如今的软件工程都会依赖于大型标准库例如: .NET 框架、Java SDK 或者 Android 等等, 这些库会提供大量多样的提前实现好的功能, 例如匹配正则表达式、解析 XML 和发送电子邮件等等. 对于不熟悉的库或者软件框架对于开发者学习 API 的使用是比较困难的[1]. 一个大型的软件库例如: .NET 框架或者 JDK 可能包含成千上万个 API, 例如 JDK8 的基础库中有超过 14 000 个类和方法, 调查结果显示如何选择 API 成为用户编程系统中六个学习障碍之一[2]. 微软 2009 年的一项调查中发现, 67.6% 的受访者反应他们在学习如何使用这些 API 的时候没有充足的示例以供参考[3].

当面对一个 API (Application Program Interface) 相关的任务时, 开发者往往会借助搜索引擎例如: Google 或者 Bing 去查找已有的 API. 他们往往带着两个问题去搜索答案: (1) 对于他们的特定问题使用哪些 API, (2) 如何调用这些 API[4], 即 API 调用序列. 如图 1 所示, 从著名的程序问答网站 Stack Overflow 上截取的开发者和研究人员在软件开发过程中遇到的实际例子. 他们想知道如何去调用已有的 API 函数去解决自己的实际问题. 因此, 研究如何从已有的资源中 (例如: Github 或者 Bitbucket) 挖掘出用户需要的 API 以及相应的 API 调用序列已经成为亟待解决的问题并且有重要的实际意义.

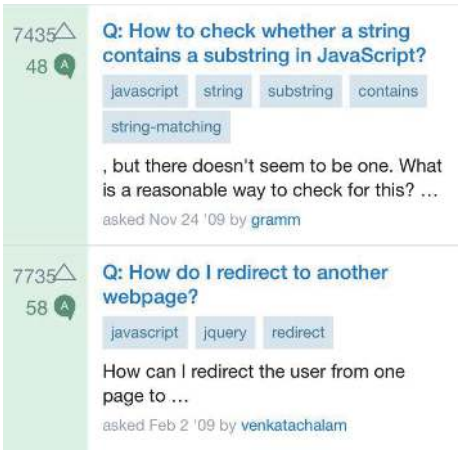

图 1 Stack Overflow 网站上的问答实例

1 问题定义及相关工作

1.1 问题定义

对于一个用户查询输入 Q, 返回一个 API 调用序列 S. Q 是对需要解决问题的英文描述, 例如: “How to open a file?”. 而 S 是解决该问题的 API 调用序列. 本文主要研究 Java 语言相关的问题, 则相应的 S 序列为: File.new(), File.open(), File.close().

1.2 相关工作

按照代码搜索方法的输入类型划分, 相关文献可分为以自由文本作为输入的研究、以 API 作为输入的研究、其它形式作为输入的研究. 而推荐的内容要么是相关的代码片段, 要么是单个 API, 或 API 的调用序列. 这里, 把推荐代码的研究称为代码搜索, 把推荐 API 及其序列的研究称为 API 推荐[5].

1.2.1 代码搜索的研究现状

在代码搜索研究中, 主要是基于基于信息检索技术, Zhong 等人[6]提出为开发者的查询推荐 API 使用模式及对应的代码片段. Bajracharya[7]等人基于开源代码的结构化信息编写的 Sourcer 是代码推荐的典型应用. McMillan[8]等人提出一个基于 PageRank 和 SAN 对复杂任务检索和可视化相关的代码片段的工具 Portfolio. Bajracharya 等人[9]在 Lucene 的多域搜索平台上, 综合 API 使用模式的相似性等一系列特征进行代码片段的推荐. Keivanloo 等人[10]利用向量空间模型、频繁项挖掘等技术为自由文本的查询进行代码推荐. Raghothaman 等人[4]提出了一种新的方法 SWIM, 该方法首先从用户点击数据中找到与用户输入的自由文本查询相关的 APIs, 同时从 Github 的代码中抽取出结构化的 API 调用序列, 通过匹配找到 APIs 相应的调用序列, 进而产生合成的代码片段. Jiang 等人[11]尝试将信息检索和监督学习的方法结合为自由查询提供 Top-K 个相关的代码片段.

