基于统计语言模型的
代码补全研究

（申请清华大学工学硕士学位论文）

培 养 单 位：软 件 学 院
学　　科：软 件 工 程
研　究　生：王　程　鹏
指 导 教 师：姜　宇　助　理　教　授

二〇一九年六月

Code Completion Based on Statistical Language Model

Thesis Submitted to

Tsinghua University

in partial fulfillment of the requirement

for the degree of

Master of Science

in

Software Engineering

by

Wang Chengpeng

Thesis Supervisor : Assistant Professor Jiang Yu

June, 2019

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；(2) 为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容。

本人保证遵守上述规定。

（保密的论文在解密后应遵守此规定）

作者签名：_____　　导师签名：_____

日　　期：_____　　日　　期：_____

摘 要

在软件开发过程中，开发人员的代码仓库可以为实际编程提供辅助信息。对代码仓库或历史信息的挖掘，能够获取特定的开发模式，并对当前的编辑环境提供代码预测，实现代码补全。代码补全研究内容相对广阔，根据补全对象的不同分为面向 token 的补全、面向 API 的补全以及面向表达式的补全等。

编程语言的 API 为开发人员提供了一个简便的方式进行相应功能的实现。然而选取最佳的 API 常常需要开发人员查找编程语言的文档，这一过程往往繁琐并且耗时。近年来 API 推荐系统不断涌现，很多较为成熟的 API 推荐系统已经集成到开发环境中，辅助开发人员进行程序编写。比如 Eclipse 和 IntelliJ IDEA 两个主流的 Java 集成开发环境都能够进行不同程度的 API 推荐。学术界也提出了很多模型、框架和工具来实现更加准确的 API 补全。

本文研究了基于统计语言模型的代码补全，聚焦到 API 级别的代码补全，包括编程语言文档中的 API 和开发人员自定义的函数或方法。首先使用传统的 n-gram 模型对编程语言进行解析建模，模型能够很好地获取编程语言的局部重复性。同时引入缓存模型，将传统的 n-gram 模型拆分成混合模型。本文对代码关联度展开了研究，尝试将代码挖掘的信息加入到模型训练中，从而提高原有模型在代码补全时的准确率。本文提出启发式的模糊搜索算法，提高补全的鲁棒性，保证工具在多情形下补全的有效性。

本文的主要贡献表现在以下三个方面：

- 基于宽度优先搜索和启发式的策略，提出了一种模糊搜索的算法获取可能的补全的上文，提高了补全算法的鲁棒性。
- 设计并实现了代码关联度挖掘算法，并将其应用于单个 n-gram 模型训练，提高了 API 补全算法的准确率。
- 基于 n-gram 模型和缓存模型实现了 API 级别的代码补全工具 CRMAC，并在实际工程项目中进行了充分的测试，与现有典型工具相比，补全的 Top 3 准确率平均提高 4.79%。

关键词：API 补全；n-gram 模型；缓存模型；模糊搜索；代码关联度挖掘

Abstract

In the software development, the code repositories can provide important information for programming. The mining based on the code repositories and the history of interactions can extract the specific patterns of development and complete code at the current editing point. There are many dimensions of research on code completion, including token completion, API completion, statement completion and etc.

While Application Programming Interface (API) enables an easy and flexible software development process, selecting a best-fit API is often non-straightforward in practice and has always been time-consuming and error-prone. In recent years, API recommendation systems have been introduced to automatically help developers choose an API, e.g., Eclipse and IntelliJ IDEA can generate an internal or user-defined API on the fly. Many models, frameworks and tools are proposed to make API recommendation more accurate.

In this paper, we research on code completion based on statistical language models, focusing on API completion including APIs in program language documentation and methods defined by developers. We firstly use n-gram model to model the program and the model can capture the regularity of program very well. A cache model is introduced at the same time, and a traditional n-gram model is split into two n-gram models. We also study the code relevance and try to take advantage of information obtained in code mining when training the model. Meanwhile, we hope to improve the robustness of the tool by applying heuristic fuzzy searching algorithm.

In this paper, our contributions include the following three aspects:

- We propose a fuzzy searching algorithm to get all possible contexts based on BFS and heuristic strategies, and this technique improves the robustness of the algorithm.
- We implement a module to obtain the code relevance, and apply it to n-gram training phase, which can improve the accuracy of API completion.
- We implement an API completion tool CRMAC based on cache model and test its performance on the real projects. The experimental results demonstrate that CRMAC improves the Top 3 accuracy by 4.79% on average.

Key words: API completion; n-gram model; cache model; fuzzy searching; code relevance mining

目 录

第 1 章 引言 1
1.1 研究背景与意义 1
1.2 相关研究 2
1.2.1 统计语言模型 2
1.2.2 代码补全中的缓存模型 4
1.2.3 编程语言特性 6
1.2.4 工具研发现状 8
1.3 本文的研究内容 10
1.4 本文的主要贡献 11
1.5 本文的组织结构 11
1.6 本章小结 12
第 2 章 代码补全中的语言模型 13
2.1 语言模型的评价标准 13
2.2 n-gram 模型 14
2.2.1 n-gram 模型的前提假设 14
2.2.2 n-gram 模型的缺陷 15
2.2.3 n-gram 模型的改进 16
2.3 PCFG 模型与 PHOG 模型 16
2.4 递归神经网络模型 19
2.5 语言模型的比较 20
2.6 缓存模型 20
2.6.1 缓存模型的特点 20
2.6.2 缓存模型的局限 21
2.7 本章小结 22
第 3 章 基于 BFS 的模糊搜索算法 23
3.1 问题场景 23
3.2 基础知识介绍 24
3.2.1 串匹配算法 24
3.2.2 字符串相似性度量 25
3.3 算法设计与分析 27

3.3.1 算法设计 27
3.3.2 算法复杂度分析 29
3.3.3 算法局限 30
3.4 本章小结 31
第 4 章 代码关联度挖掘原理与实现 32
4.1 代码关联度挖掘的动机 32
4.1.1 经典缓存模型的局限 32
4.1.2 基于代码关联度挖掘的缓存模型 33
4.2 代码关联度挖掘算法 33
4.2.1 算法框架 33
4.2.2 算法设计 34
4.2.3 算法复杂度分析 35
4.2.4 算法局限 36
4.3 代码关联度挖掘的其他实现 36
4.4 本章小结 37
第 5 章 工具实现、应用与评估 38
5.1 代码补全工具架构与实现 38
5.1.1 总体架构 38
5.1.2 补全流程 39
5.2 实验设计与结果分析 41
5.2.1 实验设计 41
5.2.2 结果分析 42
5.3 本章小结 44
第 6 章 总结展望 45
6.1 工作总结 45
6.2 研究展望 46
6.2.1 语言模型 46
6.2.2 代码挖掘 47
6.2.3 模糊搜索 48
6.2.4 中间语言表示 49
6.2.5 代码补全中的其他方法 50
6.3 本章小结 51
参考文献 52

致 谢 56

声 明 57

个人简历、在学期间发表的学术论文与研究成果 58

第 1 章　引言

本章简要介绍基于统计语言模型的代码补全的背景及意义，并对相关研究进行综述。之后简要介绍本文的主要工作及贡献，并介绍论文的章节安排。

1.1　研究背景与意义

随着计算机科学的发展，各种集成开发环境不断涌现，比如 Visual Studio，Eclipse，IntelliJ IDEA 等。在编写大规模代码的过程中，编程人员需要有良好的编程体验，希望能有自动补全或者提示的选项。比如在编写 Java 代码时，开发者希望能够在调用 API 时有提示选项，以节省查找相应文档的时间。因此这需要具有较为精准的代码补全功能的开发工具辅助开发人员完成代码编写任务。

代码补全具有不同层次的分类，包括 API 级别、表达式级别、语句块级别的补全等。另外，还有一部分的代码补全关注类初始化代码的自动生成。由于不同层次补全的侧重点不同，采用的补全算法以及补全的准确率差异较大。在 Eclipse 和 IntelliJ IDEA 中，代码补全为 API 级别和 token 级别的补全。Eclipse 中的某些插件能对自定义类补全初始化类的代码。然而在实际编程的过程中，仅仅有这两个级别的补全是远远不够的，实际开发还需要有表达式级别和语句块级别的补全。

在 API 级别的补全中，不同集成开发环境采取的补全方式有所不同。在 Eclipse 中，IDE 只会选取 Java 文档中的 API 进行推荐，根据对象的类型推断出可能的 Java 文档中的 API；而在 IntelliJ IDEA 中，IDE 会同时选取 Java 文档中的 API 以及用户自定义的函数进行推荐。表达式级别和语句块级别的代码补全在主流 IDE 的默认补全功能中尚未实现。此外，近年来学界有相关研究实现了类和接口的初始化代码，并形成了 Eclipse 插件。

本课题重点针对 Java 代码的 API 自动补全算法展开研究。论文的主要创新点与贡献包括：

（1）结合统计语言模型中的 n-gram 模型，参考国际同行的工作，将其与缓存模型相结合，使得在本地代码库中的补全正确率有较大提升；

（2）将模糊搜索的方法应用于 API 补全中，大大提高了 API 补全算法在不同上文场景下的鲁棒性；

（3）提出了代码关联度挖掘算法，并整合进代码补全算法中，一定程度上提高了原算法的准确性；

（4）实现了基于统计语言模型的代码补全工具 CRMAC，并使用 6 个 Java 开

源项目对工具进行评估，在评估 API 补全的准确性的同时，分析了不同配置参数对补全准确性的影响。

1.2　相关研究

本章将简要综述代码补全的理论研究现状与工具研发的情况。其中理论研究部分将从统计语言模型、代码补全中的缓存模型、编程语言特性三个方面展开。工具研发部分将列举学界和工业界近年来新提出的工具或已普遍使用的成熟工具，简要剖析其实现原理以及各自的优缺点。

1.2.1　统计语言模型

基于统计语言模型的代码补全通过借鉴自然语言处理中的文本建模的方法，实现对代码的建模。下面综述自然语言处理中几种常用的统计语言模型。第2章将对各个统计语言模型进行详细讨论。

1.2.1.1　N-gram 模型

N-gram 模型用于构建字符串 s 的概率分布 $p(s)$，其中 $p(s)$ 为“字符串 s 是合法句子”的概率。假设字符串 $s = w_1 w_2 ... w_l$，由条件概率的定义，其概率密度函数可以表示为：

$$p(s) = p(w_1) \cdot p(w_2|w_1) \cdot p(w_3|w_1 w_2) \cdot ... \cdot p(w_l|w_1 ... w_{l-1}) \tag{1-1}$$

在上面的公式中，产生第 i 个词的概率由已经产生的 i-1 个词 $w_1 w_2 ... w_{i-1}$ 决定。对于第 i 个词，前 i-1 个词为它的历史。在实际语言中，一个合法句子的第 i 个词出现的概率一般只与它之前的 n-1 个词相关，因此：

$$p(w_i|w_1 .. w_{i-1}) = p(w_i|w_{i-n+1} ... w_{i-1}) \tag{1-2}$$

其中 n 为 n-gram 模型中的参数，由此可以改写公式 (1-1)，将其中每个因子中的字符数控制在 n 以下（包括 n）。以 3-gram 为例, (1-1) 式可改写为：

$$p(s) = p(w_1) \cdot p(w_2|w_1) \cdot p(w_3|w_1 w_2) \cdot ... \cdot p(w_l|w_{l-2} w_{l-1}) \tag{1-3}$$

