编程语言提供一个统一的混合类型推理模型）。

**关键词：**代码片段；类型推理；完整数据类型名；预训练掩码语言模型；提示调优

Abstract

Code snippets often involve undeclared receiver objects and non-fully qualified names. Resolving the undeclared received objects and non-fully qualified names into the corresponding fully qualified names (i.e., type inference) is a prerequisite for efficiently using the knowledge in code snippets. To infer the Fully qualified name (FQN) in the code snippet, existing work is based on the symbol knowledge base and adopts the keyword matching "dictionary lookup" strategy. However, constructing a symbolic knowledge base depends on parsing compilable code files. This compilable overhead limits the number of fully qualified names and code contexts stored in the symbolic knowledge base. When type inference is made by using keyword-matching strategies, the out-of-of-Vocabulary problem occurs due to the limited knowledge in the symbolic knowledge base, i.e., null values are returned because the FQN query is not stored in the symbolic knowledge base. To solve this problem, this paper proposes to use the prompt-tuned Code masked language model as the neural knowledge base for type inference (type inference model) and adopts the fill-in-blank strategy. Compared with the construction of the symbolic knowledge base, there is no compilable overhead in the construction of the neural knowledge base because the code is treated as the text based on the naturalness of the code. In addition, to improve developers' efficiency in reusing code snippets in the actual development environment, two plug-ins based on the type inference model are designed in this paper, namely the Integrated development environments (IDE) plug-in and the WEB plug-in.

In the experimental part, this paper systematically evaluates the type inference model proposed in this paper from three aspects: effectiveness, practicability, and capability exploration. Effectiveness experiment results show that the proposed type reasoning model has low resource learning ability, i.e., excellent type reasoning performance can be achieved by prompt-tuning the mask language model with only 10% data. The practical experiment results show that the performance of the type inference model is better than the latest type inference tools, and it can deal with the out-of-of-vocabulary problems in existing work well. The capability exploration experiment results show that the type inference model has the generalization ability (i.e., the type inference model can be used in the different programming languages for the type inference). And hybrid capabilities (i.e., the ability to provide a uniform type inference model for different programming languages for the type inference).

**Keywords**: Code snippet; Type Inference; Fully-qualified name; Pre-trained masked language model; Prompt tuning

目 录

[摘 要 III](#_Toc126014383)

[Abstract IV](#_Toc126014384)

[目 录 V](#_Toc126014385)

[1 绪 论 1](#_Toc126014386)

[1.1 研究背景及意义 1](#_Toc126014387)

[1.2 研究现状及问题 1](#_Toc126014388)

[1.2.1 类型推理方法研究及问题 1](#_Toc126014389)

[1.2.2 预训练语言模型适配方法研究 3](#_Toc126014390)

[1.2.3 提示文本的设计研究 4](#_Toc126014391)

[1.3 研究内容及创新点 5](#_Toc126014392)

[1.4 论文章节安排 5](#_Toc126014393)

[2 研究路线 7](#_Toc126014394)

[2.1 问题定义 7](#_Toc126014395)

[2.2 技术路线 8](#_Toc126014396)

[2.3 本章小结 10](#_Toc126014397)

[3 神经知识库激活方法研究 11](#_Toc126014398)

[3.1 语言模型预训练方法设计 11](#_Toc126014399)

[3.2 提示调优策略研究 13](#_Toc126014400)

[3.2.1 提示文本设计 13](#_Toc126014401)

[3.2.2 提示调优任务目标设计 16](#_Toc126014402)

[3.3 本章小结 16](#_Toc126014403)

[4 基于神经知识库的类型推理 17](#_Toc126014404)

[4.1 完型填空式的类型推理方法设计 17](#_Toc126014405)

[4.2 类型推理模型载体设计 19](#_Toc126014406)

[4.2.1 WEB载体插件设计 20](#_Toc126014407)

[4.2.2 IDE 载体插件设计 21](#_Toc126014408)

[4.3 本章小结 22](#_Toc126014409)

[5 实验及结果分析 23](#_Toc126014410)

[5.1 实验环境及参数 23](#_Toc126014411)

[5.2 类型推理模型的有效性评估 24](#_Toc126014412)

[5.2.1 有效性评估动机 24](#_Toc126014413)

[5.2.2 实验数据与评价指标 24](#_Toc126014414)

[5.2.3 内部因素对类型推理模型的影响 26](#_Toc126014415)

[5.2.4 外部因素对类型推理模型的影响 30](#_Toc126014416)

[5.2.5 实验小结 32](#_Toc126014417)

[5.3 类型推理模型的实用性评估 33](#_Toc126014418)

[5.3.1 实用性评估动机 33](#_Toc126014419)

[5.3.2 实验数据与评价指标 33](#_Toc126014420)

[5.3.3 最新类型推理方法对比 34](#_Toc126014421)

[5.3.4 极限能力评估 36](#_Toc126014422)

[5.3.5 失败案例分析 38](#_Toc126014423)

[5.3.6 实验小结 40](#_Toc126014424)

[5.4 类型推理模型的能力边界探究 41](#_Toc126014425)

[5.4.1 能力边界探究动机 41](#_Toc126014426)

[5.4.2 实验数据与评价指标 41](#_Toc126014427)

[5.4.3 泛化能力探究 42](#_Toc126014428)

[5.4.4 混合能力探究 43](#_Toc126014429)

[5.4.5 实验小结 44](#_Toc126014430)

[5.5 本章小结 44](#_Toc126014431)

[6 总结与展望 45](#_Toc126014432)

[6.1 总结 45](#_Toc126014433)

[6.2 展望 45](#_Toc126014434)

[参考文献 47](#_Toc126014435)

[致 谢 53](#_Toc126014436)

[在读期间公开发表论文（著）及科研情况 54](#_Toc126014437)

1 绪 论

1.1 研究背景及意义

在日常编码的过程中，开发人员广泛地重用软件框架中的程序接口（Application Programming Interface），以节省开发过程中所需的时间和精力。这需要在软件开发过程中不断学习新的Application Programming Interface（API）的使用方法。然而，程序文档的不充分性和时效性，会阻碍开发人员对于API的学习[1，2]。为此，开发人员更加偏向求助盛行于各种问答网站（如Stack Overflow）当中的代码片段，以解决他们在实际开发过程中所遇到的API使用问题[3]。为此，有效地解析并利用代码片段当中的知识是非常重要的[4，5]。

然而，代码片段当中包含了不完整的数据类型名，即未声明接收对象和非完全限定名称。对于开发人员而言，在复用该类代码片段时，会遭遇编译错误“symbol cannot be resolved”。此外，如果未将未声明接收对象和非完全限定名称解析成对应的完整数据类型名（Fully-qualified name），代码片段只能被当作自然语言文本处理，这会限制其它与代码相关技术在软件工程领域当中的应用。例如，程序分析技术[6]，将无法从代码片段获取API的完整数据类型，或者准确的代码语法结构；代码搜索技术[7]，将无法准确地推荐出与搜索语句相关的代码。此外，研究[8，9]表明，代码片段当中常包含API误用甚至恶意行为。如果未将未声明接收对象和非完全限定名解析成对应的完整数据类型名，代码漏洞分析技术[10-12]将无法检测出代码片段当中存在的不安全行为，进而造成严重的经济危机。为此，推断出未声明接收对象和非完全限定名所对应的完整数据类型，对于利用代码片段当中的知识是非常重要的。

1.2 研究现状及问题

将代码片段中的未声明接收对象和非完全限定名，解析为对应的完整数据类型（即，类型推理），是重用代码片段并挖掘其中知识的先决条件。本文的任务目标是解决现有类型推理方法[13-16]在类型推理任务中所存在的Out-of-vocabulary问题。本文的关键问题是如何将包含海量先验知识的预训练语言模型运用到类型推理任务当中。为此，此章节将从现有类型推理方法研究及问题进行介绍，预训练语言模型适配方法研究，以及提示文本的设计研究三个方面对研究现状进行介绍。

1.2.1 类型推理方法研究及问题

为了推理出代码片段当中所缺失的完整数据类型名，现有的研究方法[13-16]均采用字符匹配式的“查字典”策略。该策略依赖构建的符号知识库，将代码片段中的API映射到符号知识库当中所对应的完整数据类型名。例如，Khaled等人[15]提出一种基于代码上下文进行类型推理的方法。具体而言，该方法将代码片段中的API和此API对应的上下文信息，组成查询语句。然后，通过此查询语句从构建的符号知识库中匹配得到API的完整数据类型名。对于图1-1代码片段中第一行的API“List”。Khaled等人将图1-1中的①-④行代码作为List的上下文信息，然后和List共同构成查询语句。此查询语句通过关键字匹配的策略，在构建的符号知识库中查找List所对应的完整数据类型名。再例如，Dong等人[16]使用API之间的约束关系作为所要推理API的上下文信息。同样，对于图1-1代码片段中第一行的API“List”。Dong等人[16]首先将此API在构建的符号知识库中通过关键字的方式进行查找，以得到一系列候选的完整数据类型名，例如，“java.util.List”*、*“java.awt.List”。然后在利用List所处的代码上下文信息（例如，List存在一个泛型参数String），对候选列表进行筛选以得到List所对应的完整数据类型名（即“java.util.List”）。

然而，这些工作所依赖的符号知识库，均是通过对可编译的代码文件解析而构建的，即，符号知识库的构建存在可编译的开销。然而，这种可编译的开销，会限制符号知识库中所存储的完整数据类型名和代码上下文的数量。当匹配的完整数据类型名或上下文信息未出现在所构建的符号库中时，将出现“超出字典范围（Out-of-vocabulary）”的问题。例如对于Dong等人[16]所提出的方法而言，如果所要推理的完整数据类型名未出现在符号知识库中，返回的候选列表则为空。如果API在代码片段中所使用的上下文信息未存储在符号知识库中，则将无法从候选列表中筛选出正确的完整数据类型名。

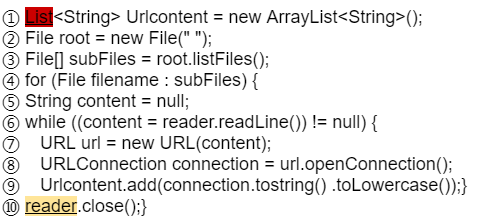


图1-1 代码片段示例

虽然，理论上可以通过从更多的软件项目中构建更全面的知识库来克服Out-of-vocabulary的问题，但可编译的开销使得该解决方案变得不现实。为此，本文创新性的使用，经提示微调的预训练掩码语言模型，作为支撑类型推理的神经知识库（类型推理模型）。基于代码语言的自然性[17，18]，无论代码是否可编译，均可作为微调语言模型的语料，故而解决了前人工作中由于可编译开销而带来的Out-of-vocabulary问题。

1.2.2 预训练语言模型适配方法研究

近年来，随着深度学习的发展，在自然语言处理领域出现了一种包含大量先验知识的预训练语言模型（如BERT[19]，RoBERTa [20]，GPT-3[21]，T5[22]）。并且，基于代码的自然性[17，18]，这种语言模型的预训练方法，也成功迁移到在软件工程领域当中，并催生了一系列包含代码知识的代码预训练语言模型（如CuBert[23]，CodeBERT[24]，CodeT5[25]，Copilot[26]）。研究[27-32]通过对预训练语言模型的深度探索，发现模型可作为包含了丰富代码语法、标识符、命名空间和自然语言命名等与代码相关的语义信息的神经知识库。然而，由于预训练任务的无关性，直接将预训练语言模型运用到下游任务并非最优的方案[33]。

为了将代码预训练语言模型当中丰富的代码语义信息运用到软件工程任务当中，现有工作[19-23]遵循“pre-train, fine-tune”范式。该范式将预训练语言模型作为特征提取器，与针对特定下游任务的神经网络层进行连接。然后，预训练语言模型和连接的神经网络，在一个特定的下游任务数据中进行训练。如图1-2所示，遵循“pre-train, fine-tune”范式，对预训练语言模型进行微调（fine-tuning）以进行分类任务。此种微调方法，可以将预训练语言模型应用到不同的下游任务当中。然而，此种方法当中，预训练语言模型的参数可以被固定，或是只对后几层的网络参数进行微调。这是一种异构的微调方法，即：语言模型的预训练任务目标和微调任务目标不一致。研究[33-36]表明，这种方法并不能充分发挥预训练语言模型的潜力，以解决下游任务。最近，研究[37]提出一种全新的范式“pre-train, prompt and predict”。该范式通过所定义的提示模板，将下游的任务转换成与预训练任务一致的形式，进而以一种同构的方法，对预训练语言模型进行微调，即：语言模型的预训练任务目标和微调任务目标一致。此种对预训练语言模型进行调优的方法被称为提示调优（Prompt tuning）。提示调优能够最大限度地将预训练语言模型当中的知识，转移到下游任务当中。如图1-2便是遵循“pre-train, prompt and predict”，通过提示调优的方法对预训练语言模型进行调优，以进行分类任务。由图1-2可知，预训练（pre-training）和微调（fine-tuning）之间是一种异构的将语言模型中的知识迁移到下游任务当中，而预训练（pre-training）和提示调优（prompt tuning）则是一种同构的方式。本文遵循“pre-train, prompt and predict”这一全新的范式，通过提示调优的方法激活预训练语言模型当中与完整数据类型名有关的知识，以解决类型推断的任务。

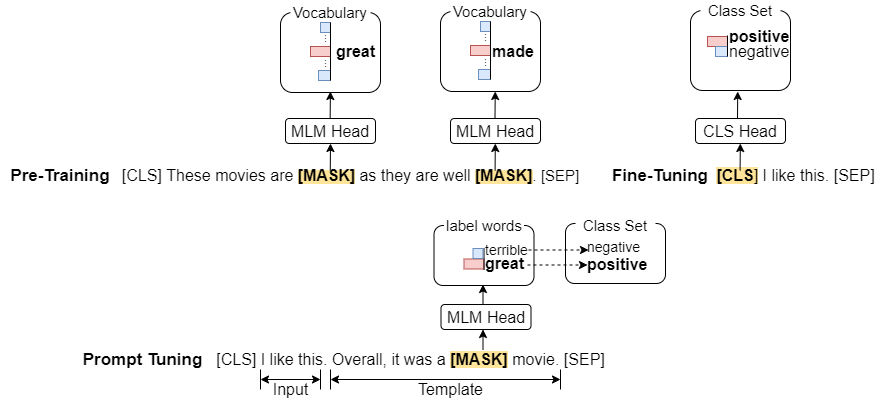


图1-2 预训练（pre-training）、微调（fine-tuning） 和提示调优（prompt tuning）之间的区别

1.2.3 提示文本的设计研究

随着对范式“pre-train，prompt and predict”研究的不断深入，研究[38-40]表明预训练语言模型当中的只是可以通过提示文本被检索出来。此时，如何设计一种高效的提示文本以检索预训练语言模型当中的知识，也逐渐成为研究的热门话题。目前，通过手工定制统一的模板，将输入转换成提示文本是一种常用的方式[30，31，32，35，36]。如图1-2所示，通过模板“[X] Overall, it was a [Z] movie.”，将分类任务转换成一种填空式的语言任务。注意，X为原始的输入文本，Z是由模型根据上下文生成的标签词。然而，预训练语言模型在下游任务中的效果对提示文本的变体非常敏感[41]。在随后的工作中提出以自动搜索的方法生成提示文本，包括离散提示[41，42]和连续提示[43-45]。但是，这些工作中需要提供统一且固定的输入。在本文中不同完整数据类型名，对应了不同的上下文信息，故而输入也非统一且固定的。为此，本文设计了一种自动的，且与上下文相关的完整数据类型提示文本（Fully-qualified name prompt）。此外，Fully-qualified name prompt（FQN prompt）当中的掩码策略也非现有工作[45，46]中采用的随机掩码策略。因为完整数据类型名是统一的整体，模型需要对这一整体进行预测，而非其中的几个单词。为此，本文采用了全跨度掩码（Full span masking）策略。该掩码策略与完整数据类型名的特性以及类型推断任务的目标相符合。本文通过设计的FQN prompt，对掩码语言模型进行提示调优，以刺激预训练的掩码语言识别FQN的形式以及使用语法，进而使其能够成为支撑类型推理的神经知识库，以进行类型推理任务。