1.2.2 API 序列推荐的研究现状

然而传统的代码搜索引擎由于大部分使用了关键字匹配的技术, 导致在处理自然语言查询方面不能有良好的表现[12]. 在 APIs 序列推荐研究中, 近几年也有了较大的发展. Thung 等人[13]整合了用户对于搜索引擎结果的反馈并利用其重新排序返回结果列表使得真正相关的条目排在列表中的较前位置. Niu 等人[14]提出了使用监督学习的方法, 在两阶段的基础上, 借助于多种特征进行 API 的推荐. Rahman 等人[12]尝试从 Stack Overflow 的问答对中抽取出与开发者的自由文本的查询相关的 API. Gu 等人[15]提出了一种基于深度学习的 API 序列推荐方法 DeepAPI, 该方法受机器翻译的启发, 使用经典的 seq2seq 的 RNN 模型, 将“查询-推荐”

任务视作一种语言翻译的过程, 并取得了较好的效果.

1.3 目前 API 序列推荐方法存在的问题

SWIM[4]训练的统计词对齐模型是基于词袋模型的, 而没有考虑到 API 的单词序列以及位置关系, 例如: 它很难区分查询语句“convert date to string”与“convert string to date”. 而之后 Gu 等人[15]提出的 DeepAPI 使用 RNN 模型更好的学习到了句子的语义信息. 经测试 BLEU (BiLingual Evaluation Understudy)[16]值比基于传统模型的 SWIM 提高了约 173%.

但是, DeepAPI 采用的基于 RNN 的模型并没有充分考虑到每个单词的重要程度, 以及词与词之间的相互依赖关系, 在句子过长的情况下会丢失之前的依赖信息, 所以本文使用了完全基于注意力机制的模型, 充分考虑了每个词的重要程度和相互依赖关系, 经实验结果可得 BLEU 值在 Top1 上比 DeepAPI 提高了约 28.5%.

2 基于 Attention 机制模型的设计和实现

由于没有公开的可供训练的数据集, 需要我们自己生成可供训练测试的数据. 我们首先通过网络爬虫从 Github 上爬取关注度比较高的 (即星标数的倒序排列)Java 语言的开源项目. 如图 2 所示, 首先使用 GrouMiner[17]将这些工程文件解析为程序依赖图, 再以方法级为粒度提取出我们需要的 API 序列和注解对, 接下来使用我们设计的基于注意力机制的模型对这些序列对进行训练得到训练模型, 接下来针对每个查询语句使用 beam search 算法给出推荐的序列列表. 具体设计分以下几个方面:

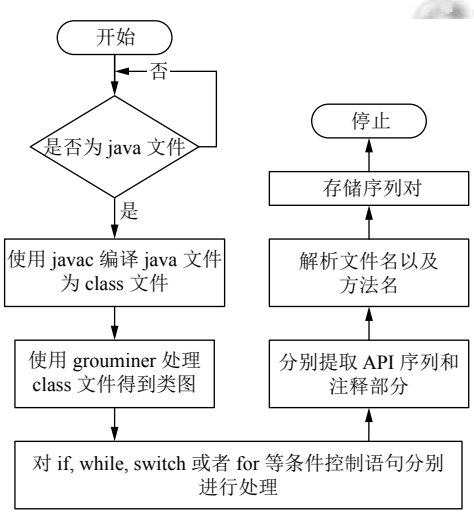

图 2 从 Java 代码提取 API 序列对流程图

2.1 使用 GrouMiner 提取需要的 API 序列

如图 3 所示, 需要将 main 方法上的注释“This method demonstrates square()”提取出来, 而 main 方法中的 API 调用序列为每个对象生成以及调用的方法序列依次追加. 对于条件控制语句例如: if 和 else 语句, 分别将 if 和 else 语句块中的调用序列依次追加, 其他条件控制语句例如 switch 和 while 等同理. 对于提取 API 序列我们使用 GrouMiner, 它是一个基于图方法用来挖掘程序中出现的使用惯例. 如图 4 所示, 通过 GrouMiner 可以将源代码转换为对象依赖图. 对于每个需要解析的方法, 依次生成调用序列以及方法上面的注释对, 以供后面训练使用.

```
/**
* This method demonstrates square().
* @param args Unused.
* @return Nothing.
* @exception IOException On input error.
* @see IOException
*/
public static void main(String args[]) throws IOException
{
   SquareNum ob   new SquareNum();
   double val;
   System.out.println("Enter value to be squared: ");
   val   ob.getNumber();
   val = ob.square(val);
   System.out.println("Squared value is " + val);
}
```