当 token 的取值空间很大，即字典容量很大时，n-gram 模型中的数据具有极强的稀疏性，即存在大量的 abc，使得 $p(a|bc) = 0$，此时上述连乘式的结果为 0。然而在

很多情况下，某一个位置上的 token 可能存在异常变异，从而导致整个序列的概率为 0，这样的概率估算是不合理的。为了解决数据稀疏性的问题，往往需要使用**回退操作**和**平滑操作**对 n-gram 模型做处理。在 n-gram 模型中，若 $p(a_i|a_{i-n+1}...a_{i-1}) = 0$，则可尝试使用回退操作，即假定

$$p(a_i|a_{i-n+1}...a_{i-1}) = bow(a_{i-n+1}...a_{i-1}) * p(a_i|a_{i-n+2}...a_{i-1}) \tag{1-4}$$

其中 $bow(a_{i-n+1}...a_{i-1})$ 为序列的回退权重。在实际计算中，需要重复进行回退操作，直到计算出的概率不为 0 为止。N-gram 模型中的平滑操作有很多种类，其中常用的平滑操作包括 Laplacian 平滑、Lidstone 平滑、Good-Turing 平滑和 Witten-Bell 平滑。从效果上看，平滑操作与回退操作都是寻找一个非零的概率值替代公式（1-1）中的零因子。

1.2.1.2　PCFG 与 PHOG 模型

除了 n-gram 模型外，在自然语言处理领域和代码自动补全领域，常常也采用**上下文无关文法**及其变种对语言进行建模。其中 **PCFG(Probabilistic Context Free Grammar)** 为一种有效的建模方式[1]。它在传统的上下文无关文法的基础上，将概率引入到每个产生式中，从而句型的推导从确定过程变为随机过程。这样的改进使得模型的表现能力得以增强，但是由于在对代码进行实际建模时，PCFG 的拓展性具有一定的局限性。尽管近几年出现了基于 PCFG 的代码补全，但是都将其进行改进并与其他模型进行了组合，比如在 Martin Vechev 的 API 补全的工具中将 PCFG 推广成为 **PHOG（Probabilistic High Order Grammar)**，在 PCFG 的基础上加入了语言的上下文信息，并与决策树模型进行组合，取得很好的效果[2]。

1.2.1.3　递归神经网络模型

随着深度学习技术的发展，传统的统计语言模型也逐渐被深度学习模型替代，或者与深度学习模型进行组合使用。比如**递归神经网络模型 (RNN)** 因其自身的网络结构在处理自然语言和代码片段上就有突出的优势[3] [4]。包括 n-gram 模型、PCFG 模型、PHOG 模型在内的大部分统计语言模型能够有效捕捉语言的局部特性，但是很难获取语言跨区域的特性，而递归神经网络模型能够有效地解决统计语言模型面临的这一困境。

1.2.2　代码补全中的缓存模型

Zhaopeng Tu 在 2015 年提出了新的缓存模型，发布了代码补全工具 CACHECA[5][6]。缓存模型一定程度上借鉴了 Witten-Bell 平滑的思想，它在语料库上训练得到 n-gram 模型 A，同时在当前编辑的文件（缓存文件）上训练出另一个 n-gram 模型 B。两个模型对应的概率分布为 P_{corpus}、P_{cache}，将两个模型进行差值拟合得到缓存模型。缓存模型与 Witten-Bell 平滑不同之处在于其权重因子不是由极大似然估计得到，而是根据 n 元组出现的次数以及关注因子 γ 确定。

缓存模型下当前位置的 token 为 t_i 的概率计算公式为：

$$p(t_i|h, cache) = \frac{\gamma}{\gamma + H} p_{corpus}(t_i|h) + \frac{H}{\gamma + H} p_{cache}(t_i|h) \tag{1-5}$$

其中 H 为当前位置下前缀 h 在缓存文件中出现的次数，关注因子 γ 取正数。当 γ 固定时，若前缀出现次数较少，则通过语料库训练出的 n-gram 模型占主要部分；若前缀出现次数较多且远远大于 γ，则在缓存文件上训练出的 n-gram 模型占主要部分。这样的设计显然是合理的。当缓存文件中出现了大量前缀 h 时，意味着当前文件包含了大量与前缀 h 有关的信息，从而应当有侧重性地优先考虑当前文件对应的模型。

Algorithm 1 GetProbFromCM

Require: $model_{corpus}$: n-gram model trained in code corpus
$model_{cache}$: n-gram model trained in current project
context: content in the current file
γ: concentration factor

Ensure: p(h): probability of suffix of current context

```
h ← ExtractCompletionPrefix(context)
H ← GetOccurenceCount(model_cache, h)
weight1 ← γ/(γ + H)
weight2 ← H/(γ + H)
prob1 ← GetProbFromComponent(model_corpus, h)
prob2 ← GetProbFromComponent(model_cache, h)
prob ← prob1 * weight1 + prob2 * weight2
return prob
```

缓存模型的核心算法见算法 1。缓存模型首先获取当前编程环境下的上文，提

取出补全的前缀。第 2 行的函数 GetOccurenceCount 用于获取补全前缀在缓存中出现的次数。第 3 行到第 7 行根据公式 (1-5) 将两个 n-gram 模型加权拟合，从而得到根据缓存模型产生的概率估计值。

由于 Eclipse 本身带有 API 补全功能，并且 Eclipse 的 API 补全是基于对象的类型检查实现的，因此候选的 API 都符合编程语言文档中的相关定义。在进行 API 补全设计时，有必要充分利用这些信息。因此在 CACHECA 的设计中，通过启发式的组合策略，将 Eclipse 的补全信息加入到自定义的补全方案中，模型如图 1.1所示。当两个列表中出现推荐选项数目都不小于 3 时，按照以下两个步骤从上到下构建组合后的推荐列表：

（1）选取在两个列表的 Top 3 中都出现的选项。

（2）将两个列表中的剩余选项从上到下依次交叉添加到新的列表中。

如果某个列表中的推荐选项数小于 3，则在构建组合列表时，在上述两个步骤之间加入操作：将推荐选项数小于 3 的列表中的 Top 1 选项加入组合列表中。完整的算法见算法 2。

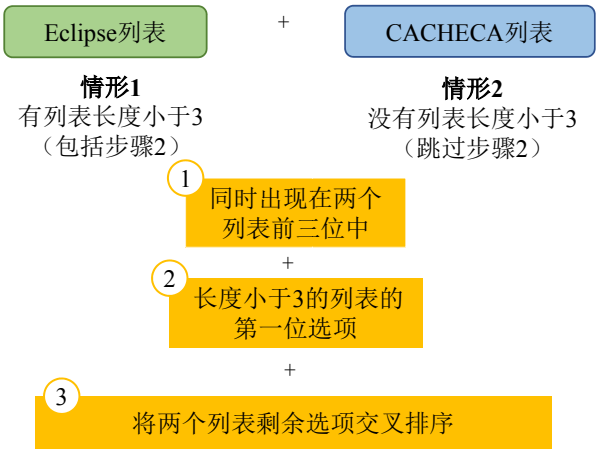

图 1.1　CACHECA 中的混合模型

Algorithm 2 $\mathcal{MSE}$(eproposal,nproposals,maxrank,minlen)

Require: eproposals and nproposals are ordered sets of Eclipse and N-gram proposals.

1: elong:=$\{p \in \text{eproposals}[1..\text{maxrank}] \mid strlen(p) > 6\}$

2: **if** elong $\neq \emptyset$ **then**

3:　**return** eproposals[1..maxrank]

4: **end if**

5: **return** eproposals$[1..\lceil \frac{\text{maxrank}}{2} \rceil] \circ$ nproposals$[1..\lfloor \frac{\text{maxrank}}{2} \rfloor]$

可以看出，CACHECA 的混合模型的启发式策略是合理的。因为 Eclipse 的列表和自定义模型生成的列表中的推荐选项均按照可能性大小从上到下排列，所以在组合两个列表时，同时在前三位出现的选项一定是最可能的选项。交叉两个列表中的推荐选项可以将两个列表中相对重要的选项都排在组合列表的前面。而当某个列表中的选项小于 3 时，表明对应列表的第一项极有可能是正确的补全结果，因此在第一步后加入该选项。

1.2.3　编程语言特性

基于统计语言模型的代码补全将编程语言视为一种特殊的自然语言进行处理。虽然相比于自然语言，编程语言更加复杂、灵活、多变，但是在实际编程的过程中，编程人员编写的代码大部分都是简单重复的且易于理解的。在这一点上，编程语言与自然语言十分相似。依据这个观察，有相关研究借鉴自然语言处理中的语言建模方法，将统计语言模型引入到代码建模中实现代码补全。

对编程语言特性的研究是代码补全、代码综合工作的基础，通过对其特性的调研和分析，能够挖掘出编程语言的内在属性，从而为代码补全、代码综合的算法提供充分的理论支撑。Abram Hindle 等人在 2012 年对软件代码的特性做了系统的调研[7]，通过分析不同类型的软件系统中的代码的混乱度，回答了代码补全中的几个重要问题，得到了几个基本结论。

结论一：编程语言在理论上是复杂灵活的，但编程人员实际编程中写出的代码大多为简单重复的代码。因此可以通过代码的统计学性质挖掘出代码中潜在的统计语言模型，再利用得到的统计语言模型完成特定的软件工程任务。

结论二：基于语料库的统计语言模型能够捕捉到编程语言具有的局部规则性，其规则程度甚至高于英语等自然语言。自然语言在使用时场景丰富，用法多样；而编程语言的语法决定其具有更确定的语言结构。因此编程语言的交叉熵低于自然语言，具有更强的规则性。

结论三：语言模型获取的局部规则性不仅仅属于编程语言本身，每个代码工程也有其特定的局部规则性。不同类别的工程往往会采用特定的编程语言，编程框架、类库的广泛使用导致同一类别的代码具有更多相似的结构特征。

Abram Hindle 等人在他们的论文中用 n-gram 模型对包括 Ant 在内的 10 个 Java 开源项目进行建模并计算它们的交叉熵，将计算结果与英语语料库的交叉熵做比较，得到的曲线如图1.2所示，其中折线为英语语料库的交叉熵图像，离散的矩形区域为 10 个 Java 项目的交叉熵分布。数据表明随着 n-gram 模型的参数 n 的增大，自然语言和编程语言的交叉熵随之减小，且自然语言的交叉熵始终大于编程语言。