1.3 研究内容及创新点

本文的主要研究内容是如何克服现有工作在推理任务上，由于可编译的开销所带来的Out-of-vocabulary问题。理念上，本文提出使用一种包含知识体量更大的神经知识库，替代现有工作中依赖可编译的开销而构建的符号知识库。技术上，本文使用提示调优的方法刺激预训练的掩码语言模型当中所蕴含的代码知识，进而使其能够作为支撑类型推理的神经知识库。相比现有的类型推理和提示文本设计工作，本文的创新点如下：

1. 本文创新性地将经过提示调优的预训练掩码语言模型作为类型推理的神经知识库，并通过“完型填空”式的策略进行类型推理。这与现有工作依赖解析可编译的代码文件而构建的符号知识库，通过“查字典”策略进行类型推理的方法不同。
2. 在进行提示调优的过程中，本文设计了一种自动的、且与上下文相关的完整数据类型提示文本（Fully-qualified-name prompt）以激活预训练的掩码语言模型当中与完整数据类型名相关的知识。本文中使用的Fully-qualified-name prompt（FQN prompt）与现有基于固定模板生成的提示文本不同。此外，为了预测不同长度的完整数据类型名，本文设计了一种可变长度的掩码预测方法。这与现有工作只能预测固定长度的自然语言文本不同。
3. 基于类型推理模型，本文设计了两款用于类型推理的载体插件，即：Integrated development environments（IDE）插件和WEB插件，以提高开发人员在实际开发场景复用代码片段的效率。
4. 本文提出的类型推理模型具有低资源学习的能力，即：可以通过少量数据（如现有类型推理方法所用数据的10%）得到一个性能优异的类型推理模型。此外，该类型推理模型具备泛化能力（即：类型推理模型可以泛化到不同的编程语言当中进行类型推理的任务。）及混合能力（即：可为不同的编程语言提供统一的混合类型推理模型，为类型推理任务提供一个一对多的解决方案。）。

1.4 论文章节安排

本文共有六个章节，其组织结构如下：

第1章是绪论。本章节阐述了类型推理的研究背景及意义，并从现有类型推理的方法，预训练语言模型适配方法研究和提示文本的设计三个方面对研究现状进行介绍。

第2章是研究路线。本章节对本文所要解决的问题进行详细的阐述和定义，以及介绍本文所采用的技术路线。

第3章是神经知识库激活方法研究。本章节首先是对语言模型预训练方法设计进行系统性的阐述，然后对本文所采用的提示调优策略进行阐述。通过本文所设计的提示调优策略，以激活预训练语言模型当中与代码相关的知识，从而作为支撑类型推理的神经知识库。

第4章是基于神经知识库的类型推理。本章节将阐述如何利用所构建的神经知识库来解决类型推理任务，并基于类型推理模型开发了两款类型推理的载体插件，即WEB载体插件和IDE载体插件。

第5章是实验评估。本章节从有效性，实用性以及能力边界探究三个方面对类型推理模型进行探究。有效性方面，本实验从影响类型推理模型的内部因素和外部因素进行探究。实用性方面，本实验将本文所提出的类型推理模型与现有最新的类型推理方法进行对比，并对类型推理模型的极限性能进行评估，以及对失败案例进行总结分析。能力边界探究方面，本实验主要探究了类型推理模型的泛化能力和混合能力。

第6章是结束语。总结本文的研究工作，并阐述未来对于此工作可以进行深入研究的方向。

2 研究路线

根据“pre-train, prompt and predict”这一全新范式，本文对掩码语言模型进行提示调优，以将其应用到类型推理任务当中。本节，首先对本文所要解决的问题，进行详细的阐述说明和定义。接着，对本文所采用的技术路线进行说明。

2.1 问题定义

代码片段是由几行代码语句组成的代码单元，在语法结构上是通常是不完整且不可编译的。如图2-1所示的代码片段由9行代码组成。该代码片段缺少相应的方法名或者类名。此外，该代码片段中存在两个最主要的问题，即：使用了未声明接收对象和非完全限定名称。

未声明接收对象指的是：未被声明的变量直接在代码片段中用于调用方法。例如图2-1中的“reader”未被声明，但是直接被用于调用方法名“readLine()”。

非完全限定名称指的是：代码片段中所使用的API未指定来自哪一个具体的包。这会导致API歧义的产生。例如对于List而言，它的完整数据类型名可能是“java.awt.List”，也有可能是“java.util.List”。

类型推理指的是将代码片段中的未声明接收对象和非完整数据类型名解析为对应的完整数据类型名称。这是重用或是从代码片段中挖掘信息的先决条件。例如，基于代码片段中的上下文信息，将“reader”的类型推理为“java.io.BufferedReader”，将“List”的类型推理为“java.util.List”*。*

为了对问题的边界进行清晰的定义，本文对代码片段中所要进行类型推理的三种情况进行介绍（图2-1中的高亮部分），分别为：1）变量声明处的数据类型名称（如第4行的“File”）；2）类实例化和数据创建的数据类型名称（如第6行的“URL”）；3）调用方法或访问字段对象的数据类型名称（如第5行的“reader”）。此外，本文不对以下几种情况进行推理：1）变量赋值；2）方法的参数及数组访问。因为这两种情况中相关的数据类型，可在解析代码片段中缺失的类型后，从赋值表达式或方法的返回值中推导出来。本文中将非完整限定类型名(如“File”、“URL”)和未声明的接收对象(如“reader”)称为类型推理点。

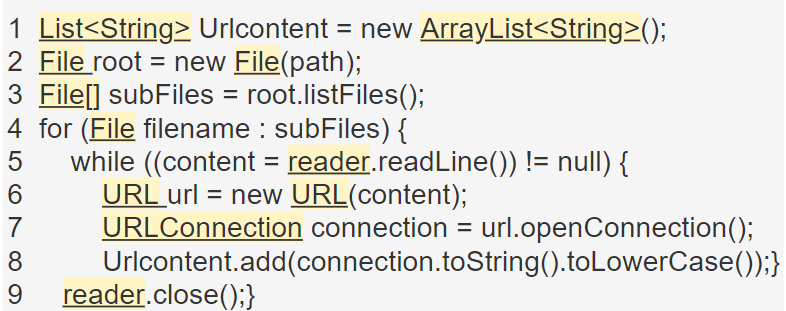


图2-1 Java代码片段

2.2 技术路线

与现有通过“查字典”策略进行类型推理的方法不同，本文创新性地将类型推理任务转换成“完型填空”式的语言任务，即：未声明接收对象（undeclared receiving object）和非完全限定名称（non-fully-qualified name）所对应的完整数据类型名称将以填空的策略得到。如图2-2所示。由于变量名的无关性（即：完整数据类型名与所声明的对象名称没有关系），未声明接收对象被替换成了七个空格，然后由经过提示调优的掩码语言模型进行填充（即：预测）。而对于非完整数据类型名称，则在前面添加三个空格，同样由模型进行填空，从而得到对应的完整数据类型名称。此处，空格指的是特殊标记[MASK]。而对于所添加的[MASK]数量，本文设计了一种启发式算法（见章节4.1）。

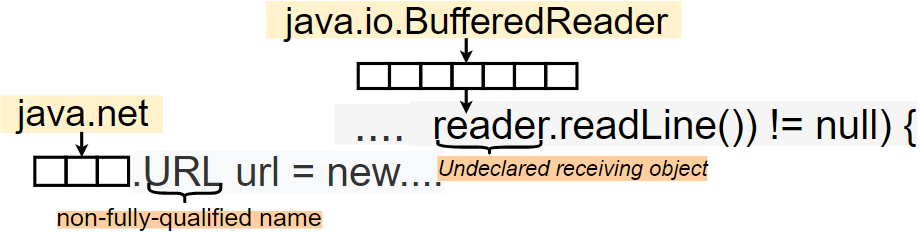


图2-2 类型推理任务转换为完形填空式的任务

为了将代码片段中的未声明接收对象和非完整数据类型名解析为对应的完整数据类型名，本文选用了经提示调优的掩码语言模型作为类型推理的神经知识库，这与现有工作依赖的符号知识库不同。首先，神经知识库的构建不依赖解析可编译的代码文件。因为基于代码自然性，代码可以被当作自然语言文本进行处理，以训练语言模型。其次，神经知识库中所包含的知识体量远大于符号知识库（例如，CodeBERT[24]拥有1.25亿个参数）。为此，为了得到能够进行类型推理的神经知识库（即：类型推理模型），本文提出了如下技术路线。方法的技术路线如图2-3所示，由三个阶段组成。

**预训练阶段**。本文选用了一个经过大型代码语料库预训练得到的掩码语言模型（CodeBERT Masked Language Model[24]），作为本文方法的基础模型。注意，基于代码语言的自然性[17，18]，语言模型的预训练过程不存在代码可编译的问题。这意味着，语言模型的预训练操作是简便的。

**提示学习阶段**。由于API的上下文信息会随着所处的代码上下文而发生改变，所以无法通过固定的模板将输入转换成提示问题。为此，本文设计了一种自动的，且与上下文相关的完整数据类型提示文本（FQN prompt），以对语言模型进行微调，进而激活掩码语言模型当中所存储的知识，用于下游的类型推断任务。而FQN prompt当中的掩码策略，为本文专门针对完整数据类型的特性以及类型推理任务的目标专门设置的FQN全跨度掩码（fully span masking）策略。

**类型推理阶段**。本文将代码片段中包含的非完整类型对象和未声明的对象视为类型推理点。在类型推理点处添加掩码标记，并将代码片段转化为带掩码的代码提示文本（Code prompt）。使用经过提示调优的代码预训练掩码语言模型作为神经知识库来预测代码提示文本（Code prompt）中的掩码标记，进而得到相应的完整数据类型名。注意，在进行类型推理的过程中，由于不同的完整数据类型名的长度是不同的，为此本文设计了一种启发式算法，以预测不同长度的完整数据类型名称。此外，以实际开发场景为切入点，本文还开发了两款类型推理模型的载体插件，即：WEB载体插件和IDE载体插件，以帮助开发人员提升对代码片段的复用效率。

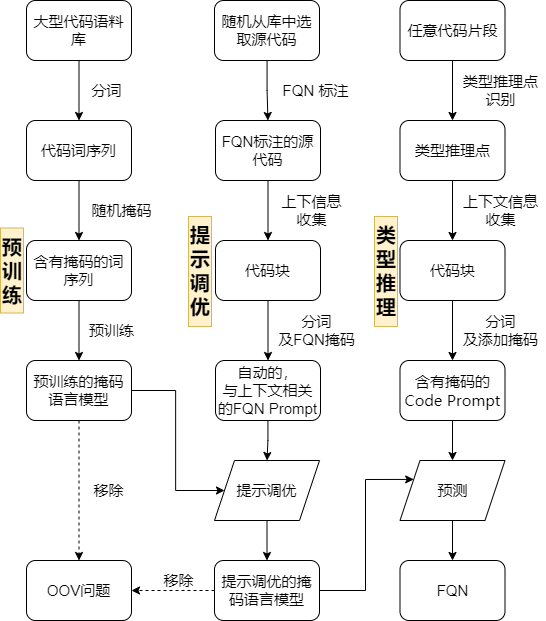


图2-3 技术路线图

2.3 本章小结

本章首先对本文所要解决的类型推理任务进行了清晰的说明和定义，其中涉及到代码片段中三种类型推理点的说明和定义。围绕所要解决的类型推理任务，本章对本文所提出的技术路线进行了阐述，其中涉及到解决类型推理任务过程中的三个阶段，包括预训练阶段、提示学习阶段和类型图例阶段。

3 神经知识库激活方法研究

本文通过提示调优的方法，激活预训练的掩码语言模型中与完整数据类型名相关的知识，以作为支撑类型推理任务的神经知识库。与符号知识库中以离散的形式存储知识的方式不同，神经知识库是将知识编码在神经网络参数的连续空间当中。本章节从语言模型的预训练方法设计，探究选择适合类型推理任务的预训练语言模型，再从提示调优的策略，研究如何激活预训练语言模型当中的知识，以作为支撑类型推理任务的神经知识库。

3.1 语言模型预训练方法设计

现有的类型推理的方法[13-16]均依赖解析可编译的代码文件构建的符号知识库。然而，这种可编译的开销会限制符号知识库中所存储的完整数据类型名和代码上下文的数量，进而导致Out-of-vocabulary的问题。为了消除这种Out-of-vocabulary问题，本文基于代码的自然性[17，18]将代码视为自然语言文本以训练模型。为此，本文使用大型的代码语料库（如CodeSearchNet[60]）通过设计一种掩码任务以训练模型。掩码任务的目标是将输入词序列中被掩码屏蔽的词还原成原来的词。掩码屏蔽是指将词序列中的词，随机替换成特殊字符[MASK]。图3-1展示了预训练过程中所需要执行的步骤。给定一段代码语料。首先对该代码语料进行分词，以得到对应的词序列。然后对该序列中词进行随机掩码的操作，该操作表示为：。在进行掩码操作时，随机将字符序列中总词数的15%随机替换成特殊字符[MASK]。注意，15%的掩码操作被Devlin等人[19]证明是有利于模型从代码语料中学习知识的比例。该操作可被形式化表达为公式3-1和3-2。最后，将得到的词序列输入到模型当中，以训练语言模型。模型的预训练任务目标，可形式化表达为公式3-3。其中，为掩码此前所对应的词。为鉴别器，用于帮助模型判定预测之后的词是否为掩码之前的词。训练目标是将损失函数3-3最小化。

为了减少预训练所带来的额外开销，在具体实现中本文采用了一个在大型代码预料上通过掩码任务训练得到的代码语言模型CodeBERT[24]的变体模型CodeBERT Masked Language Model（MLM）作为本文方法当中的基础模型。该模型使用了六种不同的编程语言所编写的代码语料进行训练。所用于模型训练的语料为CodeSearchNet[60]。在预训练阶段，该模型从训练的代码语料中捕获了丰富的代码信息，并在很多下游任务中被证明是有效的[52-54]，例如，代码搜索任务，代码翻译任务。此外，值得注意的是由于训练模型的代码语料，不存在可编译的开销。这意味着，本文中所使用的基础模型并不仅限制CodeBERT的变体模型，它可以是任何的以掩码任务训练得到的预训练语言模型（例如GraphCodeBERT[55]）。此处需要注意的是，将语言模型应用于大型代码语料库，一个技术挑战在于代码元素名称当中存在罕见的单词。当使用词分词器处理罕见单词时，同样会出现Out-of-vocabulary的问题。为了克服这个问题，本文采用子词分词器WordPiece[50]用于对代码进行分词。WordPiece根据最大似然估计将单词拆分成最小单元。例如，代码语料C中的“Arrays.asList();”被WordPiece分割成了“Ar，rays，.，as，List，();”六个部分。