图 3 代码注释示例图

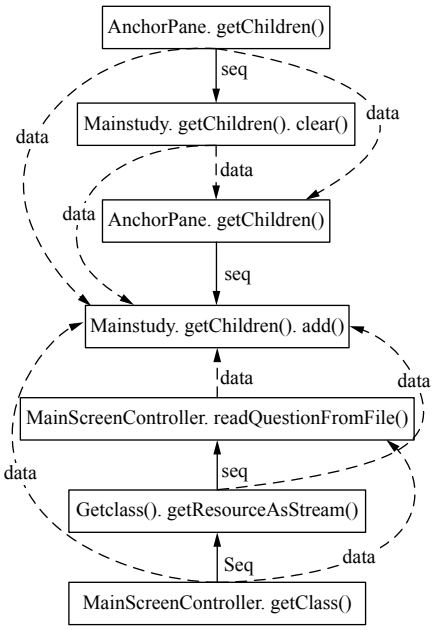

图 4 通过 GrouMiner 将源代码转换为程序依赖图

2.2 注意力机制的 Encoder-Decoder 模型

2.2.1 基于 RNN 的 Encoder-Decoder 模型的缺点

如图 5 所示, 对于编码器 Encoder 就是对输入句

子$x_1, \cdots, x_{T_i}$进行编码, 通过非线性变换f将上一时刻的隐藏层h_{t-1}与当前时刻的输入x_t转化为当前时刻的隐藏层h_t, 结束时刻的h_{T_x}即为c(c为固定长度的向量).

$$h_t = f(h_{t-1}, x_t) \tag{1}$$

$$c = h_{T_x} \tag{2}$$

由此可见: (1) 输入语句序列x_{T_i}(在训练模型的时候即注释语句) 中每一个英文单词对推荐的 API 序列中每一个单词的贡献都是一样的, 但实际上每个单词的贡献是有区别的. (2) 由于输入语句映射在固定长度的向量中, 因此对于较长输入语句, 特别是输入语句长度大于向量长度或者是训练集中没有的长语句时, 会损失上下文特征的很多信息. 所以对于较长语句的处理上, 基于 RNN 的 Encoder-Decoder 模型表现的不好.

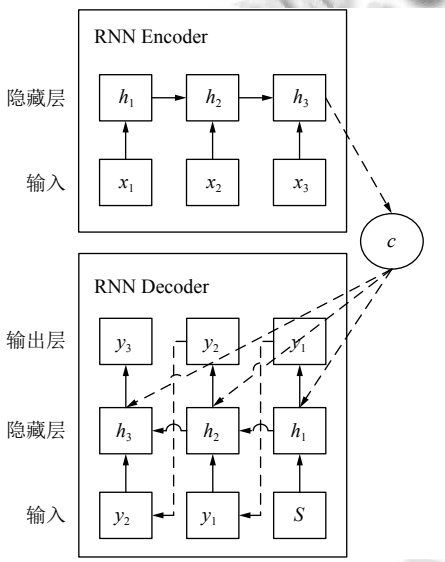

图 5 基于 RNN 的 Encoder-Decoder 模型

2.2.2 使用注意力机制的 Encoder-Decoder 模型

如图 6 所示, 使用注意机制, 我们不再尝试将完整的源序列编码为固定长度的向量. 而是通过 Scaled Dot-Product Attention[18]对输入语句的每个单词进行计算, 得到单词的权重矩阵$Attention(Q,K,V)$, 每一行为一个单词的权重向量. 式 (3) 中, Q、K 和 V 均为矩阵, 每一行为一个向量, d_k为矩阵维度的大小, 模型的实现如图 7 所示. 其中$Attention(Q,K,V)$就是注意力值, 代表对于输出序列的不同部分需要关注输入序列的哪些部分. 所以, 使用注意力机制的 Encoder-Decoder 模型不会受到长距离依赖的影响.