这说明编程语言比自然语言更加规则，混乱度更低。因此 n-gram 模型能够有效地捕捉编程语言的规则性。

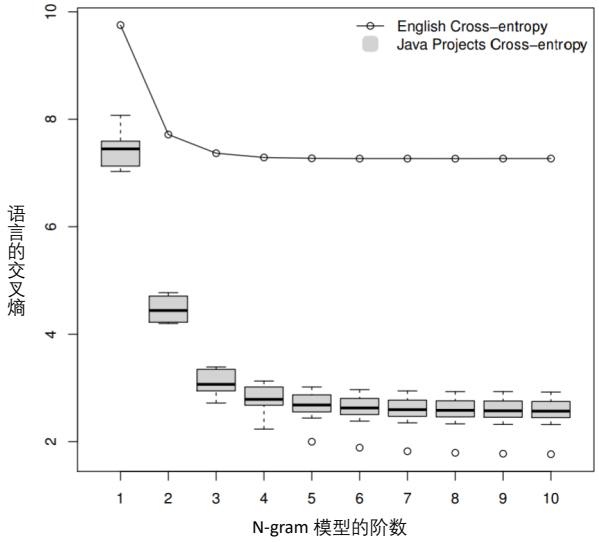

图 1.2　编程语言与英语语料库的交叉熵对比

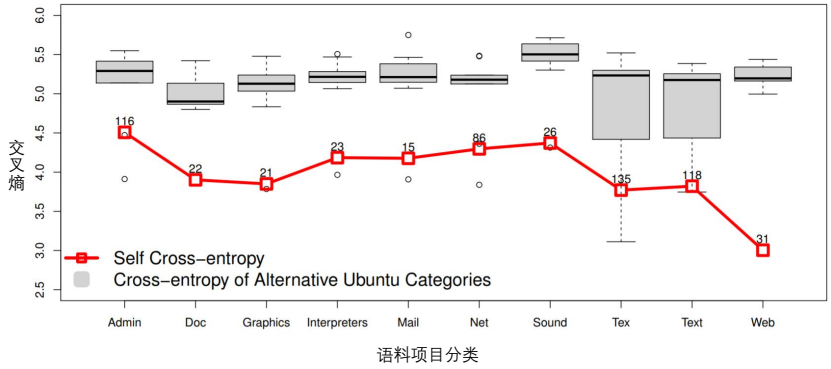

图 1.3　不同分类的代码交叉熵对比

Abram Hindle 等人还比较了统计语言模型在捕捉不同领域代码规则性的差异。图1.3为实验结果，实验数据表明不同领域的代码的规则程度具有一定的差异性。其中 Web 程序的交叉熵最低，这表明 Web 程序的规则性相比其他分类下的代码规则性更强。这一点符合直观上的判断，因为 Web 应用代码大多使用了框架进行编程，结构上的重复性和规则性很强。

需要进一步说明的是编程语言的规则性和重复性都是局部的性质，对于大段的代码，编程语言因其固有的灵活多变性，规则性表现得并不明显。这种局部空间上的规则性和重复性成为了代码自动补全的理论基础，同时也是 Abram Hindle 等人对代码的局部特性进行研究的必要前提。

Abram Hindle 等人的工作表明，实际工程中的代码具有局部空间上的规则性和重复性，因此采用统计语言模型能够很好地挖掘出代码仓库或历史代码中的统计规律。同时，由于编程语言具有自然语言类似的上下文关联，在处理自然语言处理中常用的一些深度学习模型也不断应用于代码自动补全领域。

1.2.1小节简要介绍了几种常见的语言模型。在代码补全中，常用的模型为 n-gram 模型、LSTM 模型（一种 RNN 模型）、基于 PCFG 的 PHOG 模型。这三类模型都已经形成较成熟的工具，其中最为成熟且简单有效的模型为 n-gram 模型。

N-gram 模型应用到代码补全的方式多样，最直接的方法为将代码文档视为普通的自然语言文档，每一个 token 视为自然语言文档中的一个单词，从而计算当前环境下末尾序列为在语料库中出现次数最多时对应的的 token，将其加入候选 token 列表。但是这种直接序列化的方法存在一个明显的问题：源代码文件中存在很多无效冗余的 token，比如分号、空格、括号等其他无效信息。因此，另一种有效的做法为过滤掉源码中与代码补全无关的字符，之后再序列化进行建模预测。

对于表达式级别的代码补全，需要对源代码做更高层次的抽象，即设计能够有效提取代码中关键语义信息的中间语言。比如遍历源代码的抽象语法树，将源代码转化成自定义的中间语言。在这个过程中，不仅仅能过滤掉冗余无效的 token，同时能够挖掘出代码文件、语句块之间的关系，从而提高表达式补全的准确性。

对于其他几种模型，比如 PHOG 模型常常与其他的机器学习模型进行组合在 API 补全上取得了很好地效果。此外，伴随着深度学习技术的发展以及硬件的不断升级，以 LSTM 为代表的递归神经网络在处理大规模代码时的能力不断增强，也逐渐成为了代码补全的一个主流方法。

1.2.4　工具研发现状

随着编程语言研究的不断深入发展，学界和工业界涌现出很多优秀的代码补全插件和研究工具。目前常用的代码补全工具包括在 Eclipse 和 IntelliJ IDEA 中的 Code Assistant，两者在各自的 IDE 中具有良好的用户体验。近年来，学术界也出现了不少与代码补全有关的优秀工作，包括 Zhaopeng Tu 在 2015 年发表的论文中发布的工具 CACHECA[5][6]，以及 ETH 的 Veselin Raychev 等人在 2014 年 PLDI 上发表的论文中发布的工具 SLANG[8]、CMU 开发的类初始化函数补全工具 Calcite[9]、清华大学发布的表达式补全工具 PCC[10] 等。表格 1.1列举了这些工具，并简要说明了这些工具的原理和优缺点。其他具有类似原理的 API 补全工具未列入到表格，包括 API 补全工具 MAPO[11]、Precise[12] 等。

表 1.1　代码补全工具

工具	说明
Code Assistant in Eclipse	基于对象的类型检查实现 API 级别的补全，默认版本只能根据 Java 文档中的 API 进行推荐。由于其原理为类型检查，提供的 API 选项均可以通过编译时检查。Eclipse 支持各类插件开发，可以自定义补全插件，增强 Eclipse 的代码补全功能。
Code Assistant in IntelliJ IDEA	API 级别的补全。不仅能够推荐 Java 文档中的 API，还能推荐开发人员自定义的方法和函数。定义类成员变量的获取方法时，能够补全函数名称、特定的关键词等 token，但不能补全表达式。尽管相比 Eclipse 默认补全工具更有效，但是补全场景依然有限。
Calcite	CMU 开发的 Eclipse 插件，能够补全类和接口的初始化代码。不能进行 API 补全、表达式补全，因此能够补全的对象有限。
CACHECA	实现 API 级别的代码补全的 Eclipse 插件，根据当前编程环境中的代码文件和语料库代码分别训练出两个 n-gram 模型，将两者组合成缓存模型。同时，采用启发式组合策略与 Eclipse 中默认的 Code Assistant 结合，将两种推荐列表合并。缺点为只能推荐 token，不能补全表达式。
SLANG	实现了 API 级别的代码补全，原理为将 p 层隐藏层的 RNN 与最大熵模型结合，利用组合后的模型训练出代码补全模型。优点在于算法框架可配置性强，可以使用其他语言模型替代 RNN 和最大熵模型；同时 RNN 的引入可以获取代码跨区域的模式，比统计语言模型更加精准。缺点为不能进行表达式级别的补全。
PCC	PCC 工具为表达式级别补全的 Eclipse 插件，同时支持 API 级别的补全。将 Java 代码转化成中间语言后，采用 n-gram 模型对中间语言进行建模。与前面的工具相比优势在于补全对象不仅仅局限于 API 或更一般的 token，但与 CACHECA 相比缺点在于无法提取出当前编辑环境下的代码信息。

在表格 1.1中列举的代码补全工具中，PCC 工具为面向 Java 语言的表达式补全工具[10]。与其他诸多主流代码补全工具不同，PCC 工具的补全对象不仅仅局限

于以 API 为代表的 token，对于表达式的补全支持使其成为现阶段代码补全工具中的佼佼者。它基于合成的中间语言表示，使得表达式在序列化过程中能够保持语义信息，并利用 n-gram 模型补全中间语言，经过反向合成后得到补全后的源代码。本文的工作在 PCC 工具的基础上进行改进，通过使用更加有效的语言模型和挖掘算法提高补全效果。由于时间所限，本项工作将补全对象限定在以 API 为代表的 token 级别上，表达式级别的补全可以类比 PCC 工具中的设计进行推广。

通过调研上述代码补全工具及相关研究工作，发现代码补全领域存在以下挑战。后续章节将依次设计对应的模型及算法解决以下问题。

1. **代码仓库中的代码信息未有效利用**。尽管各类补全工具采用强度不同的语言模型提取待补全项目的信息，外部代码仓库中海量的代码却未被充分发掘。随着大数据时代的发展，代码本身作为数据也应当进行充分挖掘。

2. **补全的鲁棒性待提升**。由于代码补全涉及复杂的人机交互过程，补全上文难免出现由于编程人员的疏忽造成的噪音数据。噪音数据会导致补全的前缀与编程人员期望的内容有所出入，补全结果因此会出现很大偏差。现有的工作中大多忽略了这一普遍存在的现象，并没有提出行之有效的解决方案。

3. **代码挖掘缺乏针对性**。由于补全上文的代码与语料库及项目内其他位置的代码片段具有不同的关联度，不同区域内的代码提供的信息具有不同的价值。单一项目内的代码信息需要根据补全上文进行有针对性的提取，将有价值的信息反馈给补全引擎。现有工作中尚未出现针对这一观察设计的挖掘算法。

1.3　本文的研究内容

本文工作将围绕缓存模型、模糊搜索、关联度挖掘三个方面展开，并实验论证不同参数配置对补全效果的影响，主要研究内容具体包括以下四个部分：

1. 经过广泛调研多种统计语言模型并进行分析比较，选择 n-gram 模型作为基础模型。根据实验预处理结果，3-gram 能很好地捕捉编程语言中的结构，故将其作为缓存模型的内核模型。

2. 设计并实现了一个启发式模糊搜索的方案，当进行补全时，根据代码上文搜索出所有可能的补全上文。在实际补全的场景中，补全的上文在 n-gram 模型中可能不存在，但是存在多个类似的上文。如果按照精确匹配的条件进行补全，往往会出现无候选项的情况。然而模糊搜索可以获取所有可能的上文，补全结果更具实际参考价值。

3. 在传统模型的基础上引入代码关联度挖掘模块。通过提取不同层次的代码特征、比较代码片段之间的关联度，调整 n-gram 模型中不同 n 元组的比重。在传

统的 n-gram 模型和缓存模型中，语料库和缓存文件各个位置上的 n 元组的权重相同，即不论 n 元组在文件的哪个位置上出现，对整体的贡献度均为 1。但是如果当前正在编辑的代码片段与语料库或者缓存文件中的某个代码片段的关联度很高，则这段代码中的信息对于当前编辑的代码片段更重要，因而它在 n-gram 模型中对应的比重应该更高。

4. 实现了 API 补全工具 CRMAC，并使用交叉熵和混乱度评价语言模型，将提出的代码关联度挖掘算法、优化后的缓存模型与传统的 n-gram 模型、传统的缓存模型进行比较，展示算法在 API 补全中的贡献效果，并分析算法存在的优缺点。除缓存模型外，算法框架中的模糊搜索算法和代码关联度挖掘算法都可以使用其他具有相同功能的算法实例进行配置。参数化的特性保证了框架的高扩展性和可配置性。