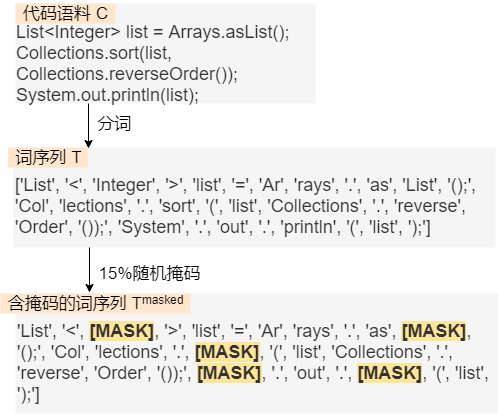


图3-1 预训练过程示例

3.2 提示调优策略研究

由于预训练的Code Masked Language Model（MLM）仅在原始的代码语料上进行预训练，它对完整数据类型名的语法和语义并没有深入的理解[45]。为此，本文提出了一种基于提示调优的方法，通过使用少量的完整数据类型名提示文本（fully-qualified name Prompt），来刺激Code MLM识别完整数据类型名的语法和使用语义。本文当中所使用的提示调优的方法，与现有工作[19，20，23，24，25]使用异构的下游任务来微调Code MLM的方法是不同的。本文当中所使用的提示调优的任务目标与Code MLM的预训练目标是完全一致的，是一种同质的调优方法。该方法能够最大限度地激活Code MLM适应下游任务的能力。并且，在进行提示调优的过程中，本文设计的Fully-qualified-name prompt（FQN prompt）是一种基于上下文，从源代码语料库中自动生成的一种提示文本。接下来，将围绕如何设计该提示文本（即：FQN prompt）以及如何使用该提示文本对预训练掩码微调进行阐述。

3.2.1 提示文本设计

给定一个包含行代码的源代码文件，形式化表示为。然后，本文根据以下四个步骤：FQN标注，上下文信息收集，分词和掩码，将代码文件，最终输出为个互不相同的FQN prompt。本文使用来表示根据第行代码生成的FQN prompt。注意，此处的代码行是根据分号（;）切割而不是换行符。这是因为，此种分割方式分解了代码元素之间的语法边界，同时它将代码处理的更加类似于模型在预训练时期的输入，并更能产生带有丰富语义知识的FQN prompt。这能够使得Code MLM在提示调优阶段，学到更加丰富的FQN语法知识和使用信息。图3-2使用了一个来自Joda Time库中的源代码，对FQN prompt的生成过程进行举例。接下来，本文将对FQN标注，上下文信息收集，分词和掩码的具体实施步骤进行详细地阐述。



图3-2 FQN prompt生成示例

1. FQN标注

给定一个来自Joda Time库的源代码文件。本文首先使用静态分析工具（如Spoon[50]），将问题定义章节（章节2.1）所定义的类型推理点解析成对应的FQN。例如图3-2中，第1行中的类型推理点“LocalTime”，被解析成对应的FQN“org.joda.time.LocalTime”。对于方法调用或字段访问的接收对象，本文将变量名替换成对应的FQN。比如图3-2中，第3行的“base.toDateTime()”被标记并替换成“org.joda.time.LocalTime.toDateTime()”。该操作抽象了具体的变量名称，但是得到了方法调用或字段方法的具体类型名称。此外，代码来源于某一个特定的库文件，但是它也可能使用其它库的API。比如第四行，包含了来自JDK库当中的“Calendar”。这些API也会被标注为对应的FNQ，即：“java.util.Carlendare”。本文将经过FQN标注的代码文件，表示为。

1. 上下文信息收集

当通过FQN标注得到之后，本文对其从上而下进行扫描，以收集包含上下文信息的代码块。表示为第行代码的所收集的代码上下文。注意，如果第行代码当中不存在完整数据类型名，则不会为该行代码收集上下文信息以得到对应的代码块。本文所收集的代码块最多由与代码行相邻的前后t行代码组成。当前扫描代码行的前后t行代码分别表示成和。此时，代码块。在图3-2当中，代码块为第3行代码所收集到的5行代码，即：第一行和第二行（），第三行（）以及第四行和第五行（）。本文将设置为2，意味着一个代码当中至多可以包含5行代码。将t设置为2，这也是受到Code MLM最大窗口单词数为512的影响。但是，有限的上下文数量可以迫使Code MLM在微调过程中，利用有限的上下文信息中进行推理，这有利于微调后的Code MLM在部分代码上进行类型推理的工作。

1. 分词

针对收集好的代码块，本文使用子词分词器WordPiece进行分词。该分词器，也被用于Code MLM预训练任务当中。本文将每一个分割的代码块所对应的词序列表示为。其中，是所对应的词序列，是所对应的词序列，是所对应的词序列，以及是序列连接符。代码块所对应分割好的词序列，如图3-2所示。

1. FQN掩码

针对分词后得到的词序列，本文将对其进行掩码操作。具体而言，本文将当中的FQN进行掩码操作，同时保持和当中原有的单词不发生改变。通过FQN掩码操作，最终可从源代码文件中获得个互不相同的FQN prompt。本文将对第代码中的FQN掩码操作得到的FQN prompt表示为：。由于每一个提示文本，都是针对特定的代码行所生成的，为此这个FQN prompt是各不相同的。此外，本文当中所设计的FQN prompt之间存在重叠的代码行，这使得模型在微调的过程中能够学习到代码行之间的长期依赖关系。

对于掩码操作是指将当中所有的FQN进行掩码屏蔽，即：FQN掩码。具体而言，本文将当中未出现在源代码文件中的单词，进行掩码操作。例如对于图3-2中，第当中的FQN“org.joda.time.DataTime”。本文将“DataTime”的包名前缀“org.joda.time”进行掩码操作。对于方法调用“base.toDateTime()”，本文将“DateTime()”前的整个类型名“org.joda.time.LocalTime”进行掩码操作。本文所采用的掩码策略与完型填空式的语言任务完美对齐，这促使产生的FQN prompt在微调的过程中能够刺激Code MLM去学习填充FQN缺失的类型信息。注意，在掩码过程中，本文不仅对单词进行掩码，而且还对连接符进行掩码（例如java当中所使用的连接符“.”）。这能够促使模型可以学习，并在类型推理的时候生成适当形式的FQN。此处，本文考虑了两种掩码策略，即：随机掩码（random masking）策略和全跨度掩码（full span mask）策略。随机掩码策略[39，40]是自然语言领域当中常用的一种掩码策略。在本文的实现中，考虑遵循CodeBERT[24]当中所使用的配置，即：将FQN当中15%的单词进行掩码。考虑到FQN当中的各个子词为一个整体，本文还设计了符合FQN特性的FQN全跨度掩码策略，即将FQN当中所有的子词进行掩码操作。研究[53]表明，全跨度掩码策略更能促使模型在微调的时候，进行知识的嵌入学习。

3.2.2 提示调优任务目标设计

本文将依次通过FQN标注、上下文信息收集、分词和掩码四个步骤，生成的个互不相同的FQN prompt用于对Code MLM进行调优。本文将微调的数据总量，形式化表示为：。其中，表示数据的总量，表示为当中被掩码掉的原始单词集合。将输入到Code MLM当中，让其预测出被掩码掉的原始单词。该学习过程可以形式化表示为公式3-7。其中为Code MLM的参数，是调优后的模型参数。

该提示调优的任务目标与Code MLM预训练时期的任务目标是一致的。这种同质的学习，能够以最大限度地挖掘Code MLM当中所存储的知识，然后将其应用到类型推理任务当中。

3.3 本章小结

本章节设计了一种掩码任务以训练语言模型。在具体实现中，本章节选用经过大型代码语料预训练得到的CodeBERT[24]变体模型—CodeBERT Masked Language Model（MLM）作为基础模型。然而，由于CodeBERT MLM预训练时期的任务无关性，本章节设计了一种提示调优策略用于激活模型当中与完整数据类型名相关的知识，使其能够作为支撑类型推理任务的神经知识库（类型推理模型）。此外，由于不同的完整数据类型名拥有不同的代码上下文信息，本章节在提示调优的过程中设计了一种自动的，且与上下文相关的FQN prompt用于对CodeBERT MLM微调。并且，FQN prompt当中使用了一种符合类型推理任务特点的全跨度掩码策略。

4 基于神经知识库的类型推理

预训练语言模型在经过提示调优之后，其识别完整数据类型名的语法和语义知识的能力被激活，进而能够作为支撑类型推理的神经知识库（即，类型推理模型）。本章节，首先从完型填空式的类型推理方法设计，探究如何将经提示调优的掩码语言模型应用于下游类型推断任务当中；再从类型推理模型载体设计，探究如何将本文提出的类型推理模型应用到实际开发场景中，以帮助开发人员进行提高复用代码片段的效率。

4.1 完型填空式的类型推理方法设计

给定一个代码片段，本文利用经过提示微调的掩码语言模型，推断出代码片段当中未声明接收对象和非完全限定名称所对应的完整数据类型名。由于代码片段是不可编译的，为此本文利用现有工作[11，12]当中所定义的启发式规则，从代码片段当中识别章节2.1所定义的类型推理点。

在进行类型推理的过程当中，本文每次只为一个类型推理点，执行类型推理的操作。假设*I*为当中第行的一个类型推理点。首先，本文为该类型推理点，执行特殊标记符的添加操作，即：在类型推理点处添加三个[MASK]标记。此处，特殊标记符[MASK]的个数为3，这是因为FQN前缀包名的最小单词数为3，其中包括一个连接符（“.”）以及两个单词（如“java”和“util”）。然而，由于类型推理点有两种不同的类型，即：未声明接收对象和非完全限定名称。本文采取了两种不同特殊标记符的添加策略。对于，未声明接收对象，本文将变量名替换为三个标记符。例如，图2-2当中的*reader.readLine()*当中的*reader*被替换成 “[MASK][MASK][MASK].readLine()”。而对于，非完全限定名称，本文则直接在其前方添加三个[MASK]特殊标记符。例如，对于图2-2中的*URL*，将被转换为“[MASK][MASK][MASK].URL”。当执行完，特殊标记符的添加操作之后，本文将使用章节3当中所描述的收集代码块的方法，为收集对应的上下文信息，记为。然后再使用字词分词器WordPiece，对进行分词操作，最终得到由词序列组成的代码提示文本（Code prompt）。最终生成的Code prompt和用于提示微调的FQN prompt在形式上是一致的。接着，由微调的掩码语言模型对Code prompt当中的特殊标记[MASK]进行预测，即：根据Code prompt当中的上下文信息对[MASK]位置进行填空。最后将预测出来的词进行拼接，以组成未声明接收对象或非完全限定名称所对应的完整数据类型名。然而，与自然语言处理领域当中包含固定长度的特殊标记[MASK]不同，不同FQN的长度是不相同的。为此，本文设计了一种长度可变的掩码策略，以自动预测不同长度的FQN。该策略的伪代码如下算法1所示。其中MLM为微调后的掩码语言模型，为输入的Code prompt，以及Step为执行的总步骤数。此外，初始化当中的*i*为[MASK]的数量，*predict*存储了当前所预测的词序列和当前词序列的平均得分。本文通过该启发式算法，实现了对不同长度的FQN进行预测。例如，对于“[MASK][MASK][MASK].readLine()”而言，通过算法1，最终增加至七个[MASK]，即：“[MASK][MASK][MASK][MASK] [MASK][MASK][MASK].readLine()”，并经过提示调优的掩码语言模型预测为“java.io.BuffereReader”作为“readLine()”的完整数据类型名。

|  |  |
| --- | --- |
| **算法1：自动预测不同长度的FQN。** | |
| **Input:** |  |
| **Initialization:** |  |
| 1. **While** *True* | |
|  | |
|  | |
|  | |
|  | |
|  | |
|  | |
|  | |
|  | |
|  | |
|  | |

4.2 类型推理模型载体设计

本文将经过提示调优的预训练掩码语言模型，作为支撑类型推理的神经知识库，即：类型推理模型。该类型推理模型可以很方便的扩展到各种需要准确FQN信息的应用场景当中。例如，可以集成到程序分析工具当中，以辅助工具在获取API完整数据类型的基础上对代码片段进行程序分析。

此处，本文将提出的类型推理模型集成到WEB和Integrated Development Environment（IDE）当中，进而帮助开发人员在实际开发过程当中提高对代码片段的使用效率。在实际开发场景中，开发人员经常会在各种在线资源网站上阅读代码片段，或者将其复制到IDE当中，进行二次开发利用。但是，由于代码片段当中的未声明接受对象和非完全限定名称，会阻碍开发人员对于代码片段的使用。例如，当在在线资源网站（如Stack Overflow）上阅读代码片段时，代码片段中类型信息的缺失可能是使得开发人员不能够正确的了解某个API的使用功能。或者，当开发人员将在线资源上的代码片段拷贝到IDE环境当中时，类型信息的缺失会让编译器抛出编译错误“symbol cannot be resolved”。为此，为了解决开发人员在使用代码片段时，因类型缺失而带来的各种问题，本文设计了两款类型推理模型的载体插件，即：WEB载体插件和IDE载体插件。类型推理模型的载体插件工作流程图，如图4-1所示。类型推理模型载体插件主要包含两个功能，即类型推理功能和包自动导入功能。类型推理功能是为代码片段中某个API推理出正确的完整数据类型名而设计的，包自动导入功能是补充代码片段中缺少的包导入语句。注意，包自动导入的功能在在类型推理功能的基础之上设计的。两款类型推理载体插件的具体工作，见章节4.2.1和4.2.2。

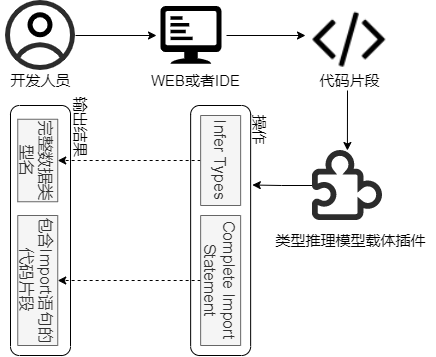


图4-1 类型模型载体插件工作流程图

4.2.1 WEB载体插件设计

对于WEB载体的类型推理插件，它不仅可以实时为代码片段中的未声明接收对象和非完整数据类型名推理出对应的完整数据类型名称（即类型推理功能），还能够在完成类型推理之后将推理出的完整数据类型名称与其对应的API官方文档链接起来，以帮助开发人员理解API的具体实现功能或者使用注意事项。如图4-2所示。图中展示了一个来自Stack Overflow帖子当中的代码片段[[1]](#footnote-1)。其中包含了未声明接收对象（如*file*）和非完全限定名称（如*FileWriter*）。此时，开发人员可以通过将鼠标悬停在代码片段中特定API上方的方法将其选定。然后，被选定的API所对应的完整数据类型名称，将会被WEB载体的类型推理插件展示出来。例如，当开发人员选定API“*FileWriter*”。此API所对应的完整数据类型“*java.io.FileWriter*”会被WEB载体的类型推理插件展示出来，如图4-1 (a)所示。此外，被选定的API所对应的官方开发文档也会被WEB载体的类型推理插件展示出来，如图4-1(b)所示。