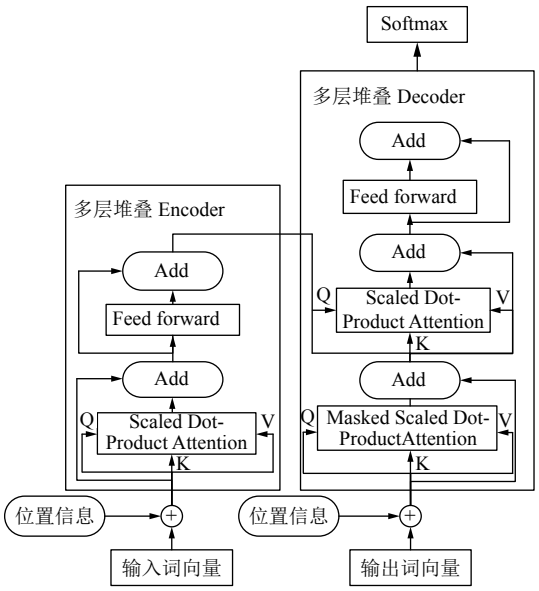

图 6 基于注意力机制的 Encoder-Decoder 模型

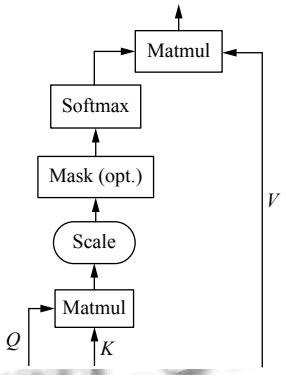

图 7 Scaled Dot-Product Attention 结构

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{3}$$

图 6 中, 编码器网络由多个相同的层堆叠而成, 每层有 2 个子层, 第一层是一个 Scaled Dot-Product Attention 的注意力层, 第二层是一个全连接的 feed-forward 层, 每层都由一个残差网络连接, 解码器同编码器一样也是由多个相同层堆叠而成, 不同的是插入了一个子层, 如图 7 所示, Mask 为可选部分, 这个掩码确保了位置 i 的输出序列只能依赖比 i 小的已知输出, 即通过将其后位置的数据设置为$-\infty$.

3 实验分析

实验总共收集了 649 个关注度较高的 Java 语言的 Github 开源项目, 约 20 GB 的数据量, 共加工处理

得到了 114 364 个<annotation, API sequence>对. 为节约我们训练的时间以及方便测试, 我们只随机取 50 000 个序列对用于训练, 并随机取 500 条序列对用于测试. 实验机器配置如表 1 所示, Encoder -Decoder 模型主要的参数有: 词向量的最大长度设置为 100, 最小出现阈值为 5 次, 隐藏单元为 512 个, 迭代 20 万次.

表 1 实验机器配置

类型	型号	参数
操作系统	Ubuntu	16.04.2 LTS
处理器	Intel(R) Core(TM) i7- 4770	8 核
内存	DDR3	24 GB
显卡	GeForce GTX TITAN X	12 GB

3.1 BLEU 测试结果

BLEU[16]是一种用于在机器翻译领域评估从一种自然语言翻译到另一种自然语言的文本质量的算法. 我们通过 BLEU 计算我们输出的 API 调用序列与标准参考的 API 序列相似度得分可以评判结果的好坏. 如图 8 所示, 我们分别取查询结果的第一个, 前五个以及前十个进行对比得出, 在返回一个查询结果的时候我们的方法高出 DeepAPI 约 28.5%.