1.4　本文的主要贡献

本文的主要贡献如下：

1. 基于统计语言模型中的 n-gram 模型，参考国际同行的工作，将其与缓存模型相结合，在本地代码库中的补全正确率有较大提升。

2. 将模糊搜索算法应用于 API 补全中，获取更多可能的补全上文，提高了 API 补全算法的鲁棒性。

3. 设计并实现了代码关联度挖掘算法，并整合进代码补全算法中，一定程度上提高了原算法的准确性。

4. 实现了基于统计语言模型的代码补全工具 CRMAC，并在实际项目中进行实验评估，将现有典型工具的 Top 3 准确率平均提高 4.79%，Top 10 准确率平均提高 5.28%，平均 Top 10 准确率达到 74.4%。

1.5　本文的组织结构

本文第1章首先介绍了研究背景与意义、统计语言模型的知识、本文的研究内容与贡献；第2章介绍了代码自动补全中的语言模型，将简要介绍 n-gram 模型的优缺点以及相应改进与拓展；第3章介绍了基于宽度优先搜索的启发式模糊搜索算法，通过模糊搜索获取所有可能的补全上文，提高代码补全的鲁棒性；第4章主要介绍了代码关联度挖掘的原理与实现，提出了代码关联度挖掘的一般框架，并实现了一个具体算法实例，整合到传统的缓存模型中；第5章对工具进行实现、应用与评估，分析不同的参数配置对 API 补全正确率的影响，论证模糊搜索算法和代

码关联度挖掘算法的有效性；第6章为总结与展望，总结了本文的主要研究工作和贡献，并分析研究中值得进一步深入探究的问题以及可能的应用场景。

1.6　本章小结

本章介绍了基于统计语言模型的代码补全的问题背景，系统分析了现有研究工作和工具的基本原理及优缺点，并简要介绍了主要工作与贡献。

第 2 章　代码补全中的语言模型

在进行代码自动补全时，需要对代码进行建模。由于编程语言具有与自然语言类似的局部空间上的规则性和重复性，因此采用自然语言处理中常用的统计语言模型对编程语言建模，其中 n-gram 模型为最简单也是相对有效的模型[13] [14]。同时 n-gram 模型具有很好的可拓展性，多个 n-gram 模型相互组合能够形成表达能力更强的模型。本章中将简要介绍 n-gram 模型与其他几种主流的代码补全模型，分析其优缺点，最后讨论如何将传统的 n-gram 模型拓展为缓存模型。

2.1　语言模型的评价标准

给定一个具有重复特性和可预测的语料库或代码库，可以通过某些语言模型对自然语言或编程语言进行建模。在讨论具体的建模方式的之前，选取一个量化的指标对模型的建模能力进行评价十分重要。一个好的语言模型应当可以充分地捕捉语料中重复的模式，并能很好地预测一篇文档是否属于某类相似的文档语料库。换而言之，模型不会认为该文档与语料库中的文档相比具有很大的差异性。在自然语言处理中，研究者定义**交叉熵 (Cross Entropy)**[15]，其完整定义如下。

给定一篇文档，将其序列化后得到一个长度为 n 的字符序列 $s = a_1 a_2 ... a_n$。对于语言模型 $\mathcal{M}$，假设文档 s 被语言模型 $\mathcal{M}$ 赋予的概率为 $p_{\mathcal{M}}(s)$。交叉熵的定义由公式 (2-1) 给出：

$$H_{\mathcal{M}}(s) = -\frac{1}{n} log p_{\mathcal{M}}(a_1 ... a_n) \tag{2-1}$$

由条件概率公式，公式 (2-1) 可以化为：

$$H_{\mathcal{M}}(s) = -\frac{1}{n} \sum_{i=1}^{n} log p_{\mathcal{M}}(a_i | a_1 ... a_{i-1}) \tag{2-2}$$

类似的可以定义**困惑度（Perplexity）**如下：

$$P_{\mathcal{M}}(s) = 2^{H_{\mathcal{M}}(s)} \tag{2-3}$$

这一度量指标可以表现出语言模型对某一个文档的“意外程度”，即新的文档为模型提供的信息量。当一个文档和语料库中的文档高度相似时，这个新文档不能为

根据该语料库建模出的模型提供充分的有效信息。相反，当一个文档很与众不同时，相应的度量结果会很大。因此，对于给定的文档和语料库，好的语言模型具有较低的交叉熵，从而说明其能够充分提取出语料库中的信息，以导致新的文档并不能对其有新的信息补充。

下面将讨论几种典型的语言模型。由于交叉熵适合度量统计语言模型，因此仅仅使用交叉熵对 n-gram 模型及其变种等统计语言模型进行评价。

2.2　n-gram 模型

本小节将围绕 n-gram 模型[16] 展开介绍，分别介绍 n-gram 模型的前提假设、模型缺陷、针对缺陷进行的模型改进操作等内容。同时分析 n-gram 模型的计算复杂度，根据分析结果讨论在实际使用过程中采用的经验设置，并结合研究数据论证经验设置的合理性。

2.2.1　n-gram 模型的前提假设

假设字符串 $s = w_1 w_2 ... w_l$，由条件概率的定义，其概率密度函数可以表示为：

$$p(s) = p(w_1) \cdot p(w_2|w_1) \cdot p(w_3|w_1 w_2) \cdot ... \cdot p(w_l|w_1 ... w_{l-1}) \tag{2-4}$$

由条件概率的计算公式，上面公式严格成立。公式中的的每一项为一个条件概率，即由已经产生的 i-1 个词 $w_1 w_2 ... w_{i-1}$ 决定的产生第 i 个词的概率。对于第 i 个词而言，前 i-1 个词为它的历史。

无论是自然语言还是编程语言，都具有一定程度的局部性，即某个位置的 token 大多只与其相邻的几个位置上的 token 有关。因此在实际语言中，在一个合法句子中第 i 个词出现的概率一般只与它之前的 n-1 个词相关，从而在 n-gram 模型中，做出如下假设：

$$p(w_i|w_1 .. w_{i-1}) = p(w_i|w_{i-n+1} ... w_{i-1}) \tag{2-5}$$

n 为 n-gram 模型中的参数。以 3-gram 为例，在 n-gram 模型假设下，一个序列为合法句子的概率计算公式可以近似为：

$$p(s) = p(w_1) \cdot p(w_2|w_1) \cdot p(w_3|w_1 w_2) \cdot ... \cdot p(w_l|w_{l-2} w_{l-1}) \tag{2-6}$$

这个便是 n-gram 模型核心的思想。在应用于代码自动补全这一特定任务时，根据补全环境的上文信息后计算使得 $p(s)$ 取得最大值的 w_l 作为补全的 API。由于在实际的 API 补全中可能会推荐多个可能的 API，因此一般会列举出前 k 个最可能的 API 供用户选择。

2.2.2　n-gram 模型的缺陷

尽管 n-gram 模型通俗易懂且在大多数的问题中非常有效，但是模型本身存在很明显的缺陷：随着模型阶数 n 的增大，模型的稀疏现象变得极为明显，即对于大多数 n-gram，乘积项中的每一项通常为 0，导致存储的复杂度大大提高。

为了弥补 n-gram 模型的这一缺陷，研究者提出了多种方法对其进行改进，改进的方法为对取值为 0 的条件概率进行估计，用一个非 0 值进行替代。由此产生了回退操作和平滑操作。然而这样的处理方法缺少合理的理论解释，因此这一缺陷成为 n-gram 模型在对语言进行建模时的软肋。

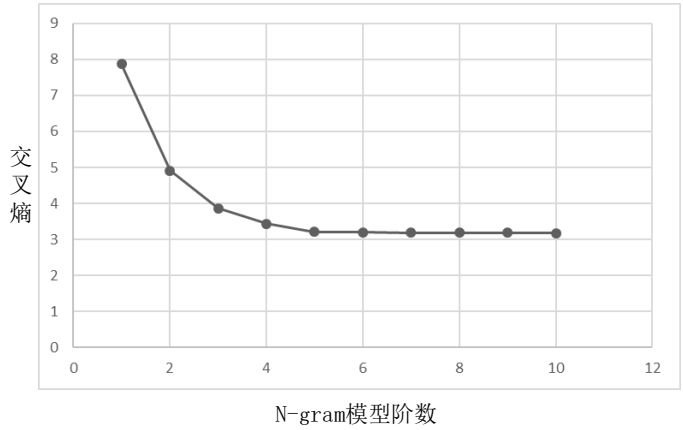

图 2.1　阶数在 10 以内的 n-gram 的模型熵

另一方面，随着模型阶数 n 的增大，训练 n-gram 模型的时间复杂度也呈现指数增长，而这一缺陷理论上不能避免。为了避免在实际工具中出现 API 推荐时间开销大、交互体验不友好的缺陷，在经过充分的项目调研后发现：在大多数实际工程中，3-gram 已经能够很好地捕捉到代码的局部特性，更高阶的 n-gram 模型不会提供更多的有效信息[5]。在图2.1中，计算了 n-gram 模型在阶数为 1 到 10 之间的模型交叉熵，数据图像表明比 3-gram 更复杂的模型几乎不能使模型的交叉熵有明显下降，因此采用 3-gram 对语言进行建模是合理的。幸运的是，3-gram 模型在一般的硬件下都能在很短的时间内完成模型训练和 API 推荐。

2.2.3　n-gram 模型的改进

N-gram 的回退操作的核心思想为：使用一个概率值非 0 的低阶 n-gram 的概率估计高阶 n-gram 的概率，在低阶概率值非 0 的 n-gram 的基础上，乘以取值为 0 到 1 之间的回退因子作为惩罚项。由于本项研究工作采用平滑操作对 n-gram 模型进行改进，回退操作这里不展开详细讨论。

N-gram 模型中的平滑操作有很多种类，其中常用的平滑操作有：Laplacian 平滑、Lidstone 平滑、Good-Turing 平滑和 Witten-Bell 平滑。

Lidstone 平滑将每个 token 出现的次数加上相同的正参数 α，定义为公式 (2-7)。其中 $\#t$ 为 n 元组 t 出现的次数，N 为 n 元组总数。

$$p'(t) = \frac{\#t + \alpha}{N + \alpha} \tag{2-7}$$

Laplacian 平滑为 Lidstone 平滑的特例。当 α 取值为 1 时，Lidstone 平滑即为 Laplacian 平滑；当 α 取值接近与 0 时，Lidstone 平滑与无平滑的情形近似。

Good-Turing 平滑由 I.J.Good 于 1953 年援引图灵的方法提出来。在 Good-Turing 平滑中，假定任何一个出现 r 次的 n 元组都出现了 r' 次，其中：

$$r' = (r + 1) * \frac{n_{r+1}}{n_r} \tag{2-8}$$

n_r 为训练语料库中恰好出现 r 次的 n 元组的数目。

相比于之前基于加法的平滑操作，**Witten-Bell 平滑**为一种特殊的平滑方法，它采用高阶和低阶 n-gram 模型的线性差值，差值的权重根据极大似然估计得到。但是由于极大似然估计的计算复杂，Witten-Bell 平滑不如前面几种平滑操作应用广泛。