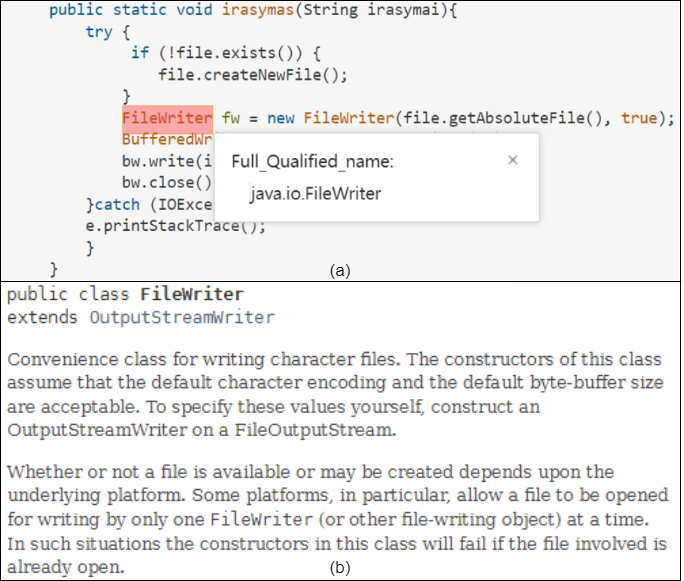


图4-2 WEB载体的类型推理插件演示示例

4.2.2 IDE 载体插件设计

IDE载体的类型推理插件可用于帮助开发人员补全代码片段当中所缺失的导入语句（import statement），既包自动导入功能。例如，将来自Stack Overflow帖子**错误!未定义书签。**当中的代码片段复制粘贴到IDE中时，由于类型信息缺失，编译器将会抛出编译错误“symbol cannot be resolved”。如图4-3(a)当中标红的API所示。此时，开发人员可以手动选择一个代码行区域，然后启动IDE载体的类型推理插件。当类型推理插件被启动后，点击“Complete Import Statement”按钮。IDE载体的类型推理插件将会从所选择的代码行区域当中，自动选择未声明接收对象和非完全限定名称，并推断出对应的完整数据类型名。然后将推理的完整数据类型名转换成导入语句，并补全在程序的起始位置。例如，IDE载体的类型推理插件为“FileWriter”导入了“import java.io.FileWriter”，如图4-3.(b)所示。在处理未声明对象时，IDE载体的类型推理插件会为其补全对应的声明语句。具体而言，当IDE载体的类型推理插件为未声明对象推理出对应的完整数据类型名之后，会根据推断得到的完整数据类型名生成对应的声明语句，并补全对应的导入语句。例如，IDE载体的类型推理插件为“file”变量补全了声明语句“File file = new File(“”)”，并导入了语句“import java.io.File”，如图4-3.(b)所示。除此之外，IDE载体的类型推理插件也可以选定特定的API，然后点击“Infer Types”按钮，以获得对应的完整数据类型名。

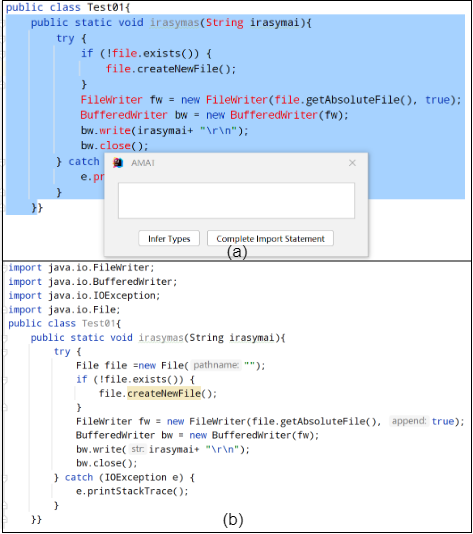


图4-3 IDE载体的类型推理插件演示示例

4.3 本章小结

基于类型推理模型，本章节设计了一种“填空式”的策略来完成类型推理任务。然而，不同的完整数据类型名的长度是不同的。为此，本章节设计了一种启发式算法，以通过“填空式”的策略预测不同长度的完整数据类型名。此外，本章节还设计了两款类型推理模型载体插件（分别是WEB载体插件和IDE载体插件），用于帮助开发人员在实际开发过程中推理出代码片段中未声明接受对象和非完整数据类型名的完整数据类型名称，进而提高复用代码片段的效率。

5 实验及结果分析

本文提出使用经过提示调优的预训练掩码语言模型，作为类型推理的神经知识库（即，类型推理模型）。本章节，从以下三个方面对该类型推理模型进行系统性的评估：

1. 类型推理模型的有效性评估。本实验分别从影响神经知识库有效性的内部因素（即：微调数据的输入策略，微调数据的大小以及FQN的掩码策略）和外部因素（即：提示文本相似度和API基数）进行探究。
2. 类型推理模型的实用性评估。本实验将类型推理模型与现有最新的类型方法进行比较，并且评估神经知识库在未见过的FQN和代码上下文中的性能，并对类型推理神经知识库典型的失败案例进行总结。
3. 类型推理模型的能力边界探究。主要是对类型推理的神经知识库的泛化能力和混合能力进行探究。

5.1 实验环境及参数

本实验是用CUDA在NVIDIA GeForce RTX 3090上进行的。操作系统是Ubuntu 20.04.3 LTS。Python版本为3.7。此外，在进行提示调优时，本实验所使用的模型超参数如表5-1所示。

表5-1 模型提示调优时的超参数设置

|  |  |  |  |
| --- | --- | --- | --- |
| 超参数名称 | 值 | 超参数名称 | 值 |
| Optimizer | Adam W | Warm up steps | 10% |
| Learning rate | 5e-5 | Batch size | 64 |
| LR scheduler | Linear | Adam epsilon | 1e-8 |
| Gradient norm | 1.0 | Epoch | 10 |

5.2 类型推理模型的有效性评估

本文提出了一种基于提示微调的掩码语言模型，对代码片段当中的未声明接收对象和非完全限定名进行类型推理的方法。本实验将从有效性对类型推理模型进行评估。

5.2.1 有效性评估动机

本文将经过提示调优的掩码语言模型作为类型推理的神经知识库（即：类型推理模型），以支撑填空式的类型推理任务。而对于类型推理模型的有效性，会受到三个内在因素的影响：

1. 数据输入策略的影响（即One-by-one输入和Mix-all输入）；
2. 提示调优时，所用的微调数据大小的影响；
3. FQN掩码策略的影响（即FQN随机掩码策略和FQN全跨度掩码策略）。

除了以上三个内部因素，会对类型推理模型的性能产生影响，还有两个外部因素，也会影响模型的类型推理性能：

1. 提示调优时所用的FQN prompt和推理时所用的Code prompt之间的文本相似度；
2. API基数。API基数指的是一个API所对应的FQN数量。一个API可以对应多个不同的FQN，而在特定的代码上下文中只有特定的一个FQN满足要求。

研究这些内部因素和外部因素如何影响本文所提出的类型推理模型的性能，是非常重要的。

5.2.2 实验数据与评价指标

1. 实验数据

为了获得可靠的实验结果，本实验需要大量带有真实FQN的代码片段。为此，本实验使用了Java当中的六个库的源代码，它们包括Android、GWT、Hibernate、Joda Time、Xstream和JDK。这六个库的源代码，也被使用到前人的类型推断工作当中[14-16]。本实验使用这些库的源代码，来验证类型推理模型的有效性。具体而言，本实验从Github当中来下载这些库的源代码。据统计，一共下载了39255个代码源文件，其中包含了3190个Android源代码文件，7096个GWT源代码文件，10670个Hibernate源代码文件，17404个JDK源代码文件，330个Joda Time源代码文件，以及39255个Xstream源代码文件。由于此实验评估的重点是掩码语言模型在提示调优时的学习能力和类型推理能力，所以本实验将这些文件当中所有的方法和类名都进行了收集。

本实验将收集到的源代码文件进行切分。具体而言，本实验首先从每个库的源代码随机抽取40%作为提示调优时的学习数据集，剩下的60%则作为类型推断时的测试数据集。然后，进一步将学习数据集随机等分为四份。请注意，每份都同时涉及了来自六个库的数据。本实验使用x-pl（x=1、2、3、4）来表示提示调优时所使用的数据份量。接着，使用章节3中描述的FQN标注的方法，将这些源代码中的API解析为对应的完整数据类型名。这些完整数据类型名作为提示调优时的标答。经过FQN标注之后，从这些源代码中一共获得了361048个FQN prompt。其中涉及了133739个不同的完整数据类型名。对于类型推理阶段，本实验将测试数据集中的源代码转换成Code prompt，以模拟输入的代码片段。本实验将测试数据集中的导入语句删除了，以确保非完整数据类型名的存在。此外，本实验还随机删除测试集中的0到5条变量声明语句，以确保未声明接收对象的存在。此时，一共获得了541020个Code prompt可作为本实验的实验数据，其中涉及了183298个需要进行类型推理的完整数据类型名。这183298个完整数据类型名当中，分别存在71.34%，57.18%，46.82% and 38.78%的完整数据类型名没有出现在1-*pl*，2-*pl，*3-*pl，*4-*pl*当中。请注意，调优时所用的FQN prompt和推理时所用的Code prompt互不相同。表5-2展示了各个库中FQN prompt和Code prompt中的具体数量。

表5-2 FQN prompt和Code prompt的数量

|  |  |  |
| --- | --- | --- |
| Libraries | Number of FQN prompt | Number of Code Prompt |
| JDK | 107394 | 159857 |
| Android | 97368 | 142783 |
| GWT | 47254 | 75357 |
| Hibernate | 48405 | 73675 |
| Joda Time | 23190 | 37150 |
| Xstream | 37437 | 52198 |
| Total | 361048 | 541020 |

1. 评价指标

本实验使用准确率（Accuracy）和BLEU[51]评分来衡量本文所提出的类型推理模型的性能。注意，本实验未使用现有工作[11-13]中使用的召回率（recall）和F1-score作为衡量指标，是因为现有工作中存在因Out-of-vocabulary而遗漏推理的情况，而本文的类型推理方法不存此情况。此时，Accuracy、召回率（recall）和F1-score三者的值是相等的。为此，本实验只选用了Accuracy作为衡量指标。Accuracy是计算推断总数当中，推理正确的完整数据类型名的百分比。计算公式如5-1所示。其中，“inference requested”为代码片段中的类型推理点的总数，“inference corrected”为推理正确的完整数据类型名的总数。

对于BLEU评分是用于机器翻译任务中的评价指标，用于计算生成的文本（candidate）和参考文本之间（reference）的相似度。其计算公式如5-2所示。其中，“candidate”是为模型生成的文本（此处为提示微调的掩码语言模型所生成的完整数据类型名）。对于分子中的第一个求和符为candidate的总数量，第二个求和符是统计一条candidate中所有的，而表示某一个n-gram在reference（此处为完整数据类型名的标答）中的个数。所以，整个分子的含义是给定的candidate中有多少个n-gram词语出现在reference当中。而对于分母中的表示在candidate中的个数，即：分母的含义是获得的candidate中n-gram的个数。此处，本实验使用此处，本实验使用BLEU-2评分来评估预测的完整数据类型名和标答的完整数据类型名之间的相似性，即：公式5-2当中的n表示为2。其BLEU的分值范围为0到1，分值越高表示类型推理模型所预测的完整数据类型名和标答中的完整数据类型名越相似。本实验在字词的层面上计算BLEU-2分数，有助于在更细粒度的级别上理解类型推理模型在完整数据类型明上的质量。

5.2.3 内部因素对类型推理模型的影响

在此实验当中，对影响类型推理模型的三个内部因素进行评估，即：提示调优时所用的数据输入策略，提示调优的数据大小，以及FQN的掩码策略。

1. 微调数据输入策略的影响

在本实验的有效性评估当中，使用了六个Java库的源代码文件。然而，如何将这个六个库的源代码文件输入到模型当中，以进行微调是需要探究的。在此实验当中，本实验使用*1-pl*大小的数据，来进行数据输入策略的探究。*1-pl*当中包含六个库的源代码，每个库占10%。此处，本实验考虑了两种数据输入策略：One-by-one和Mix-all。对于One-by-one，是将六个库中的数据逐个输入到掩码语言模型当中进行微调。这意味着，当使用第一个库的源代码对Code MLM微调之后，得到微调的掩码语言模型，然后再使用第二个库的源代码对继续微调，得到微调的掩码语言模型，依次类推。最后通过这种方法得到了6个微调的掩码语言模型。最后的掩码语言模型是依次通过“JDK”、“Android”、“GWT”、“Hibernate”、“Joda Time”、“Xstream”微调得到的。此处，将“JDK”作为第一个微调的源代码库，是因为它是一个基本库。该库中的API可以在其它五个库中使用。与One-by-one相反，Mix-all则是将所有六个库的源代码混合在一起，然后用于微调掩码语言模型，最后获得一个微调后的掩码语言模型。所有微调得到的模型，均在同一份测试集上进行测试，并计算准确率（Accuracy）。

**实验结果和实验分析。**实验结果如表5-3所示。首先对于的平均准确率在0.43和0.54之间波动，并且准确率并没有随着库的源代码增加而呈现线性增长。此外，与的准确率相比，的准确率提高了32%，尽管这两个模型都使用了六个库的源代码进行微调。这表明，对于微调掩码语言模型而言，相比One-by-one的数据输入策略，Mix-all的数据输入策略是一种更有效的数据输入策略。具体而言，只在当前所使用的微调的库上有较高的准确率，而在其它五个库上的准确率较低。例如，是依次通过“JDK”，“Android”和“GWT”微调得到的。而只在“GWT”上具有较高的准确率（即0.852），而对于“JDK”和“Android”均具有较低的准确率（即0.51和0.507）。这表明掩码语言模型对只对当前使用的调优数据集更加敏感，并且容易遗忘之前已经使用调优数据集的数据特点。例如，在“JDK”上的准确率为0.809，而只有0.452。此外，在六个库上的准确性接近甚至高于在六个库上的准确性。例如，在“Hibernate”上的准确率为0.785，而在上的准确率增加至0.866，增加了8.1%。这意味着，Mix-all的数据输入策略比One-by-one策略更能促使掩码语言模型有效地适应下游类型推断的任务。

表 5-3 不同数据输入策略下的模型准确率

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Libraries |  |  |  |  |  |  |  |
| JDK | 0.809 | 0.684 | 0.51 | 0.421 | 0.378 | 0.452 | 0.787 |
| Android | 0.334 | 0.835 | 0.507 | 0.32 | 0.282 | 0.322 | 0.822 |
| GWT | 0.342 | 0.517 | 0.852 | 0.32 | 0.308 | 0.377 | 0.866 |
| Hibernate | 0.246 | 0.293 | 0.262 | 0.785 | 0.604 | 0.408 | 0.866 |
| Joda Time | 0.381 | 0.367 | 0.264 | 0.269 | 0.88 | 0.732 | 0.872 |
| Xstream | 0.497 | 0.493 | 0.438 | 0.43 | 0.431 | 0.864 | 0.863 |
| Average | 0.435 | 0.532 | 0.472 | 0.424 | 0.481 | 0.526 | 0.846 |