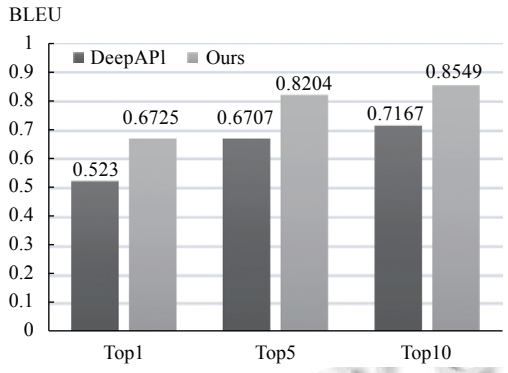

图 8 模型的 BLEU 得分图

4 结论与展望

本文首先使用了 GrouMiner[17]开发了一个可以用来从 Java 代码中提取 API 序列以及注释对的工具. 其次, 针对 RNN(1) 不能充分考虑每个单词的权重, (2) 以及将输入序列压缩为一个固定长度的向量, 损失了很多可用信息, (3) 对于句子过长产生的依赖信息丢失等缺点, 使用了完全基于注意力机制的模型取得了比目前所知最好结果的 DeepAPI 在 BLEU 的 Top1 得分上高出 28.5% 的结果. 我们下一步的工作计划是将该工具可视化以及开源, 供其他科研工作者以及软件开发人员使用.

参考文献

1 Fowkes J, Sutton C. Parameter-free probabilistic API mining across GitHub. Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Seattle, WA, USA. 2016. 254–265.

2 Ko AJ, Myers BA, Aung HH. Six learning barriers in end-user programming systems. Proceedings of 2004 IEEE Symposium on Visual Languages-Human Centric Computing. Rome, Italy. 2004. 199–206.

3 Robillard MP. What makes APIs hard to learn? Answers from developers. IEEE Software, 2009, 26(6): 27–34. [doi: 10.1109/MS.2009.193]

4 Raghothaman M, Wei Y, Hamadi Y. SWIM: Synthesizing what I mean: Code search and idiomatic snippet synthesis. Proceedings of the IEEE/ACM 38th International Conference on Software Engineering. Austin, TX, USA. 2016. 357–367.

5 聂黎明, 江贺, 高国军, 等. 代码搜索与 API 推荐文献分析. 计算机科学, 2017, 44(6A): 475–482. [doi: 10.11896/j.issn.1002-137X.2017.6A.106]

6 Zhong H, Xie T, Zhang L, *et al*. MAPO: Mining and recommending API usage patterns. Proceedings of the 23rd European Conference on ECOOP 2009. Berlin, Heidelberg. 2009. 318–343.

7 Bajracharya S, Ngo T, Linstead E, *et al*. Sourcerer: A search engine for open source code supporting structure-based search. Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications. Portland, OR, USA. 2006. 681–682.

8 McMillan C, Grechanik M, Poshyvanyk D, *et al*. Portfolio: Finding relevant functions and their usage. Proceedings of the 33rd International Conference on Software Engineering. Honolulu, HI, USA. 2011. 111–120.

9 Bajracharya SK, Ossher J, Lopes CV. Leveraging usage similarity for effective retrieval of examples in code repositories. Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Santa Fe, NM, USA. 2010. 157–166.

10 Keivanloo I, Rilling J, Zou Y. Spotting working code examples. Proceedings of the 36th International Conference on Software Engineering. Hyderabad, India. 2014. 664–675.

11 Jiang H, Nie LM, Sun ZY, *et al*. ROSF: Leveraging information retrieval and supervised learning for recommending code snippets. IEEE Transactions on Services Computing, 2019, 12(1): 34–46. [doi: 10.1109/TSC.2016.

2592909]

12 Rahman MM, Roy CK, Lo D. Rack: Automatic API recommendation using crowdsourced knowledge. Proceedings of IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). Suita, Japan. 2016. 349–359.

13 Thung F, Wang SW, Lo D, *et al*. Automatic recommendation of API methods from feature requests. Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering. Silicon Valley, CA, USA. 2013. 290–300.

14 Niu HR, Keivanloo I, Zou Y. Learning to rank code examples for code search engines. Empirical Software Engineering, 2017, 22(1): 259–291. [doi: 10.1007/s10664-015-9421-5]

15 Gu XD, Zhang HY, Zhang DM, *et al*. Deep API learning. Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Seattle, WA, USA. 2016. 631–642.

16 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, PA, USA. 2002. 311–318.

17 Nguyen TT, Nguyen HA, Pham NH, *et al*. Graph-based mining of multiple object usage patterns. Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. Amsterdam, The Netherlands. 2009. 383–392.

18 Vaswani A, Shazeer N, Parmar N, *et al*. Attention is all you need. Advances in Neural Information Processing Systems 30. 2017. 5998–6008.