在 CRMAC 的实现中，我们采用了 Lidstone 平滑，平滑参数 α 取经验值 0.5。

2.3　PCFG 模型与 PHOG 模型

在语言的形式化建模中，常常用上下文无关文法对自然语言进行分析。然而传统的上下文无关文法是一个确定型的文法，不能很好地刻画自然语言中的随机部分，因此有研究者在传统的上下文无关文法中加入了概率的思想，提出了 **PCFG 模型（Probabilistic Context Free Grammar）**[17]。PCFG 模型为上下文无关文法的概率形式，将原本确定的产生式赋予了特定的概率，从而可以刻画具有随机性质的语言。

作为一种统计语言模型，PCFG 模型可以应用于编程语言的建模。图2.2和图2.3是 PCFG 模型对编程语言建模的一个简单的例子。

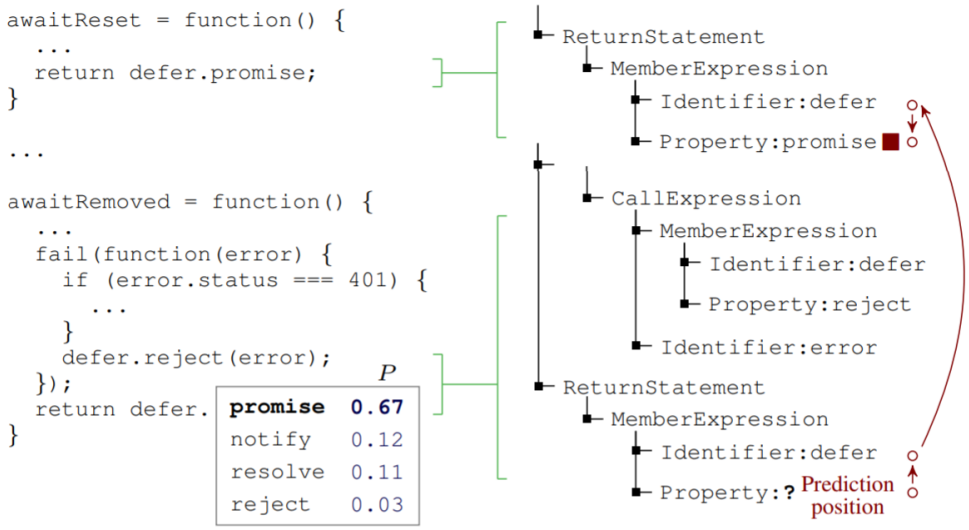

图 2.2　PCFG 与 PHOG 建模举例

PCFG 规则	P
Property → x	0.005
Property → y	0.003
Property → notify	0.002
Property → promise	0.001

图 2.3　PCFG 文法举例

在 PCFG 模型中，语言由事先定义好的文法形成的，文法中的每一条规则形如 $\alpha \rightarrow \beta$。比如图2.3中的文法中，左边部分为 CFG 文法，右边部分为 PCFG 文法中赋予每个产生式的概率。在这个例子中，非终结符 Property 有多个产生式，转移概率分别为 0.005, 0.003, 0.002, 0.001。由此可以计算由该文法产生一段给定文档的概率，从而实现了与 n-gram 模型相同的功能。

同样，PCFG 模型在预测下一个字符时也需要计算转移概率。在上述文法和例子中，下一个 token 为 promise 的概率为 0.67，notify 的概率为 0.12，resolve 的概率为 0.11，reject 的概率为 0.03。

然而由于 PCFG 模型本质上仍然是上下文无关文法随机化的版本，只能对上下文无关语言进行建模。然而编程语言具有很强的上下文相关特性，用 PCFG 模型对编程语言建模难以捕捉到上下文信息，因此 ETH 的研究人员 Pavol Bielik 等人在

PCFG 模型的基础上提出了表现能力更强的概率模型 **PHOG（Probabilistic High Order Grammar)**[2][17]，并实现了基于 PHOG 模型的 API 补全工具。PHOG 模型在 PCFG 模型的基础上加入了上下文信息，能够解析上下文敏感语言。图2.4为使用 PHOG 模型对图2.2中的代码进行建模的文法，可以将 PHOG 模型视为向原始的 PCFG 模型中的非终结符传入一个上下文信息作为参数。由此可见 PHOG 模型对代码建模的能力比 PCFG 模型更强。

1. 获取关注文本$context$
2. 使用PHOG规则：$\alpha[context] \rightarrow \beta$

PCFG 规则	P
Property[promise] → promise	0.67
Property[promise] → notify	0.12
Property[promise] → resolve	0.11
Property[promise] → reject	0.03

图 2.4　PHOG 文法举例

ETH 研究人员的相关工作表明，PHOG 模型比传统 n-gram 模型具有更好的补全效果，实验数据如图2.5所示。PHOG 模型中引入了上下文信息，s 可将 PCFG 模型的补全错误率平均下降了 20% 以上。

MODEL	LOG-PROBABILITY	ERROR RATE
NON-TERMINALS		
PCFG	-1.99	48.5%
3-GRAM	-1.46	30.8%
10-GRAM	-2.55	35.6%
PHOG	**-1.09**	**25.9%**
TERMINALS		
PCFG	-6.32	49.9%
3-GRAM	-3.92	28.7%
10-GRAM	-4.70	29.0%
PHOG	**-2.50**	**18.5%**

图 2.5　PHOG 模型与 PCFG 模型、n-gram 模型的 API 补全数据

由此可见，尽管编程语言具有与自然语言相似的特性，但是由于编程语言本身的特异性，仍然需要对传统的统计语言模型进行进一步的修改。比如 PCFG 模型不能捕捉语言的上下文信息，因此将其修改成更高阶的 PHOG 模型。这一思想为之后对 CACHECA 模型中的缓存模型进行优化提供了一个很好地思路：结合编程语言的某些独有性质，将传统的自然语言处理中的统计语言模型进行优化，使其更加适用于编程语言的建模。

2.4　递归神经网络模型

递归神经网络（RNN）可用于语言建模等诸多自然语言处理领域，常见的框架为 LSTM 等。随着计算能力的不断提高，神经网络应用场景越来越广泛，递归神经网络在自然语言处理中的应用愈加成熟，可以预测序列为合法句子的概率，从而达到与 n-gram 模型相同的效果[4]。然而与 n-gram 模型不同的是，RNN 模型获取的不仅仅是一个单词与其之前几个连续单词的关系，还可以获取较长距离的单词之间的关系模式。比如在图2.6中，v^i 和 y^i 为 d 个实数构成的向量，其中 d 为字典中单词的个数。c^i 为由 p 个实数构成的向量，p 为隐藏层的的个数。RNN 使用两个函数 f 和 g，通过以下两步估计 $s = w_1 w_2 ... w_m$ 为合法句子的概率。

（1）将所有的 v^i 都设置为 0，除了 $v^i_{w^i}$ 对应的位置；

（2）根据函数 f 和函数 g 计算 $c^i = f(v^i, c_{i-1})$，$y_i = g(c_i)$。

向量 y_i 可用于估计下一个单词 w_{i+1} 的概率。ETH 的 Veselin Raychev 等人将 RNN 与最大熵模型结合，提出了 **RNNME 模型**，开发出代码补全工具 SLANG[8]。

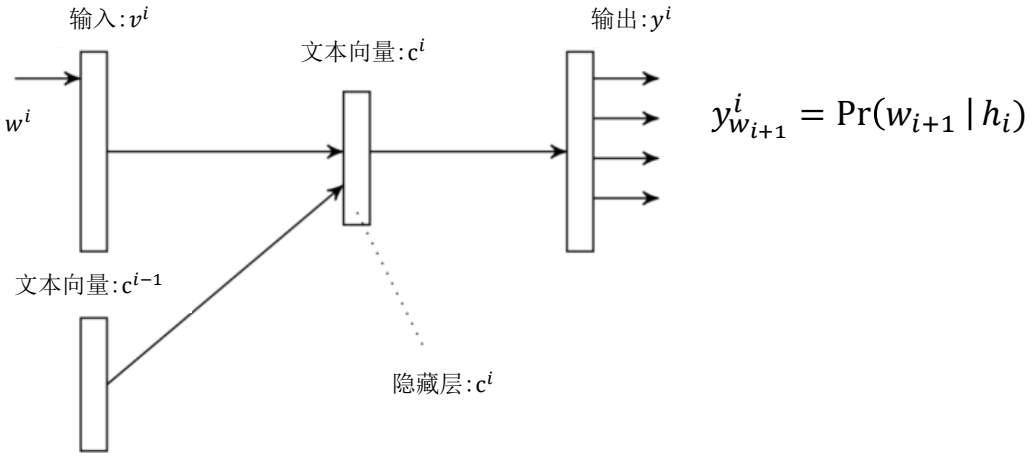

图 2.6　LSTM 模型举例

另一种可以替代 n-gram 和 PCFG、PHOG 的模型是 **Log-bilinear 模型**[18]。这一模型适用于特征集合很大时的语言建模，最终能学习到能够反映特征与预测值相关性的最优权重。这些特征一般情况下是通过简单的方法提取的，比如之前 10 个终结符和非终结符，或者给定一个任务进行的特征设计。

以上简要综述了常见的几种统计语言模型。如今常用的语言模型都在语言模型训练工具 SRILM 中有对应实现[19]。SRILM 用来构建和应用统计语言模型，实现语音识别、统计标注和切分以及机器翻译等任务。它将常见的语言模型封装成

支持以上任务的 C++ 类库，研发人员可通过编写各种脚本执行相应自然语言处理的任务。

2.5　语言模型的比较

代码自动补全领域常用以上三种模型，针对不同的补全对象通常会选取不同的语言模型。

N-gram 模型相比另外两种模型适用面更广，由于其本质是对代码做序列化后进行的 token 预测，在不同的预处理和后续操作下，n-gram 模型可以支持 API 补全、更一般的 token 级别的补全。另外，如果引入中间语言表示后对原始代码做序列化处理，n-gram 模型还可以进行表达式级别的补全。

PHOG 模型和 LSTM 模型在大部分工作中都用于 API 级别的补全，然而由于 PHOG 模型自身拓展性有限，LSTM 模型对计算能力有一定要求不适合后续集成，因此在本工作中未选取两者。

N-gram 模型形式简单，不同的 n-gram 模型之间易于组合，同时还可以对代码仓库进行分割，分别训练出多个 n-gram 模型以发掘不同代码部分中的统计特性。在本项工作中，选取了 n-gram 模型作为核心模型，同时参考了基于它的缓存模型并进行了进一步推广，实现了代码关联度挖掘与 n-gram 模型的有效结合。

2.6　缓存模型

使用传统的 n-gram 模型对代码补全任务建模时，通常将外部语料库和当前待补全的项目看成一个整体，训练成同一个 n-gram 模型。然而相关研究工作表明，项目内部的 n-gram 具有一定的特异性，即在某些情形下，当前项目中的 n-gram 对补全的参考价值更大[5] [20]。比如在图1.3中，不同类型的项目代码的交叉熵有明显的区别。因此一个很自然的想法为：将外部语料库与当前项目分别训练成两个 n-gram 模型。补全时根据上文信息，将两个 n-gram 模型进行拟合，得到的条件概率更加精确。根据这一思想构造的模型为缓存模型[5] [6]。这一思想在自然语言处理中也有相关的应用[21]。