1. 微调数据的大小

此实验当中使用了四种不同大小的数据集（即1-pl，2-pl，3-pl，4-pl）来探究微调数据大小对类型推理模型性能的影响。注意，此实验将六个库当中所有的数据混合在一起来微调掩码语言模型（即Mix-all策略），同时使用FQN全跨度掩码策略来对掩码语言进行微调。此处，将通过不同比例微调得到的掩码语言模型记为，x表示微调的数据大小，F表示使用的为全跨度掩码策略。另外，为了凸显提示调优的有效性，本实验将未经微调的掩码语言模型（MLM）直接在应用在测试数据集上进行类型推理任务，作为对照组。注意，所有的类型推理模型均在同一份数据集上进行测试，并计算准确率（Accuracy）和BLEU-2以评估不同类型推理模型的性能。

**实验结果和实验分析。**实验结果如表5-4所示。未经微调的掩码语言模型MLM的平均准确率和BLEU-2分别为0.307和0.478。这表明，即使是未经微调的掩码语言模型MLM当中也存储了一定的完整数据类型名信息。这些信息是模型在预训练阶段，从代码语料库中学习到的。然而，这些信息并不足矣为API推断出准确的完整数据类型名。通过对实验结果更深入的分析，发现未经微调的掩码语言模型对于较短且常用的FQN有较高的准确率。例如，MLM能够将“File”正确预测为“java.io.File”。而对于相对复杂的完整数据类型名却不能预测正确。例如，MLM为*objectAnimator*预测了错误的完整数据类型名“.animationation.Animator.ObjectAnimator”而不是对应正确的完整数据类型名“android.animation.Animator.ObjectAnimator”。

当使用10%（即1-pl）的库的源代码对掩码语言模型进行提示微调时（即），类型推理的准确率从0.31骤升到了0.85，增加了0.54。这表明，基于提示的微调能够有效的刺激掩码语言模型学习完整数据类型名的语法和使用知识，即使是只使用了10%的源代码。例如，能够准确为“objectAnimator”预测出正确的完整数据类型名“android.animation.Animator.ObjectAnimator”。此外，BLEU-2的平均得分也从0.48上升到了0.93。这表明，预测的完整数据类型名即使与标答中的完整数据类型名不同，但是也非常地相似。此时，开发人员可以通过额外的搜索工具（例如，谷歌搜索），从几乎正确的预测中，找到正确的完整数据类型名。例如，为“clipRect()”预测了完整数据类型名“android.raphics.Canvas.clipRect()”。相比正确的完整数据类型名“android.graphics.Canvas.clipRect()”只是缺少了字符“g”。此时在谷歌中搜索“android.raphics.Canvas.clipRect()”，排在第一位的就是正确的完整数据类型名。此外，71.34%的完整数据类型名没有出现在1-pl提示学习数据中。这意味着，经过提示调优后的类型推理模型展现出了卓越的低资源场景下的推理能力。

随着微调数据的逐渐增加，平均准确率和BLEU-2的分数也逐渐增加。但是相比未经微调的MLM到的性能骤升，到的性能逐渐平缓。最终，模型的性能稳定在，获得了0.888的平均准确率和0.95的BLEU-2得分。并且，的性能相比只有微乎其微的增长。这表明，对于升级原始的掩码语言模型MLM，是否进行提示学习比提示学习时所使用的数据大小更为重要。

与微调后的模型在其它四个库上的性能相比，在JDK和Android上的准确率和BLUE-2的分数相对较低。这是因为JDK和Android是基础的软件工具开发包，它当中包含了很多不一样的包，拥有很多不一样的变体完整数据类型名。特别是对于JDK当中所包含的完整数据类型名，其变化很大。比如，*java.xml*，*org.w3c.dom*，*org.omg.CORBA*，*org.xml.sax*。这增加了模型推断JDK中不同形式完整数据类型名的困难，特别是当提示学习的数据很少的时候（例如，）。

表5-4 模型经不同数据大小调优后的性能表现

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Libraries | MLM | |  | |  | |  | |  | |
| Acc | BL-2 | Acc | BL-2 | Acc | BL-2 | Acc | BL-2 | Acc | BL-2 |
| JDK | 0.341 | 0.505 | 0.787 | 0.906 | 0.808 | 0.909 | 0.843 | 0.923 | 0.858 | 0.926 |
| Android | 0.355 | 0.523 | 0.822 | 0.892 | 0.829 | 0.897 | 0.849 | 0.902 | 0.859 | 0.911 |
| GWT | 0.254 | 0.403 | 0.866 | 0.952 | 0.87 | 0.96 | 0.902 | 0.965 | 0.915 | 0.97 |
| Hibernate | 0.215 | 0.341 | 0.866 | 0.956 | 0.885 | 0.965 | 0.918 | 0.975 | 0.923 | 0.977 |
| Joda | 0.402 | 0.633 | 0.872 | 0.96 | 0.894 | 0.965 | 0.911 | 0.968 | 0.911 | 0.97 |
| Xstream | 0.274 | 0.465 | 0.863 | 0.939 | 0.839 | 0.939 | 0.904 | 0.958 | 0.9 | 0.954 |
| Average | 0.307 | 0.478 | 0.846 | 0.934 | 0.854 | 0.939 | 0.888 | 0.949 | 0.894 | 0.951 |

1. 掩码策略的选择

此实验比较了两种不同的掩码策略对微调模型性能的影响，即FQN随机掩码策略和FQN全跨度掩码策略。FQN随机掩码策略指的是将完整数据类型名当中的词随机掩码掉15%，而FQN全跨度掩码策略是指将完整数据类型名当中API的类型名称全部进行掩码。两种掩码策略的示例图，如图5-1所示。对于图5-1中的词序列，FQN全跨度掩码之后的得到词序列，如图5-1（a）所示。FQN随机掩码策略得到的词序列，如图5-1（b）所示。此处，本实验使用4份大小微调数据集（即4-pl）以及FQN随机掩码策略对掩码语言模型Code MLM进行微调。微调后的模型记为。然后在测试集上对模型进行测试，并计算准确率（Accuracy）和BLEU-2。

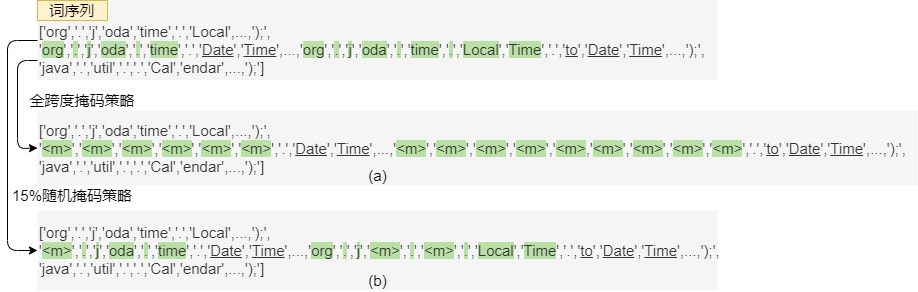


图5-1全跨度掩码和随机掩码的差别

**实验结果和实验分析。**在测试集上的实验结果如表5-5所示。与的性能骤升不同，的准确率比未经微调的掩码语言模型只提升了0.05，BLEU-2只提升了0.01。此外，仅使用10%数据微调得到的模型的性能远高于经过40%微调得到的模型。这表明，提示调优时需要一个合适的掩码策略，才能使微调后的模型有效。而由于常用的随机掩码策略，并不适合完整数据类型名的数据特征和类型推理任务，为此，在测试集上的性能较差。相比之下，本文专门为推理完整数据类型名而设计的全跨度掩码策略所微调得到的类型推理模型的性能要优异很多。

表5-5 模型经不同数据掩码策略调优后的性能表现

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Libraries | MLM | |  | |  | |
| Accuracy | BLEU-2 | Accuracy | BLEU-2 | Accuracy | BLEU-2 |
| JDK | 0.341 | 0.505 | 0.787 | 0.906 | 0.387 | 0.502 |
| Android | 0.355 | 0.523 | 0.822 | 0.892 | 0.415 | 0.527 |
| GWT | 0.254 | 0.403 | 0.866 | 0.952 | 0.292 | 0.386 |
| Hibernate | 0.215 | 0.341 | 0.866 | 0.956 | 0.232 | 0.342 |
| Joda | 0.402 | 0.633 | 0.872 | 0.96 | 0.413 | 0.607 |
| Xstream | 0.274 | 0.465 | 0.863 | 0.939 | 0.43 | 0.555 |
| Average | 0.307 | 0.478 | 0.846 | 0.934 | 0.362 | 0.486 |

5.2.4 外部因素对类型推理模型的影响

在探究外部因素对总体方法影响的实验当中，主要对提示文本间的相似度和API基数进行了探究。在本实验当中使用了微调的掩码语言模型，然后在测试集上进行测试。此处之所以选用类型推理模型是因为实验5.2.3表明类型推理模型的性能稳定于。注意，使用的是30%微调数据集以及FQN全跨度掩码策略，提示调优得到的类型推理模型。

1. 提示文本的相似度

在计算提示文本间的相似度时，FQN prompt（使用在提示微调的过程中）和Code prompt（使用在类型推理的过程中）都是词序列，并且两个提示文本中的特殊标记[MASK]都被移除了。此实验通过比较FQN prompt和Code prompt之间的单词重叠数，来衡量二者之间的提示文本相似度。计算公式如5-3所示。其中，指的FQN prompt当中的词序列，指的是Code prompt当中的词序列。

公式5-3用于计算FQN prompt当中有多少词出现在Code prompt当中。通过计算单词重叠数来计算提示文本间的相似度，是因为代码当中的词序列对于模型进行类型推理是很有意义的。给定一个来自测试集的，计算其于调优集中的FQN prompt的相似度。本实验根据提示文本相似度的范围来划分，并计算这些范围当中的类型推断准确率和BLEU-2得分。

**实验结果和实验分析。**实验结果如表5-6所示。由表可知，FQN prompt和Code prompt之间的相似度主要集在0.15和0.45，占了72.5%的Code prompt。此外，有4.6%的Code prompt与FQN prompt的相似度在0到0.15之间，还有23.1%的Code prompt与FQN prompt的相似度在0.45之上。当Code prompt和FQN prompt之间的相似度高于0.35，模型在类型推理任务上的性能趋于稳定。然而，当相似度再往上逐渐增加时，类型推理模型的性能提升不大。例如，当相似度在0.35到0.45之间时，模型的准确率为0.919。而当相似度在0.65到0.88之间时，模型的准确率也只有0.924，仅提升了0.005。即使当Code prompt和FQN prompt之间的相似度较低时，微调后的模型也能够在类型推理任务上表现良好。例如，当Code prompt和FQN prompt之间的相似度低于0.15的时候，模型的准确率和BLEU-2分布为0.76和0.90。当提示文本的相似度大于0.15的时候，模型的BLEU-2的得分在0.94周围波动。

表5-6 不同的提示文本相似度范围内的模型性能

|  |  |  |  |
| --- | --- | --- | --- |
| Similarity Range (%) | Code prompt (%) | Accuracy | BLEU-2 |
| (0, 15] | 4.6 | 0.755 | 0.899 |
| (15, 25] | 21.8 | 0.842 | 0.94 |
| (25, 35] | 25.1 | 0.891 | 0.957 |
| (35, 45] | 25.6 | 0.919 | 0.961 |
| (45, 55] | 14.6 | 0.92 | 0.945 |
| (55, 65] | 6.3 | 0.918 | 0.937 |
| (65, 88] | 2.2 | 0.924 | 0.956 |

1. API的基数

一个API可以拥有多个不同的完整数据类型名。而在特定的代码上下文当中，只有特定且唯一的完整数据类型名，才能符合当前的代码语义。此时，类型模型必须在类型推理阶段，从这多个不同的完整数据类型名当中推理出正确的完整数据类型名。此实验根据一个API对应的完整数据类型名数量进行范围的划分，并在各个范围中计算类型推理时的准确率和BLEU-2分数。

**实验结果和实验分析。**实验结果如表5-7所示。由表可知，推理数据集当中只有34.7%的API具有唯一对应的完整数据类型名（即Cardinality=1）。微调的掩码语言模型并不能在这类数据上取得100%的准确率，这是因为有时模型会产生一些不正确的单词。例如对于API“BitString”，对应的唯一完整数据类型名为“org.apache.harmony.security.asn1.BitString”。微调的掩码语言模型预测了一个近乎一样的完整数据类型名“org.apache.harm..security-.asn1.BitString”。相比正确的完整数据类型名，只有一个单词被模型预测错误，即将正确完整数据类型名当中的“harmony”预测成为“.harm”。

对于基数在1到100的区间里，微调后的掩码语言模型的准确率和BLEU-2的分数在0.89到0.92，0.95到0.97之间波动，表现出非常稳定的性能。这种小范围的波动意味着API的基数对模型类型推理的影响较小。此外，对于拥有极高基数的API（即Cardinality>500），模型的类型推理性能，有所下降，但是仍在可接收的范围内（即准确率在0.82到0.84之间，BLEU-2在0.87到0.90之间）。通过对实验结果数据的进一步分析，发现许多拥有极高基数的API是来自JDK当中的，例如，*tostring()*，*equals()*，或是*get()*方法。这类API这可以出现在任何的类当中，这使得仅基于本地有限的上下文来确定它们来自哪个特定的类是具有挑战性的。

表5-7 不同API基数下的模型性能

|  |  |  |  |
| --- | --- | --- | --- |
| Cardinality | Code prompt (%) | Accuracy | BLEU-2 |
| 1 | 34.7 | 0.925 | 0.972 |
| 2 | 10.6 | 0.91 | 0.961 |
| 3 | 4.6 | 0.903 | 0.957 |
| 4 | 3.9 | 0.9 | 0.946 |
| 5 | 10.6 | 0.897 | 0.955 |
| (5, 10] | 12.1 | 0.893 | 0.95 |
| (10, 20] | 5.4 | 0.915 | 0.936 |
| (20, 50] | 6.5 | 0.901 | 0.91 |
| (50, 100] | 2.9 | 0.892 | 0.959 |
| (100, 500] | 6.4 | 0.858 | 0.931 |
| (500, 1000] | 1.7 | 0.842 | 0.902 |
| (1000, +∞) | 1.7 | 0.826 | 0.874 |

5.2.5 实验小结

在探究内部因素对类型推理模型影响的实验中，发现Mix-all数据输入策略能够迫使微调的掩码语言模型以一种更有效的方式适应到下游类型推理的任务当中。提示调优时，使用FQN全跨度掩码策略能够产生一个具有较高的推理性能的类型推理模型。并且，该模型与转换后的“完型填空”式的类型推理任务完美结合。相比未经微调的掩码语言模型，性能也有大幅度的提升。微调数据的总量对掩码语言模型的性能，并没有显著的影响。但是如果掩码策略不符合FQN的数据特征和任务特征，则微调后的模型性能会遭遇大幅度下降。

在探究外部因素对类型推理模型影响的实验中，发现提示调优的掩码语言模型在任何范围内的提示文本相似度中，都表型出稳定且准确的性能。即使当提示文本的相似度非常低（<0.15）的时候，或者当API基数非常高的时候（>100），微调的掩码语言模型在类型推理时也表现出很惊人的性能。这表明，掩码语言模型在微调时并不是简单的将微调数据当中FQN进行记忆然后匹配，而是从提示调优的过程中，获得良好的推理能力，以处理各种变体的代码上下文，API歧义等问题。

5.3 类型推理模型的实用性评估

基于微调的掩码语言模型，采用“完型填空式”的策略，以推断代码片段当中未声明的接收对象和非完整数据类型所对应的完整数据类型名。此处，将从实用性的角度对类型推理模型的性能进行评估。

5.3.1 实用性评估动机

本实验将从以下三个方面，对类型推理模型的实用性进行评估。

1. 与现有基于符号知识库进行类型推理的方法[15，16]相比，本文所提出的类型推理模型在真实代码片段上的性能如何？
2. 在自然语言处理领域当中[21，34]，经过提示调优的掩码语言模型在未知的新任务上展现出惊人的推理能力。本实验对类型推理模型，是否能够对提示过程中未见过的FQN和代码上下文进行类型推理展开探究。
3. 本实验对类型推理模型的结果进一步的分析和探究，以总结类型推理模型的失败模式以及可能的原因，进而对本文所提出的类型推理方法进行改进。

5.3.2 实验数据与评价指标

此实验从常用的问答网站Stack Overflow当中收集两类包含代码片段的数据集。第一类是已经被用于之前类型推理工作[14-16]实验当中的数据集—StatType-SO[14]。该数据集包含从Stack Overflow当中收集的268个代码片段，其中所使用的API涉及到Android、GWT、Hibernate、Joda Time、Xstream、JDK六个库。这些代码片段所包含的代码行的范围在6到223行，平均为28行。此外，这268个代码片段当中涉及未声明接收对象和非完整数据类型名的类型推理点个数分别为2248个和1838个，并包含1454个需要推断的唯一完整数据类型名。第二类数据集是本实验通过手工的方式，从Stack Overflow当中收集的代码片段—Short-SO。该数据集当中所使用的API同样也涉及Android、GWT、Hibernate，Joda Time，Xstream，JDK六个库。但是，与StatType-SO当中的代码片段有充足的上下文信息不同，Short-SO当中的代码片段所包含的代码行不会超过3行。如图5-2所示。图5-2（a）是来自StatType-SO中的代码示例，包含8行代码。相比Short-SO中的3行代码（图5-2（b））。StatType-SO中的代码片段为推理API“RootLayoutPanel”的FQN提供了充足的上下文信息。

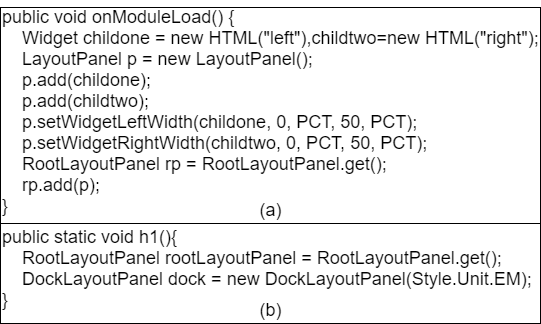


图5-2：StatType-SO (a) 和Short-SO (b)代码示例

Short-SO当中包含120个代码片段，每个库包含20个代码片段。Short-SO当中包含256个未声明接收对象的类型推理点和269个非完整数据类型名的类型推理点，其中一共涉及到286个唯一的类型推理点。在进行实验之前，本实验通过手工的方式将Short-SO当中的未声明接收对象和非完整数据类型名解析为对应的完整数据类型名，作为真实需要推断出的完整数据类型名。具体而言，本实验邀请两个计算机专业的研究生，试图通过帖子中的上下文信息，以手工的方式修复代码片段当中的未声明接收对象和非完整数据类型名。一个学生主要负责进行修复，另外一个学生主要负责对修复后的代码进行检查验证。当遭遇分歧的时候，他们通过讨论的方式以解决这些分歧。至于评价指标，本实验仅仅使用准确率来衡量类型推理模型的性能。因为相比BLEU-2分数，准确率是一种更为严苛的衡量指标。此外，本实验使用30%调优数据集微调得到的掩码语言模型来进行测试。这是因为实验5.2.3的结果结果表明，微调的模型性能稳定在。

5.3.3 最新类型推理方法对比

此实验将类型推理模型与现有最新的类型推理工具进行对比。具体而言，本实验中将类型推理模型在StatType-SO和Short-SO进行测试，并与现有最先进的Java类型推理工具（即COSTER[15]和SnR[16]）进行对比。

**实验结果和实验分析。**实验结果如表5-8所示。对于COSTER和SnR在数据集StatType-SO上的性能，此实验采用报告在工作[16]当中的准确率。首先，类型推理模型在StatType-SO上所取得的平均准确率，高于COSTER和SnR，即0.91比0.71和0.875。具体而言，在GWT，JDK和Joda Time上所取得的准确率均高于COSTER和SnR。对于在Android上，和SnR的准确率相接近（即，0.91比0.94），但是比COSTER高很多（即，0.91比9.43）。此外，在Hibernate上所取得的准确率仅为0.76，比COSTER和SnR都低。然而，许多错误是由真实的完整数据类型名使用了旧的包名，而预测的FQN使用新的包名造成的。例如，真实的完整数据类型名“javax.persistence.Entity”使用旧包名“javax.persistence”，而将其预测成使用新包名“jakarta.persistence”的完整数据类型名“jakarta.persistence.Entity”。对于Xstream，取得和COSTER一样的准确率，但比SnR低。然而，大部分错误是源于将包“com.cloudbees.api.config”中的完整数据类型名预测错误。而意外的是，此包中任何类型的完整数据类型名都没有出现在的提示调优的数据当中。这中零资源的推理能力，对于类型推理模型是具有挑战的。

在StatType-SO中的JDK部分代码中的准确率为0.99，在Short-SO中的JDK部分代码中的准确率同样为0.97。这远远高于在JDK源代码上的准确率（即0.84）。这是因为在使用JDK源码对模型进行有效性评估的过程当中，涉及到各种变体形式的完整数据类型名。相比之下，来自Stack Overflow的代码片段当中，JDK当中的FQN大多数是来自于常见的包中的API，例如，*java.util*，*java.io*。这使得降低了的预测难度。

在Short-SO上取得了较高的平均准确率，即0.89。尽管Short-SO比StatType-SO中的代码要短很多。但是，在两个数据集上的平均准确率，并没有太大的差异。即在StatType-SO上的平均准确率为0.91，在Short-SO上的平均准确率为0.89。特别是对于Hibernate，JDK，Joda Time和Xstream这四个库，在StatType-SO和Short-SO上的准确率非常接近。在Android和GWT的准确率不如StatType-SO。主要原因是Short-SO当中存在高基数的API（例如，getString()，add()）。事实上，在进行预测时，仅依赖于有限的局部上下文（即5行代码）来进行类型推断。为此，代码片段中代码行的多少，对于模型进行类型推理并不重要。此外，本文所提出的基于神经知识库的类型推理，比现有方法基于符号知识库的启发式匹配更强大以及更灵活。Saifullah等人[15]也宣称“在仅有1到2行的代码片段当中，COSTER会因为缺少足够的上下文信息，而无法进行类型推断”。

表5-8 在StatType-SO和Short-SO当中的性能

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Libraries | StatType-SO | | | Short-SO |
| COSTER | SnR |  |  |
| Android | 0.433 | 0.936 | 0.906 | 0.819 |
| GWT | 0.908 | 0.758 | 0.952 | 0.882 |
| Hibernate | 0.904 | 0.948 | 0.759 | 0.765 |
| JDK | 0.562 | 0.711 | 0.989 | 0.973 |
| Joda Time | 0.571 | 0.895 | 0.975 | 0.971 |
| Xstream | 0.884 | 100 | 0.88 | 0.913 |
| Average | 0.71 | 0.875 | 0.91 | 0.887 |

5.3.4 极限能力评估

此实验主要探究，对于未出现在微调数据中的完整数据类型名和代码上下文，类型模型是否能够同样准确的推理出未声明接收对象和非完整数据类型的FQN。此处，先为FQN是否出现给出定义：如果一个在StatType-SO或Short-SO当中的API所对应的FQN，出现在的微调数据集当中，则为出现过，记为Seen-FQN。反之，则为未出现过，记为Unseen-FQN。据统计，两个Stack Overflow当中共有3024个已经出现的FQN（即Seen-FQN）和1371个未出现的FQN（即Unseen-FQN）需要进行类型推理。然后对于代码上下文（Context）是否出现给出定义：如果StatType-SO或Short-SO当中所用进行类型推理的Code prompt和微调数据集当中的FQN prompt之间最大的提示文本相似度大于0.35，则为出现过，记为Seen-Context。反之，则为未出现过，记为Unseen-Context。此处，提示文本相似度采用实验“外部因素对类型推理模型的影响”当中所使用的计算方法。阈值选为0.35，是基于对表5-7的分析而设置的。据统计，两个Stack Overflow当中共有2526个已经出现的Code prompt和1985个未出现的Code prompt。此实验将Seen/Unseen-FQN和Seen/Unseen-Context进行四种组合，并计算在两个Stack Overflow数据集上的每种组合推理的平均准确率。

**实验结果和实验分析。**这四种组合的准确率被展示在表5-9当中。由表可知，无论是Seen-Context还是Unseen-Context，在都取得了较高的准确率，即在Seen-FQN-Seen-Context场景下的平均准确率为0.983，在Seen-FQN-Unseen-Context场景的平均准确率为0.976。具体来说，当要被预测的完整数据类型名出现在的微调数据集当中时，在Unseen-Context上的准确率仅略低于Seen-Context上的准确率。这意味着，如果在提示学习的过程中FQN被模型见过，则神经知识库中相似的代码上下文可以被查找出来以进行类型推断。与现有符号知识库会直接出现Out-of-vocabulary相比，微调的掩码语言模型可以更好的处理这种代码上下文的变体。例如图5-3当中的（b），是一个为推理“BufferedImage”所收集的代码块（code block）。而与之最为相似的是在提示过程中使用的代码块（a）。尽管如此，代码块（b）仍旧是Unseen-Context。此时，微调的掩码语言模型在基于代码块（b）为“BufferedImage”进行类型推理时，会将两个代码块中最为相似且重要的单词进行比对，以辅助推理出正确的完整数据类型名“java.awt.image.BufferedImage”。请注意，模型在进行类型推断时，完全没有进行任何的代码分析操作（例如，数据流的获取），而是纯粹基于代码上下文的知识嵌入。

当需要预测的FQN为未出现在的微调数据中（即Unseen-FQN）时，仍然在Unseen-FQN-Seen-Context场景中取得0.75的平均准确率，在Unseen-FQN-Unseen-Context场景中取得0.67的平均准确率。所展现出来的这种零资源推理能力，对于现有基于符号知识库采用字典式匹配的方法是完全不可能的，因为无论对于Unseen-FQN还是Unseen-Context都会在查找时直接出现Out-of-vocabulary的问题。对于Unseen-FQN和Unseen-Context的这种预测能力，来源于掩码语言模型在预训练时期和随后的提示微调中捕获的代码自然性。通过代码自然性，开发人员在统一命名空间中以有意义的名称命名代码元素。例如，图5-3的（c）和（d）展现一个Unseen-FQN-Unseen-Context场景中的例子。需要推断的FQN“javax.swing.JTable.DefaultTableModel”未出现在类型推理模型的调优数据集当中。但是与之在单词组成成份上相似的FQN “javax.swing.JTable.DefaultTableCellRender”出现在的调优数据集当中。尽管用于类型推理的代码块（d）与提示微调时的代码块（c）不相似，但是类型推理模型依旧可以从中捕获到“JTable”和“DefaultTable”之间细微的关系。然后，为“DefaultTableModel”预测对应的FQN“javax.swing.table.DefaultTableModel”。这种预测，不仅依赖于模型在提示微调时从可见的代码序列中学习到的单词之间的联系，还依赖于模型在预训练时期从大型语料库中学习的与“JTable”和“DefaultTable”相关且更广泛的知识。

表5-9: 模型在Seen或者Unseen当中的准确率

|  |  |  |  |
| --- | --- | --- | --- |
| FQN  Context | Seen | Unseen | Overall |
| Seen | 0.983 | 0.753 | 0.922 |
| Unseen | 0.976 | 0.671 | 0.874 |
| Overall | 0.980 | 0.712 | 0.902 |

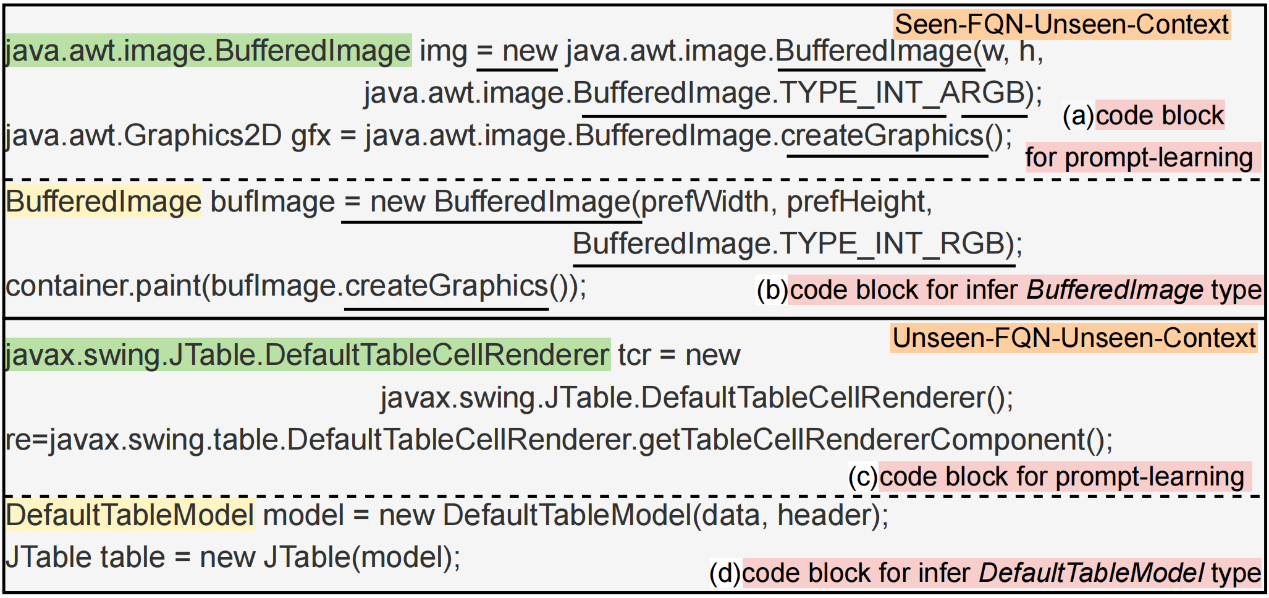


图5-3 Seen-FQN-Unseen 和Unseen-FQN-Unseen-Context例子

5.3.5 失败案例分析

此实验对的预测结果中，常见的预测错误案例进行分析，并统计对每个包（package）中预测错误的FQN数量（）。根据预测错误的FQN数量对每个包进行排序。此实验只对每个库中，含有预测错误数量最多的包进行分析（即Top-1）。此实验还为预测错误的FQN收集其对应正确的FQN。对于多个相同的正确FQN，本实验只进行一次计数。并对当前库中的错误总数进行统计（。

**实验结果和实验分析。**实验结果如表5-10所示。表中展示包含FQN预测错误最多的包名。例如，对于Joda Time。包含预测错误最多的包是“java.sql”，其中包含一个需要被正确预测的FQN（#DistGTFQN）“java.sql.Date”，已经被反复预测错误6次（），且6次均被错误的预测为“java.util.Date”。此外，这6次错误占Joda Time当中，总错误数量的46.2%（即=13）。总的来说，一个包中含有的预测错误的真实FQN的数量远远小于包中总的错误数，这意味着对于相同的FQN，模型可能会重复的多次预测错误。通过对错误案例的分析，本实验确定了三个导致预测错误的主要原因。