2.6.1　缓存模型的特点

图2.7为缓存模型的架构图。缓存模型充分利用了当前项目和语料库之间的差异性，将传统 n-gram 训练模式中的单个模型训练拆分成两个模型，之后对两个模

型进行线性组合。线性组合的公式如下所示：

$$p(t_i|h, cache) = \frac{\gamma}{\gamma + H} p_{corpus}(t_i|h) + \frac{H}{\gamma + H} p_{cache}(t_i|h) \tag{2-9}$$

其中 γ 为**关注因子（Concentration Factor）**。如果当前上文在缓存文件（即当前项目）中出现次数较多，则缓存部分的权重大于语料库部分，从而更倾向于推荐缓存中的 API。从直观上看这样的权值调整是很合理的，上面的线性组合的公式能够取得预期的效果。

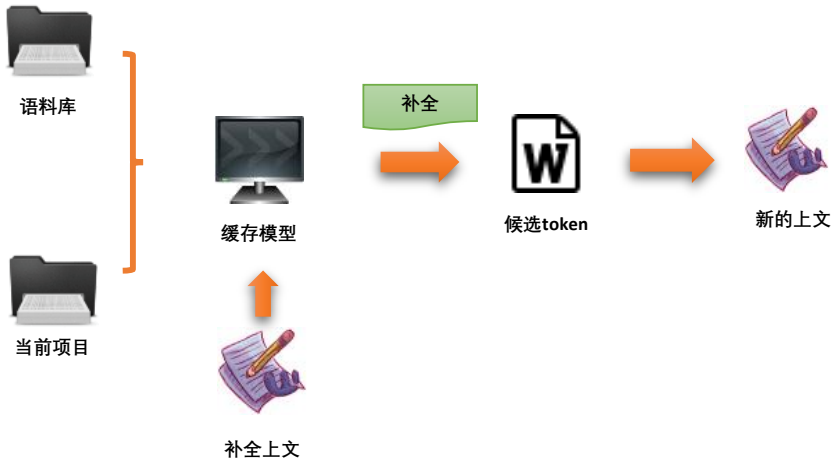

图 2.7　缓存模型架构图

在第5章中的实验分析部分中，将详细分析缓存模型的性能随关注因子 γ 的变化关系。由公式（2-9）可知，当关注因子 γ 取 0 时，缓存部分失效，整个模型退化为在外部语料库中训练得到的 n-gram 模型。第5章中的实验数据表明，当 γ 增大到一定的程度时，模型的补全正确率趋于稳定。

2.6.2　缓存模型的局限

尽管缓存模型有效地将外界语料库与当前项目代码进行分离，但是并未利用不同文件之间的特异性，而仅仅利用了项目级别的代码特异性。为了改进这一局限，第4章将设计一种基于代码关联度挖掘的算法，赋予不同文件的 n-gram 不同的训练权重。这一过程等价于对于 n-gram 进行了两次权重调整：第一次为缓存模型进行的项目级别的权重调整，权重即为公式（2-9）中的系数；第二次为代码关联度挖掘中计算出的文件之间的关联度。两次权重调整使得补全的准确率有明显

提升。如果两次权值调整统一起来看，可以认为这一改进后的模型为细粒度的缓存模型，缓存的级别从项目级别细化到文件级别。

2.7　本章小结

本章首先介绍了传统的 n-gram 模型，简要分析了 n-gram 模型的前提假设、模型缺陷以及相应的改进方法；其次系统分析了主流代码补全工具采用的 PHOG 模型和递归神经网络模型，并将其与 n-gram 模型进行比较，分析三者之间在代码补全场景中的优缺点；最后在 n-gram 模型的基础上拓展产生了缓存模型，通过改进传统的缓存模型的局限性提出了一种可行的优化方案。

第 3 章　基于 BFS 的模糊搜索算法

在处理 API 补全请求时，需要获取补全处的上文信息，将上文信息传入补全模块中。然而在大多数情形下，当前的上文在语料库和当前项目的其他部分并没有完整出现，从而在训练得到的模型中找不到与之完全匹配的序列。因此需要设计一种模糊搜索算法，找到与当前上文近似的序列进行补全。

3.1　问题场景

在理想情况下，假定编程人员在编写代码时不出现输入错误，即在考虑 API 补全时，补全处的上文信息完全正确。然而这仅仅是一个理想假设，在实际编程中，由于编程人员的疏忽，总是会出现上文信息中某些位置上的 token 错误。如果仅仅考虑上文信息的后继 API，补全的结果会受到上文信息中错误 token 的影响。另一方面，实际编写的代码与代码仓库中的历史代码并不一定完全相同，需要设置一个容错机制防止出现“过拟合”。因此更加合理、鲁棒的设计需要允许上文出现少量错误，在进行补全时查找多个可能的匹配前缀进行 API 补全。

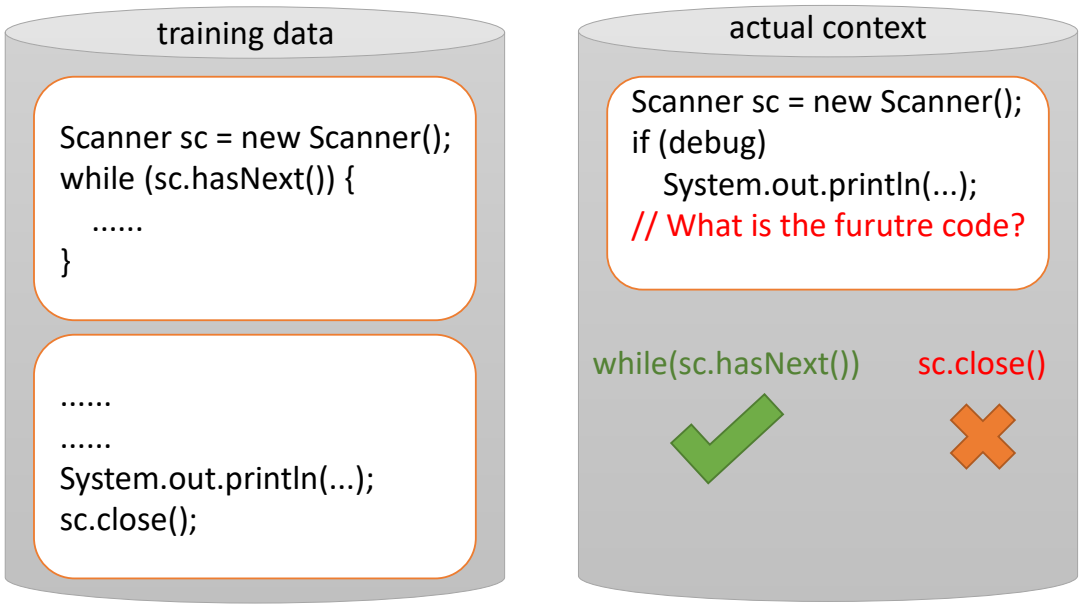

图 3.1　模糊搜索应用场景

图3.1为一个可能的场景。左图为代码仓库中的训练数据，如果按照标准的序列化的匹配结果，右图中的补全结果应该为 sc.close()。然而如果通过模糊搜索进

行上文空间的搜索会发现，更可能的下文应该为 while(sc.hasNext())。这是一个可能的场景，对于 API 补全也具有一定的参考价值。

因此，为了解决由种种因素导致的当前上文与代码仓库中的历史代码不能完全匹配的情况，在标准的字符串匹配算法的基础上加入模糊搜索机制，找到与当前上文相似的历史代码片段进行补全。下面将介绍传统的串匹配算法以及字符串的相似性度量，并结合一些启发式算法设计出一种模糊搜索的算法实例。

3.2　基础知识介绍

本小节将介绍模糊搜索的基础内容，包括经典的串匹配算法和字符串相似性度量。其中串匹配算法将简要介绍 KMP 算法、AC 自动机算法[22] 和字符串近似匹配算法。字符串相似性度量将介绍包括汉明距离、雅卡尔指数、简单匹配系数、李距离、编辑距离等度量指标。

3.2.1　串匹配算法

KMP 算法接受模式串 P 和主文本字符串 S，匹配 S 中与 P 完全相同的连续子串。最简单的暴力算法通过依次将 P 从 S 的头部移动到尾部、从左到右逐个比较每个字符，算法时间复杂度为 $O(mn)$，其中 m 和 n 分别为 P 和 S 的长度。KMP 算法在传统的暴力算法的基础上进行了优化，取得了 $O(m+n)$ 的时间复杂度。通过计算模式串 P 的失效函数，得到当进行逐位匹配失败的情形下模式串 P 可以安全向右移动的最大长度。在暴力算法下，每次匹配失败后，模式串 P 向右移动一步。然而由于模式串 P 的已匹配成功的部分带有主文本字符串 S 在当前匹配位置上的信息，因此可以通过对模式串 P 的结构进行分析从而优化算法。

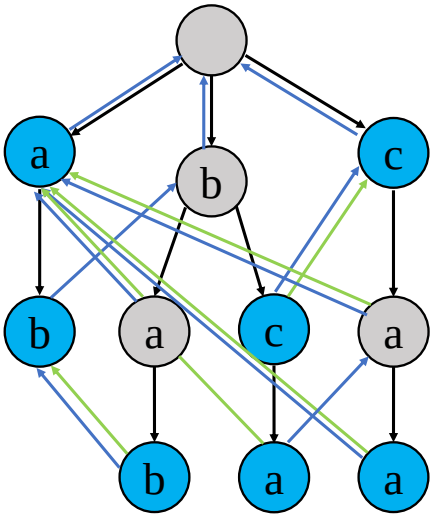

图 3.2　含有失效指针的 Trie 树

AC 自动机算法与 KMP 算法类似，通过构造辅助数据结构对模式串 P 进行结构分析[22]。AC 自动机算法通过构造一个有限自动机，在 Trie 树中添加失效指针，允许在匹配失败时进行回退，从而达到与 KMP 算法中失效函数类似的效果。比如 Trie 树的单词 cat 匹配失败，但是 Trie 树中存在单词 cart，故失效指针会指向其前缀 ca。因此失效指针会使得匹配过程转向某前缀的其他分支，避免重复匹配前缀，大大提高了算法效率。

除了以上两种算法以外，字符串匹配的经典算法还包括 **Boore-Moore 算法**[23]。这些算法的功能为精准的连续串匹配。然而在实际应用中，往往需要进行字符串近似匹配。一般而言，字符串近似匹配算法分为在线算法和离线算法两类。类似于 KMP 算法和 AC 自动机算法，在线算法中模式串需要被预处理，但是主文本串没有预处理。经典的在线算法包括 **Sellers 算法**、**Wagner-Fisher 算法**和 **Bitap 算法**等。但是由于在线算法在数据量很大时复杂度较高，因此采用文本预处理、索引等技术大幅度加速算法。离线算法不对模式串和主文本串进行预处理，应用面没有在线算法广泛。在前期调研中发现，经典的模糊搜索算法都需要进行模式串的预处理，然而考虑到本工作中代码补全的上文字符串长度有限，采用一般的离线算法即可取得很好的效果。具体设计细节见3.3小节。