首先，这些错误是由于需要推理的FQN和所预测的FQN之间的使用上下文非常相似所引起的，并且这两个FQN在单词组成上也是相似的。例如，将Joda Time当中的“Date”反复的错误预测为“java.util.Date”，但是其正确的FQN为“java.sql.Date”。事实上，“java.sql.Date”是“java.util.Date”的子类，并且两个FQN有相似的使用上下文。通过进一步分析发现，如果“Date”的使用上下文当中包含SQL相关的信息（例如“SQLException”），就能将*Date*正确的预测为“java.sql.Date”。否则，在常见的使用上下文当中，模型更加倾向于将其预测为“java.util.Date”。

其次，需要被预测的FQN所处的包，没有出现在模型的提示微调的数据集当中。也就是说，这个包中的任何FQN，都没有被用于微调模型。此种现象被称之为包级别的零资源，这对模型来说是一项非常具有挑战性的任务。比如，对于Xstream，包“com.cloudbees.api.config”中的任何一个FQN都没有被用于微调掩码语言模型。此时，此包中的两个FQN被错误预测17次，占Xstream当中总错误数量的21%。所有的这17次错误，均来源于包的零资源。与之类似，GWT中的包“com.extjs.gxt.ui.client.widge”中的三个FQN，也被错误预测11次，占据GWT当中总错误数量的22%。这些错误可以通过使用这些包中的一部分FQN对模型进行微调来减少。

最后，模型推理出的FQN实际上是正确的，但由于它们与正确的FQN使用了不同版本的包名，被判定为错误推理。这种情况被称为假阴性（false-negative）。例如，在Hibernate当中，111个错误来源于包“javax.persistence”中的17个需要推理的FQN，这111个错误占Hibernate当中总错误数量的54%，占六个库中总错误数量的24%。此处，代码片段中的真实FQN是根据其对应的代码上下文而获取的。但是，代码片段中的一些FQN是可以被更新版本的FQN所替代的。例如，为Hibernate所选择的一个代码片段[[2]](#footnote-2)，如图5-4所示。将图中的“ManyToOne”，“Column”等API的类型设定为“javax.persistence”包中的FQN。但是，将这些API的类型预测为“jakarta.persistence”包中的FQN。事实上，它们是不同版本的包中，但功能相同的FQN，即“javax.persistence”在JPA 3.0中被重新命名为“jakarta.persistence”。由于微调的代码数据是JPA 3.0中的FQN，为此预测了新版本包中的FQN。

表5-10 经常预测错误的FQN

|  |  |  |  |
| --- | --- | --- | --- |
| Library | Top-1 Package Name (PN) | #DistGTFQNs  () | The Ground Truth FQNs to be Inferred |
| Joda Time | java.sql | 1 (6/13) | PN.Date |
| JDK | Javax.sql | 1 (3/9) | PN.SwingWorker |
| Android | android.os | 1 (26/91) | PN.Bundle |
| Xstream | com.cloudbees.api.config | 2 (17/81) | PN.Environment, etc. |
| GWT | com.extjs.gxt.ui.client.widget | 3 (11/50) | PN.VerticalPanel, etc. |
| Hibernate | javax.persistence | 17 (111/216) | PN.Id, etc. |



图 5-4 Stack Overflow中的代码片段

5.3.6 实验小结

通过本实验研究发现，在来自Stack Overflow的代码片段上进行测试，本文所提出的类型推理模型取得比现有最先进的类型推理工具更高的准确率。并且，在FQN-Unseen和Context-Unseen的情况下，本文所提出的类型推理模型也取得了较为优异的结果。虽然，类型推理模型可能会在零资源的场景中，有较低的性能。此外，在本实验中有24%的错误是来源于假阴性，因为所推理的FQN和真实的FQN所使用的包来自不同版本的库，但实际功能一致。

5.4 类型推理模型的能力边界探究

本文基于代码的自然性[17，18]，将代码视为自然语言文本让掩码语言模型在提示微调的过程中学习FQN的语法和语义知识，进而使微调的掩码语言模型能够适应下游的类型推理任务。此处，将从泛化能力和混合能力两个方面，对本文提出的类型模型进行探究。

5.4.1 能力边界探究动机

本实验将从泛化能力和混合能力，对本文所提出的类型推理模型的能力边界进行探究。

1. 首先是泛化能力。在实际开发场景中，除Java代码片段包含未声明接收对象和非完整数据类型之外，其它的静态编程语言也有同样的问题。此时，研究本文所提出的类型推理模型是否能够扩展到其它静态类型的编程语言当中是非常值得的。此实验，以C#为例，以探究和评估类型推理模型的泛化能力。
2. 其次是方案的混合能力。目前，本文所提出的类型推理模型只被应用在特定的编程语言当中，以进行类型推理的任务。此时，是否可以获得一个包含不同编程语言的完整数据类型名知识的混合类型推理模型，进而为类型推理任务提供一个一对多的解决方案？此实验以C#和Java为例，以探究和评估整体方案的混合能力。

5.4.2 实验数据与评价指标

与Java含有源代码库不同，C#没有相应的源代码库。为此，本实验从微软的官方文档中下载.NET框架的源代码[[3]](#footnote-3)作为本实验的C#实验数据集。该数据集包含8871个源代码文件。类似地，此实验将其分为两部分：40%作为C#的微调数据集，剩余的60%作为测试数据集。为了从40%的代码文件中获得用于对掩码语言模型进行提示调优的FQN prompt。本实验使用章节3.2所描述的方法。但是在FQN标注阶段使用C#的静态分析工具Roslyn[[4]](#footnote-4)。最后，共获得182430个FQN prompt，涉及到26893个唯一的FQN。对于60%的测试数据，本实验将所有命名空间的声明语句进行移除（即using 语句），然后使用章节4.1所提到的方法转换成对应的Code prompt。最后，共获得268139个Code prompt，其中涉及到42467个FQN需要进行推理。此外，为了对比本文所提出类型推理方案在Java和C#是否会存在差异，本实验增加一组对照实验。为了最大限度的保证实验结果的可靠性，此实验从40%的Java微调数据集中，随机选取182430个FQN prompt用于重新微调掩码语言模型。此外，还从60%的Java测试数据集中选取268139个Code prompt用于对经微调后的掩码语言模型进行测试。注意，随机选取的提示文本对六个库的源代码均有涉及。评价指标方面，此实验依旧选用准确率和BLEU-2分数。

5.4.3 泛化能力探究

此实验将类型推理模型迁移到不同的编程语言当中，以探究其泛化能力。此处，以C#为例。具体而言，本实验使用40%的C#微调数据集对预训练的掩码语言模型进行微调，以获得一个可用于C#当中的类型推理模型。然后，将模型在60%的C#测试数据集上进行测试。此外，从40%的Java微调数据集中筛选出来的FQN prompt也被用于微调掩码语言模型，以获得可用于在Java当中的类型推理模型。然后，将模型在筛选出来的Code prompt当中进行测试。通过计算准确率和BLEU-2来衡量模型的性能。

**实验结果和实验分析。**实验结果如表5-11所示。由表可知，和的准确率分别为0.842和0.856，BLEU-2的评分分别为0.924和0.026。和在各自的测试集上的准确率仅相差0.014，BLEU-2评分也仅相差0.002。这表明，本文的类型推理方案可以应用到其它不同的静态语言当中。这种泛化性源于本文的方法以文本的形式对代码进行操作，忽略了不同编程语言之间的语法规则或命名方法的差异性。此外，不同编程语言（如Java，C#）的代码语料也在预训练时期对掩码语言模型进行训练，这使得预训练的掩码语言模型当中包含不同编程语言的先验知识，进而可以在提示微调阶段更好的泛化到不同的编程语言当中。同时，不同的静态编程语言当中的FQN形式是一致的，例如Java当中的*java.util.List*和C#当中的*System.Collections.Generic.List*。都是a.b.c的形式。这使得本文提出的类型推理方案，很容易泛化到不同的编程语言当中，以进行类型推断的任务。

表5-11 在不同编程语言上的性能表现

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Programming Languages |  | |  | |  | |
| Accuracy | BLEU-2 | Accuracy | BLEU-2 | Accuracy | BLEU-2 |
| C# | 0.842 | 0.924 | - | - | 0.833 | 0.922 |
| Java | - | - | 0.856 | 0.926 | 0.850 | 0.924 |

5.4.4 混合能力探究

此实验将类型推理模型同时应用到不同的编程语言当中进行类型推理任务，以探究其混合能力。与特定于一种编程语言的类型推理模型不同，混合类型推理模型当中包含多种不同编程语言完整数据类型知识。该理模型可以被应用到不同的编程语言当中进行类型推理，从而为类型推理任务提供一个一对多的解决方案。具体而言，此实验将40%的C#微调数据集和所筛选出来的Java FQN prompt进行混合，以组成一份混合的微调数据集。该份混合的微调数据集被用于对掩码语言模型进行微调，从而获得一个混合类型推理模型。最后将所得到的混合推理模型分别在C#和Java的测试集上进行测试，并计算准确率和BLEU-2分数。

**实验结果和实验分析。**的实验结果如表5-10所示。由表可知，在C#和Java当中的准确率分别为0.833和0.850，BLEU-2分别为0.922和0.944。这表明混合类型推理模型当中所存储的多种不同编程语言中的完整数据类型名知识，可以被解码应用于具体的编程语言当中进行类型推理的任务。此外，在C#和Java上的精度，与特定的类型推理模型（即和）在C#和Java上的精度非常接近，仅仅降低了0.006和0.009。这表明不同的编程语言之间的语法差异，并不会对微调掩码语言模型的过程产生消极影响。并且微调后的掩码语言模型可以根据所预测API的上下文信息，正确的预测出属于当前编程语言中的完整数据类型名。例如，图5-5展示Java和C#中的预测示例。类型推理模型，根据（a）中的上下文信息（例如，public class 等关键字），正确的推理出exists()的完整数据类型名，即“java.util.List.exists()”。根据（c）中的上下信息（例如，namespace等关键字），正确的推理出Exists()的完数据类型名，即“System.IO.File.Exists()”。此外，模型的混合类型推理能力也源于本文将代码视作文本进行处理，这屏蔽了不同编程语言之间的语法差异，让模型更注重从字符级别去理解不同编程语言之间的差异，而不会对特定的编程语言种的语法知识特别敏感。

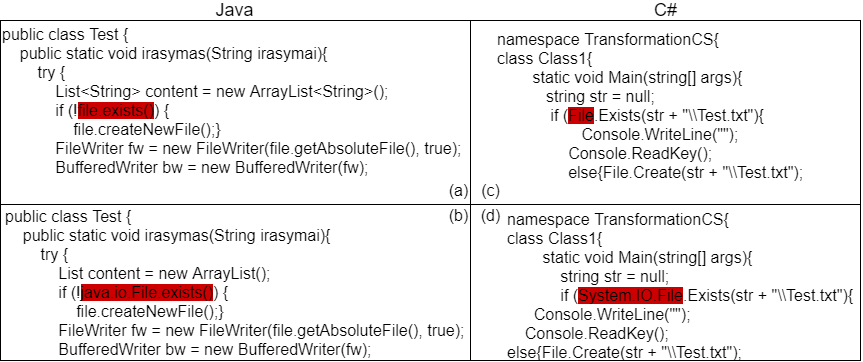


图5-5 Java和C#预测示例

5.4.5 实验小结

从本章的实验结果可知，本文所提出的类型推理方案，可以很容易的扩展到不同的编程语言当中以进行类型推断的任务。这源于本文将代码视为文本进行处理，忽略了不同编程语言之间的语法差异。同时，模型在预训练时期从不同编程语言的语料库中所捕获的语义信息，也增强了模型可以泛化到不同编程语言中的能力。此外，本文所提出的方案还可以用于获取一个混合类型推理模型，从而为不同的编程语言提供一个一对多的类型推理解决方案。

5.5 本章小结

本章节从有效性、实用性和能力边界三个方面对类型推理模型进行了系统性的评估。有效性实验结果表明，本文类型推理模型的设计是有效的，包括语言模型预训练方法设计，提示调优策略设计和完型填空式的类型推理方法设计。同时，类型推理模型也具备低资源学习能力。通过少量代码对模型调优（总数据量的10%），便能取得优异的类型推理性能。实用性实验结果表明，本文的类型推理模型的性能优于现有最新的类型推理工具（COSTER[15]和SnR[16]）。同时，类型推理模型能够处理现有工作中存在的Out-of-vocabulary问题。也就是，当API的完整数据类型名未出现在模型的调优数据当中时，类型推理模型依旧能够为此API推理出正确的完整数据类型名。能力边界实验结果表明，类型推理模型具备泛化能力（可在不同的编程语言中进行类型推理）及混合能力（可为不同的编程语言提供一个统一的混合类型推理模型）。

6 总结与展望

6.1 总结

本文基于“pre-train，prompt and predict”这一全新范式，通过提示调优的方法对掩码语言模型进行微调，以得到一个可以支撑类型推理的神经知识库（类型推理模型）。与现有工作是基于解析可编译的代码构建符号知识库不同，本文所提出的方法利用了代码语言的自然性，将代码当作文本进行处理，这消除了前人工作中所存在的可编译开销。此外，本文将类型推理任务转换成“完型填空”式的语言任务，这完美的将掩码语言模型的预训练任务目标，提示调优的任务目标和类型推理的任务目标，完美的对齐。在进行提示调优时，本文设计了一种自动的，且与上下文相关的FQN prompt，以及使用了FQN 全跨度掩码策略来微调掩码语言模型。由于不同FQN的长度是不同的，本文在类型推理时设计了一种可变长度的掩码策略，以推理不同长度的FQN。

在实验部分，本文从有效性，实用性和能力探究三个方面，对本文所提出的类型推理模型进行全方位的实验和探究。有效性的实验结果表明，本文对于提示文本以及掩码策略的设计都是有效的。当仅使用10%的代码数据对掩码语言模型进行提示调优之后，模型便能在类型推理任务当中取得非常卓越的性能。在实用性的实验中，本文所提出的类型推理模的性能高于现有的类型推理方案。并且在各种变体的上下文和现有工作中无法处理的Unseen-FQN，本文所提出的类型推理模型均取得了不错的性能表现。这展现出，类型推理模型在零资源场景中也具备良好的类型推理能力。在能力边界探究实验当中，本文的类型推理模型很容易的扩展到不同的编程语言当中，以进行类型推理工作。并且，可以为不同编程语言提供统一的混合类型推理模型。

6.2 展望

本文未来的工作主要围绕以下几点展开：

1. 通过选用参数体量更大的预训练语言模型，进一步提升本文所提出类型推理方案的性能。目前，本文所使用的是CodeBERT仅在掩码任务上训练得到的掩码语言模型，拥有1.25亿个参数。这比最新的预训练语言模型小很多，例如，GPT3[21]和它的变体模型Copilot[52]拥有1750亿个参数。可以通过选用更大的预训练语言模型，进一步提升本文在类型推理任务上的性能。
2. 通过强化学习的方法，进一步规范模型所生成的FQN格式。目前，在使用微调的掩码语言模型进行类型推理的过程中，模型会生成一些错误形式的FQN，例如，一个FQN中生成了连续的两个点（即*com..works.xstream*）。此时，可以编写一些正确的FQN语法规则，为强化学习中的奖励机制，从而使得模型可以在强化学习的指导下减少生成错误格式的FQN。
3. 将本文所提出的类型推理模型，部署到更多需要完整数据类型的下游应用当中。目前，本文以提升开发人员复用代码片段的效率为目标，开发了两款用于类型推理的插件。事实上，本文所提出的类型推理模型，可以应用在不同的任务场景中。例如，API搜索，代码搜索。解析代码片段中API的完整数据类型，可以有效地提高目前以关键字匹配的API（或代码）搜索方法。

参考文献

1. Dagenais B, Robillard M P. Creating and evolving developer documentation: understanding the decisions of open source contributors[C]//Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. 2010: 127-136.
2. Dagenais B, Robillard M P. Recovering traceability links between an API and its learning resources[C]//2012 34th international conference on software engineering (icse). IEEE, 2012: 47-57.
3. Singer J. Practices of software maintenance[C]//Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272). IEEE, 1998: 139-145.
4. Gupta P, Mehrotra N, Purandare R. Jcoffee: Using compiler feedback to make partial code snippets compilable[C]//2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2020: 810-813.
5. Thummalapenta S, Xie T. Parseweb: a programmer assistant for reusing open source code on the web[C]//Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. 2007: 204-213.
6. Dagenais B, Hendren L. Enabling static analysis for partial java programs[C]//Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications. 2008: 313-328.
7. Maji S, Rout S S, Choudhary S. DCoM: A Deep Column Mapper for Semantic Data Type Detection[J]. arXiv preprint arXiv:2106.12871, 2021.
8. Zhang T, Upadhyaya G, Reinhardt A, et al. Are code examples on an online q&a forum reliable?: a study of api misuse on stack overflow[C]//2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE, 2018: 886-896.
9. Piccolboni L, Di Guglielmo G, Carloni L P, et al. Crylogger: Detecting crypto misuses dynamically[C]//2021 IEEE Symposium on Security and Privacy (SP). IEEE, 2021: 1972-1989.
10. Zhou Y, Liu S, Siow J, et al. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks[J]. Advances in neural information processing systems, 2019, 32.
11. Ren X, Ye X, Xing Z, et al. API-misuse detection driven by fine-grained API-constraint knowledge graph[C]//2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2020: 461-472.
12. Melo L T C, Ribeiro R G, Guimaraes B C F, et al. Type Inference for C: Applications to the Static Analysis of Incomplete Programs[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 2020, 42(3): 1-71.
13. Subramanian S, Inozemtseva L, Holmes R. Live API documentation[C]//Proceedings of the 36th international conference on software engineering. 2014: 643-652.
14. Phan H, Nguyen H A, Tran N M, et al. Statistical learning of api fully qualified names in code snippets of online forums[C]//2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE, 2018: 632-642.
15. Dong Y, Gu T, Tian Y, et al. SnR: constraint-based type inference for incomplete Java code snippets[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 1982-1993.
16. Saifullah C M K, Asaduzzaman M, Roy C K. Learning from examples to find fully qualified names of api elements in code snippets[C]//2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019: 243-254.
17. Hindle A, Barr E T, Gabel M, et al. On the naturalness of software[J]. Communications of the ACM, 2016, 59(5): 122-131.
18. Allamanis M, Barr E T, Devanbu P, et al. A survey of machine learning for big code and naturalness[J]. ACM Computing Surveys (CSUR), 2018, 51(4): 1-37.
19. Devlin J, Chang M W, Lee K, et al. Bert: Pre-training of deep bidirectional transformers for language understanding[J]. arXiv preprint arXiv:1810.04805, 2018.
20. Liu Y, Ott M, Goyal N, et al. Roberta: A robustly optimized bert pretraining approach[J]. arXiv preprint arXiv:1907.11692, 2019.
21. Brown T, Mann B, Ryder N, et al. Language models are few-shot learners[J]. Advances in neural information processing systems, 2020, 33: 1877-1901.
22. Raffel C, Shazeer N, Roberts A, et al. Exploring the limits of transfer learning with a unified text-to-text transformer[J]. J. Mach. Learn. Res., 2020, 21(140): 1-67.
23. Kanade A, Maniatis P, Balakrishnan G, et al. Learning and evaluating contextual embedding of source code[C]//International Conference on Machine Learning. PMLR, 2020: 5110-5121.
24. Feng Z, Guo D, Tang D, et al. Codebert: A pre-trained model for programming and natural languages[J]. arXiv preprint arXiv:2002.08155, 2020.
25. Wang Y, Wang W, Joty S, et al. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation[J]. arXiv preprint arXiv:2109.00859, 2021.
26. Pearce H, Ahmad B, Tan B, et al. An Empirical Cybersecurity Evaluation of GitHub Copilot's Code Contributions[J]. arXiv e-prints, 2021: arXiv: 2108.09293.
27. Troshin S, Chirkova N. Probing Pretrained Models of Source Code[J]. arXiv preprint arXiv:2202.08975, 2022.
28. Wan Y, Zhao W, Zhang H, et al. What Do They Capture?--A Structural Analysis of Pre-Trained Language Models for Source Code[J]. arXiv preprint arXiv:2202.06840, 2022.
29. Buratti L, Pujar S, Bornea M, et al. Exploring software naturalness through neural language models[J]. arXiv preprint arXiv:2006.12641, 2020.
30. Heinzerling B, Inui K. Language models as knowledge bases: On entity representations, storage capacity, and paraphrased queries[J]. arXiv preprint arXiv:2008.09036, 2020.
31. Jiang Z, Xu F F, Araki J, et al. How can we know what language models know?[J]. Transactions of the Association for Computational Linguistics, 2020, 8: 423-438.
32. Roberts A, Raffel C, Shazeer N. How much knowledge can you pack into the parameters of a language model?[J]. arXiv preprint arXiv:2002.08910, 2020.
33. Ding N, Chen Y, Han X, et al. Prompt-learning for fine-grained entity typing[J]. arXiv preprint arXiv:2108.10604, 2021.
34. Han X, Zhao W, Ding N, et al. Ptr: Prompt tuning with rules for text classification[J]. AI Open, 2022, 3: 182-192.
35. Sun Y, Zheng Y, Hao C, et al. NSP-BERT: A Prompt-based Zero-Shot Learner Through an Original Pre-training Task--Next Sentence Prediction[J]. arXiv preprint arXiv:2109.03564, 2021.
36. Wang C, Yang Y, Gao C, et al. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence[C]//Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2022: 382-394.
37. Liu P, Yuan W, Fu J, et al. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing[J]. arXiv preprint arXiv:2107.13586, 2021.
38. Schick T, Schütze H. Exploiting cloze questions for few shot text classification and natural language inference[J]. arXiv preprint arXiv:2001.07676, 2020.
39. Raffel C, Shazeer N, Roberts A, et al. Exploring the limits of transfer learning with a unified text-to-text transformer[J]. J. Mach. Learn. Res., 2020, 21(140): 1-67.
40. Schick T, Schütze H. It's not just size that matters: Small language models are also few-shot learners[J]. arXiv preprint arXiv:2009.07118, 2020.
41. Gu Y, Han X, Liu Z, et al. Ppt: Pre-trained prompt tuning for few-shot learning[J]. arXiv preprint arXiv:2109.04332, 2021.
42. Liu X, Ji K, Fu Y, et al. P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks[J]. arXiv preprint arXiv:2110.07602, 2021.
43. Shin T, Razeghi Y, Logan IV R L, et al. Autoprompt: Eliciting knowledge from language models with automatically generated prompts[J]. arXiv preprint arXiv:2010.15980, 2020.
44. Gao T, Fisch A, Chen D. Making pre-trained language models better few-shot learners[J]. arXiv preprint arXiv:2012.15723, 2020.
45. Lester, Brian, Rami Al-Rfou, and Noah Constant. "The power of scale for parameter-efficient prompt tuning." *arXiv preprint arXiv:2104.08691* (2021).
46. Li, Xiang Lisa, and Percy Liang. "Prefix-tuning: Optimizing continuous prompts for generation." *arXiv preprint arXiv:2101.00190* (2021).
47. Tang, Tianyi, Junyi Li, and Wayne Xin Zhao. "Context-Tuning: Learning Contextualized Prompts for Natural Language Generation." *arXiv preprint arXiv:2201.08670* (2022).
48. Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "Bert: Pre-training of deep bidirectional transformers for language understanding." *arXiv preprint arXiv:1810.04805* (2018).
49. Feng, Zhangyin, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou et al. "Codebert: A pre-trained model for programming and natural languages." *arXiv preprint arXiv:2002.08155* (2020).
50. Yonghui W, Schuster M, Chen Z, et al. Bridging the gap between human and machine translation[J]. arXiv preprint arXiv:1609.08144, 2016.
51. Anonymous. Analyzing codebert’s performance on natural language code search. 2022.
52. Mashhadi E, Hemmati H. Applying codebert for automated program repair of java simple bugs[C]//2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR). IEEE, 2021: 505-509.
53. Pan C, Lu M, Xu B. An empirical study on software defect prediction using codebert model[J]. Applied Sciences, 2021, 11(11): 4793.
54. Khanfir A, Jimenez M, Papadakis M, et al. CodeBERT-nt: code naturalness via CodeBERT[J]. arXiv preprint arXiv:2208.06042, 2022.
55. Guo D, Ren S, Lu S, et al. Graphcodebert: Pre-training code representations with data flow[J]. arXiv preprint arXiv:2009.08366, 2020.
56. Pawlak R, Monperrus M, Petitprez N, et al. Spoon: A library for implementing analyses and transformations of java source code[J]. Software: Practice and Experience, 2016, 46(9): 1155-1179.
57. Lin C Y, Och F J. Orange: a method for evaluating automatic evaluation metrics for machine translation[C]//COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics. 2004: 501-507.
58. Pearce H, Ahmad B, Tan B, et al. An Empirical Cybersecurity Evaluation of GitHub Copilot's Code Contributions[J]. arXiv e-prints, 2021: arXiv: 2108.09293.
59. Song K, Tan X, Qin T, et al. Mass: Masked sequence to sequence pre-training for language generation[J]. arXiv preprint arXiv:1905.02450, 2019.
60. Husain H, Wu H H, Gazit T, et al. Codesearchnet challenge: Evaluating the state of semantic code search[J]. arXiv preprint arXiv:1909.09436, 2019.

致 谢

莫听穿林打叶声，何妨吟啸且徐行。竹杖芒鞋轻胜马，谁怕？一蓑烟雨任平生。

三年时光，转瞬即逝。重复，也是我这三年研究生时光的主旋律。现回忆起，却感慨良多。这期间，有初入科研时的踌躇满志，也有遇困时的焦虑难安。那一个个被push的deadline是在一次次熬夜中完成的。牙刷上的洗面奶和手心的牙膏，将自己从似睡非睡的边缘拉了回来。还记得那年夏天，那件因闭关一月有余而长霉斑的短袖。那一次次成功交稿后的欢呼雀跃，仿佛又打赢了一场胜仗。三年的研究生时光，让我立定了人生所能够追逐的方向。我感谢这条路上遇到的所有人和事，让我可以借力登高，去闻一闻那山顶清爽的风。

生逢于盛世，当不负韶华。感谢江西师范大学计算机信息工程学院为我提供受教育的机会，让我可以再一次的提升自己。研究生三年来，感谢国家和江西师范大学所给予的各种奖励资助，让我可以安心的完成学业。

师长易遇，人师难求。首先我要感谢左正康老师，在我学习路上给予我的鼎立相持，能够让我无畏向前。感谢王昌晶老师，能够让我加入高可信与智能化软件实验室。在这里，我要特别感谢黄箐老师，是这位亦师亦友的战友带我一路披荆斩棘，在我无知时予我知识的海洋，陪着我一路成长。他在我身上倾注了大量的心血，无论是在科研，还是生活当中。他以言传身教的方式，向我树立了一个学习的榜样。此外，向其他各位帮助过我的老师，表示衷心的感谢。幸得遇见诸位良师，令我三年收获颇丰，我将铭记一生。

天下快意之事莫若友，快友之事莫若谈。我要感谢3701各位一起战斗过的战友。一起熬过的夜，一起走过的路。幸有他们的陪伴，让我在这三年的科研时光虽重复但不单调。每当学校人影稀疏，便是3701热血朝天之时。感谢江会文，罗智文，廖颠书，孙彦帮等3701其他战友的帮助，让我可以一次次渡过艰苦的时刻。感谢胡颖，黄志鹏，刘智溢等3727，3729其他同门的陪伴，在我日常生活中所给予的帮助。你们的这份情谊我将铭记于心。

特别地，我还要感谢我的家人。求学二十载有余，他们一直是我坚强的后盾。幸有你们的陪伴和关爱，可以让我勇往无前，感谢你们在背后默默的支持，感谢你们在我情绪低迷时所给予的安慰。

追风赶月莫停留，平芜尽处是春山。我相信秋天俯身采摘种子的人，来年会拥有整个春天。或有困顿，或有煎熬，但只要做好当下可做之事，万事皆可破。

料峭春风吹酒醒，微冷，山头斜照却相迎。回首向来萧瑟处，归去，也无风雨也无晴。既而，别之。

在读期间公开发表论文（著）及科研情况

* 以第一作身份发表学术论文：

1. Prompt-tuned Code Language Model as a Neural Knowledge Base for Type Inference in Statically-Typed Partial Code[[5]](#footnote-5), *IEEE/ACM International Conference on Automated Software Engineering (ASE’2022),* doi: 10.1145/3551349.3556912. CCF-A
2. 1+1>2: Programming Know-What and Know-How Knowledge Fusion Semantic Enrichment and Coherent Application, *IEEE Transactions on Services Computing，2022*，doi: 10.1109/TSC.2022.3207273. CCF-A，SCI.

* 科研项目：

1. 国家自然科学基金（61902162）：基于演化数据语义理解的多元扩展代码搜索方法研究，2020-01至2022-12（参与者）。
2. 江西省研究生创新基金（YC2021-S308）：基于 API 和 TASK 多源数据的知识融合与知识发现研究，2020-09至2023-07 （主持人）。

* 主要在校荣誉：

1. 荣誉名称：2022年智德奖学金获得者

授予单位：江西师范大学，2022.12

1. 荣誉名称：2022年爱国同心奖学金获得者

授予单位：江西师范大学，2022.11

1. 荣誉名称：2021至2022学年，“三好研究生”

授予单位：江西师范大学，2022.11

1. 荣誉名称：2021年江西省研究生数学建模比赛三等奖

授予单位：江西省人民政府学位委员会办公室，2022.08

1. 获奖名称：2021年第十四届认证杯中国数学建模比赛三等奖

授予单位：中国运筹学会计算机系统生物学会分会，2021.01

1. https://stackoverflow.com/questions/16665124/ [↑](#footnote-ref-1)
2. https://stackoverflow.com/questions/3325387/infinite-recursion-with-jackson-json-and-hibernate-jpa-issue [↑](#footnote-ref-2)
3. https://referencesource.microsoft.com/ [↑](#footnote-ref-3)
4. https://github.com/dotnet/roslyn [↑](#footnote-ref-4)
5. 为本毕业论文的扩展 [↑](#footnote-ref-5)