3.2.2　字符串相似性度量

由于在进行模糊搜索后需要将得到的可能的上文字符串按照相似度从高到低进行排序，因此在排序之前需要对每个字符串进行相似性度量。下面介绍几种常用的字符串相似性度量方法。

汉明距离为两个等长字符串中对应位置的不同字符的个数[24]。汉明距离只能度量两个等长的字符串之间的相似性。比如如果需要比较两个等长的 01 串的汉明距离，可以直接将两个 01 串进行按位异或，统计结果中 1 的个数即为两个 01 串之间的汉明距离。对于一般的两个字符串，直接遍历判断每一位上字符是否相等即可。

雅卡尔指数又称为**并交比**[25]，定义为两个集合交集大小与并集大小之间的比例，如公式 (3-1) 所示。其中 $|A|$ 表示集合 A 中元素的个数，$A \cup B$ 和 $A \cap B$ 分别表示 A 和 B 的并集和交集。

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{3-1}$$

在实际计算过程中，为了避免同时计算交集和并集，可以由容斥原理将公式 (3-1)

转化成公式 (3-2)。

$$J(A,B)=\frac{|A\cap B|}{|A|+|B|-|A\cap B|} \tag{3-2}$$

在度量两个字符串之间的相似度时，可以将两者视为集合而忽略字符的顺序关系，根据上述公式计算出两者的雅卡尔指数。

简单匹配系数用于比较每一位上的属性为二值属性的等长字符串的相似度[26]。在图3.3中，假设两个字符串 A 和 B 长度均为 n，每一位上有两种取值，则简单匹配系数 SMC 的计算公式见公式 (3-3)，其中 M_{01} 表示同一位置下 A 的数值为 0、B 的数值为 1 的字符数，其他符号同理。可以看出 SMC 是汉明距离在特殊情形下归一化的结果。

$$SMC=\frac{M_{00}+M_{11}}{M_{00}+M_{11}+M_{01}+M_{10}} \tag{3-3}$$

		A	
		0	1
B	0	M_{00}	M_{10}
	1	M_{01}	M_{11}

图 3.3　简单匹配系数原理图

李距离用于评价两个长度为 n、q 进制的字符串之间的距离[27]，定义为

$$LD=\sum_{i=1}^{n}\{min|x_i-y_i|,q-|x_i-y_i|\} \tag{3-4}$$

当 q=2 或 3 时，李距离等价于汉明距离。根据李距离的定义可以看出，李距离原则上只能用于度量数字串之间的相似度。在实际的字符串相似度度量时，如果采用李距离作为度量指标，需要将字符串一一对应到数字串。这里的字符串需要满足字符取值空间确定且有限这一条件，否则无法进行数字串转化。

编辑距离度量将一个字符串变为另一个字符串所需要的最小操作数[28]。由于对操作种类的限制，编辑距离有很多不同的版本。若只允许删除和增加字符，编

辑距离的计算等价于计算**最长公共子序列距离**[29]；若考虑删除、添加、修改字符，编辑距离为**文莱斯坦距离**。

由于在编程过程中，基本操作为添加和删除，修改操作通常是通过删除和添加操作进行复合得到，因此在下面的算法中采用最长公共子序列距离进行字符串的相似性度量。最长公共子序列距离的计算采用经典的动态规划算法，其中状态转移方程如下所示：

$$LCS(X_i, Y_j) = \begin{cases} \varnothing & if \quad i = 0 \quad or \quad j = 0 \\ LCS(X_{i-1}, Y_{j-1}) \frown x_i & if \quad x_i = y_j \\ longest(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)) & if \quad x_i \neq y_j \end{cases}$$

其中 longest 操作获取两个字符串中的较长者。根据状态转移方程，下面是最长公共子序列求解举例。

	Ø	A	G	C	A	T
Ø	∅	∅	∅	∅	∅	∅
G	∅	↑,←∅	↖(G)	←(G)	←(G)	←(G)
A	∅	↖(A)	↑,←(A) & (G)	↑,←(A) & (G)	↖(GA)	←(GA)
C	∅	↑(A)	↑,←(A) & (G)	↖(AC) & (GC)	↑,←(AC) & (GC) & (GA)	↑,←(AC) & (GC) & (GA)

图 3.4　最长公共子序列计算举例

根据状态转移方程，需要通过一个二维矩阵保存求解路径，最终可以通过图3.4中的存储结果构造出 AGCAT 和 GAC 中的最长公共子序列 AC、GC、GA。

3.3　算法设计与分析

上一节简单讨论了传统的字符串匹配算法，包括准确的连续字符串匹配以及在线模糊匹配算法，并介绍了几种常用的字符串相似性度量方法。这一节将详细阐述基于 BFS 的模糊搜索算法的细节，提出一般性的框架，并根据最长公共子序列距离进行字符串排序，从而为 API 补全选取可能的上文信息。

3.3.1　算法设计

在第2章讨论了 n-gram 模型的回退和平滑操作，这两个操作用于处理概率计算公式中连乘因子取值为 0 的情形。然而在很多情况下，语料库和缓存文件中甚

至连前缀都没有出现过。因此为了处理这一特殊情况，需要设计算法寻找出当前前缀相似度最高的序列，并根据该序列进行 token 推荐。

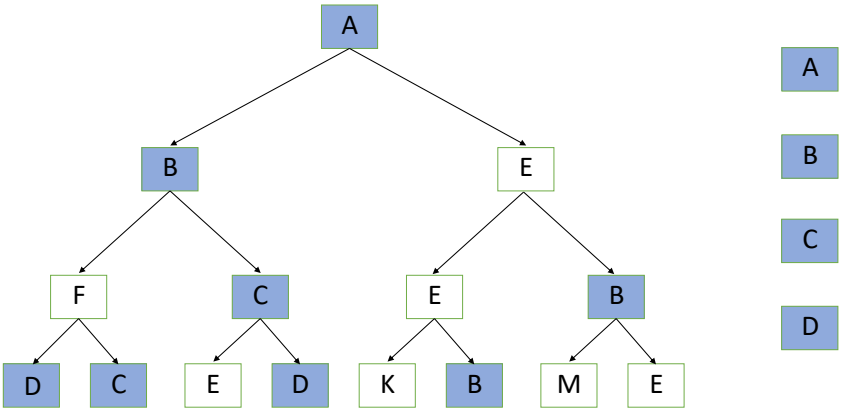

图 3.5　模糊搜索举例

由于搜索空间中的序列长度一般与当前前缀的序列长度相同，可以尝试使用宽度优先搜索的方法查找出与当前前缀序列含有相同的 token 的序列，并通过计算最长公共子序列的长度度量两个序列之间的相似度，选取相似度高的序列作为替代序列。图3.5为基于 BFS 的模糊搜索算法举例，这里将搜索空间中的序列长度的最大值设置为当前前缀长度。

假设当前上文为 ABCD，从 A 开始进行宽度有限搜索，经过 4 层 BFS 得到图3.5中的搜索树。树中的蓝色节点为在上文中出现过的字符，因此可能的上文集合为 {ABCD，ABD，ABC，AB}。在实际的算法中，搜索的层数要比上文的长度稍大。可取搜索的层数为上文长度的 1.5 倍，从而能得到更大的搜索空间。

算法的伪代码如算法3所示。第 5 行通过调用字符串库函数获取上文字符串的长度，并设置搜索空间中字符串的最大长度为上文字符串的 1.5 倍。第 6 行到第 17 行通过宽度优先搜索反复调用补全函数获取所有可能的补全上文字符串。其中，第 9 行函数 InferNextTokens 接收一个 token 序列作为输入，通过调用补全算法得到补全的 token 候选集合。第 19 行将得到的可能的补全上文按照字符串相似度从高到低进行排序，返回排序后的结果。整个算法是基于宽度优先搜索的框架进行设计的，其中第 19 行中函数 SortAndMinimizationContexts 的排序可以采用自定义的度量方法，3.3.2小节中的度量方法均可以采用。考虑到应用场景，选取最长公共子序列距离进行相似性度量更为合理。

3.3.2　算法复杂度分析

本小节分析算法3的时间复杂度与空间复杂度。假设补全处的上文 given_context 长度为 n，编程语言中的合法 token 总数为 m。

首先分析算法3的空间复杂度。算法3主要的空间开销用于搜索结果 contexts 的存储上，即图3.2中的搜索树的存储，而相似性度量的空间复杂度为 $O(n^2)$。由于编程语言中的合法 token 总数有限，且在 n-gram 模型中，n 一般取值在 3 到 5 之间，从而空间复杂度为 $O(m^{1.5n})$。算法 3 的空间复杂度不高，对于一般配置的计算机，运行 API 补全程序时占用的内存资源较少。

Algorithm 3 SearchForContexts

Require: given_context (must be a token list)

Ensure: contexts ← ∅

```
for token : given_context do
    first_token ← token
    list_set ← {[first_token]}
    depth ← 0
    max_depth ← length(given_context) *1.5
    while depth < max_depth do
        new_list_set ← ∅
        for one_list : list_set do
            next_token_set ← InferNextTokens(one_list)
            for next_token : next_token_set do
                next_list ← one_list + next_token
                contexts = contexts ∪ new_list
            end for
        end for
        list_set ← new_list_set
        depth++
    end while
end for
contexts ← SortAndMinimizationContexts(contexts)
return contexts
```

下面分析算法3的时间复杂度。对于 n-gram 模型训练完成的情况下，算法3中

第 9 行的补全操作的时间复杂度为 $O(1)$。算法3在获取 contexts 阶段的时间复杂度等同于所有补全操作的时间开销，即为搜索树中边的数目 $O(m^{1.5n})$。在进行基于相似度的字符串排序时，首先需要计算最长公共子序列的长度，由于采用的 n-gram 模型的阶数在 3 到 5 之间，序列的长度在 5 到 8 之间，因此可以认为最长公共子序列的长度计算在常数时间内完成。之后按照最长公共子序列距离进行排序，若采用快速排序，时间复杂度为 $O(k \log k)$，其中 $k = O(m^{1.5n})$ 为 contexts 的长度。故算法3的时间复杂度为 $O(m^{1.5n} n \log m)$。考虑到 n 的取值在 3 到 5 之间，算法3的整体时间复杂度同样在可接受范围内。

3.3.3　算法局限

3.3.2小节分析了基于 BFS 的模糊搜索算法的时间复杂度和空间复杂度。上述算法采用离线形式的字符串模糊匹配算法，即不需要对上文字符串进行预处理。相比于深度学习算法，该算法的更加直接简单，不会出现前者在预处理阶段使用大量的时间开销。但是算法同样存在如下局限性，下面将做简要分析，并提出可能的改进策略。

如算法3.3.2小节中分析所示，算法的时间开销主要用于对搜索结果的排序上。然而在实际使用过程中，并不需要所有的搜索结果，一般情况下只需要使用与当前上文相似度最高的几个搜索结果。根据与上文相似度最高的匹配结果，可以得到可能的上文信息，而不需要对所有可能的上文进行补全。因此排序的时间可以大大减少，降至 $O(k)$ 的时间复杂度。

另一方面，如果采用最长公共子序列距离作为相似性度量指标，根据最长公共子序列算法的原理，在进行搜索空间拓展时可以将最长公共子序列的算法嵌入其中，避免之后对每个字符串单独计算。由于搜索的结果中很多字符串对之间具有前缀关系，因此单独计算最长公共子序列距离会出现冗余的计算过程。然而，这一改进策略也有本身的局限性。如果选择其他的字符串相似性度量指标，这样的算法优化可能起不到优化作用。对于其他的字符串相似性度量指标，可以根据其本身的特性尝试探索可能的优化策略，降低在相似度计算中的计算复杂度。

纵观全局，模糊搜索算法是一个较为广阔的研究领域，近年来不同领域内均提出了领域特定的模糊搜索算法，例如在程序分析中利用模糊搜索生成测试用例[30][31]。在代码补全问题中，还可以结合某些启发式的策略，比如遗传退火算法、蚁群算法等提高算法的性能，降低算法的复杂度[32]。在这里我们提出了一个将模糊搜索引入到 API 补全的框架，其中的搜索策略、字符串相似度度量方法均可以用不同的算法进行配置，不同的算法配置可以产生不同的补全效果。

3.4　本章小结

本章综述了在文本处理中常见的几种字符串匹配的算法和字符串相似性度量方法。传统的串匹配算法的目标为寻找完美匹配的子串或者连续子串，而在实际的 API 补全中，上文信息可能会因种种原因出现间断性的“污染”，而模糊搜索算法能够有效提高工具在不同补全上文场景中的鲁棒性。本章提出了基于 BFS 的模糊搜索算法，并将其应用于 API 补全中，通过对搜索结果的相似性度量进行排序，获取最可能的几种补全上文反馈到 API 补全模块中。

第 4 章　代码关联度挖掘原理与实现

代码关联度挖掘为 API 补全算法中的另一个重要模块。通过比较不同文件的代码关联度，n-gram 模型训练过程中可以参考得到的关联度进行更有效的训练。本章提出了一种简单有效地代码关联度挖掘的算法，能够以较低的时间复杂度将当前待补全文件与其他训练文件进行比较。同时算法流程具有很强的拓展性和可配置性，可以将其他的关联度挖掘方法整合进算法中，实现更加精准的关联度挖掘算法，提高 API 补全的准确性。

4.1　代码关联度挖掘的动机

本小节将分析经典缓存模型的局限性，并针对其局限设计一种基于代码关联度挖掘的改进版本，从而提高 API 补全的准确性。

4.1.1　经典缓存模型的局限

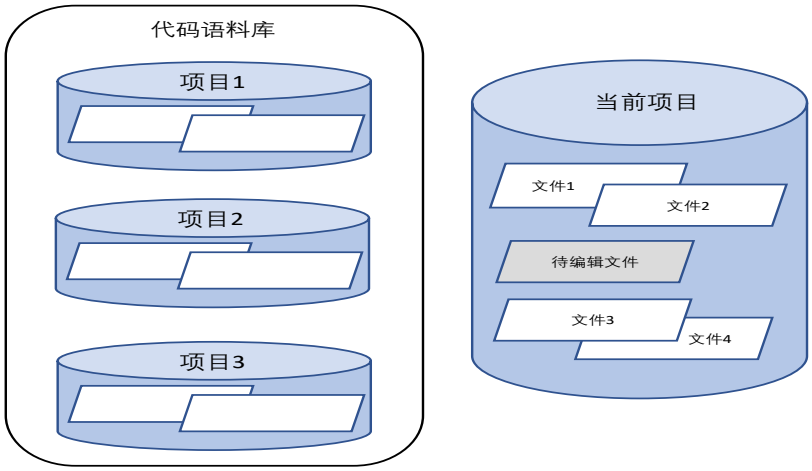

图 4.1　文件级别的代码划分

经典的缓存模型中，不同文件中的 n-gram 具有相同的权重，即假定不同部分的代码对当前编写的代码具有同等的重要程度和参考意义。显然这样的假定是不合理的。在实际编写程序时，与当前文件在同一工程内的代码对当前文件的编写具有更高的参考价值，因此缓存模型中将这一部分单独提取出来，与外界代码仓库中的代码部分分别处理。然而，即便是同一工程内的代码，它们与当前文件中的代码也具有不同的相关性，相关性高的部分对当前编写的代码具有更高的参考价

值。如图4.1所示，经典的缓存模型将代码分为代码语料库和当前项目两类，两个部分中的代码对当前待编辑文件中的代码补全有不同权重的影响。在本章中，基于上面的观察，尝试在传统的缓存模型的基础上做进一步改进，即将权重调整的粒度细化到文件级别，根据不同文件对当前待编辑文件的关联度赋予对应 n-gram 的权重。

4.1.2　基于代码关联度挖掘的缓存模型

与缓存模型引入缓存文件的概念类似，代码关联度挖掘将代码块做进一步细分。在缓存模型中，由缓存文件与语料库文件可分别训练出两个 n-gram 模型，补全时将两个模型进行加权拟合得到缓存模型中的概率估算值。换而言之，源代码被分为缓存文件和语料库文件两类，每一类中的 n-gram 序列在缓存模型中被赋予不同的权重。

在此基础之上，尝试将原来的缓存模型中代码划分的粒度加强，将划分的级别细化到文件。在基于代码关联度挖掘的模型中，不同的训练文件分别训练出 n-gram 模型，最终将所有的 n-gram 模型进行加权组合。加权组合的原则为：与当前编辑的文件关联度高的文件中的 n-gram 在模型中的比重更高。由此可以设计出基于代码关联度挖掘的缓存模型的通用框架。

显然，这一设计理念与缓存模型具有相似的设计原理，本质上是对不同空间上的代码在 n-gram 模型中权重的调整。类似的设计还包括对不同时间、不同开发人员编写的代码赋予不同的权重，从而实现对代码补全的个性化定制。这些设计在后续工作中将进行尝试，本文主要根据代码关联度将缓存模型进行优化。

4.2　代码关联度挖掘算法

本小节将实现4.1小节中提出的基于代码关联度挖掘的缓存模型。首先提出一般的基于代码关联度挖掘和缓存模型的 API 补全框架，然后将介绍一种简单、直接的代码关联度挖掘算法，并分析其算法复杂度和局限性。

4.2.1　算法框架

在这一部分中需要定义相应的代码特征提取函数、关联度度量函数以及根据关联度定义对应 n-gram 的训练权重。

特征提取函数的定义有多种方式。比如提取出文件中代码的抽象语法树，再抽取抽象语法树的特征形成特征向量。其他提取特征函数的方法还包括根据源文件的命名或者函数的命名提取特征。在进行关联度度量时，命名类似的两个文件

或者函数之间的关联度将被设置为较高的值。根据实际应用场景选择相应的特征函数时，由于文件数目较多，如果对整个语料库都采用基于抽象语法树特征抽取的方法，计算的复杂度较高。

通过度量特征向量之间的相似度比较两个文件的关联度。特征向量的相似度度量一般采用向量空间的余弦距离，即：

$$< a, b >= \arccos \frac{a \cdot b}{|a||b|} \tag{4-1}$$

其中 $a \cdot b$ 为两个特征向量 a 和 b 之间的点积，即内积。选取两个向量之间的夹角作为两者在向量空间中的距离。

N-gram 的训练权重的选取可以根据关联度度量函数的计算结果进行确定。由于通过 n-gram 模型进行 API 补全时选取的是联合概率最大的 API，因此每个 n-gram 的绝对权重并不重要，只需要选取一个单位权重即可。当计算出代码关联度后，将单位权重与关联度的乘积作为 n-gram 的实际权重即可。

通过提取代码特征、计算基于特征度量的代码关联度并赋予 n-gram 不同的训练权重，可以将传统的缓存模型推广成为基于代码关联度挖掘的缓存模型。由于其中提取代码特征等模块设计方法多样，因此这里只是提出一个基本的算法框架，应用时可以通过不同的应用场景和限定条件，选取更有效的方法对框架进行实例化。下面4.2.2小节将提出一种简单有效的算法实例。

4.2.2　算法设计

由于实际项目中函数数目较多且结构复杂，因此如果进行基于抽象语法树的代码特征提取所需的时间、空间开销很大。由于观察到具有良好编程规范的开发人员会将函数的功能通过函数的名称进行体现，因此具有相似函数名称的函数具有一定的关联度。故考虑将所有源文件中的函数名称按照大写字母等分隔符进行分割，并以文件为单位统计每个文件中函数名称构成的词典。通过比较两个文件中函数名称词典的重合程度进行关联度度量，这一度量方式类似于3.2.2小节中的雅卡尔指数。算法流程见算法4。

算法4中第 1 行和第 2 行的 ExtractFuncNameList 提取出源文件中所有的函数名。第 5 行到第 10 行根据提取出的函数名称进行字符串分割，生成由函数名称中的单词构成的词典 set1 和 set2。第 11 行到第 12 行根据词典 set1 和 set2 计算两个文件中代码的关联度。这里设置单位权重为 1，计算出的代码关联度即为实际的 n-gram 权重。

在后续流程中，算法4定义的 GetTrainingWeight 函数将计算结果反馈到 n-gram 模型训练模块。当处理 training_file 中的 n-gram 时，先通过算法4得到 n-gram 权重，之后进行 n-gram 统计时使用计算得到的权重。

Algorithm 4 GetTrainingWeight

Require: training_file, current_file

Ensure: training_weight

```
1: func_list1 ← ExtractFuncNameList(training_file)
2: func_list2 ← ExtractFuncNameList(current_file)
3: set1 ← ∅
4: set2 ← ∅
5: for funcname : func_list1 do
6:     set1 ← set1 ∪ SplitFuncName(funcname)
7: end for
8: for funcname : func_list2 do
9:     set2 ← set1 ∪ SplitFuncName(funcname)
10: end for
11: interset ← set1 ∩ set2
12: training_weight ← interset.size/(set1.size * set2.size)
13: return training_weight
```

4.2.3 算法复杂度分析

4.2.2小节针对4.2.1小节中的算法框架提出了一个具体的算法实例。算法4的时间开销主要集中在第 11 行两个集合的交和第 5 行到第 10 行的字典构建处。两个集合的交运算的算法时间复杂度为 $O(mn)$。如果采用集合作为数据结构进行实现，算法复杂度为 $O(min\{m,n\})$，其中 m 和 n 为得到的由函数名称中的单词构成的字典的大小。算法4中第 5 行到第 10 行之间的字典构建的时间复杂度为 $O(m+n)$。因此算法4的时间复杂度为 $O(m+n)$。对于其他的操作，比如函数名称的提取，这些通过构造抽象语法树可以直接得到，这里分析算法复杂度时不考虑抽象语法树构建的时间开销。

算法4的空间开销主要表现在字典的存储上，显然也是关于字典大小的线性复杂度。

综上所述，整个算法的时间复杂度和空间复杂度都是关于函数名称中单词个

数的线性规模，复杂度在可以接受的范围之内。如果要采用更加精细的代码关联度挖掘算法，可能会牺牲更多的时间开销或者空间开销。

4.2.4　算法局限

算法4是代码关联度算法的一个简单直接的实例。正如4.2.3小节中分析所示，算法的时间复杂度和空间复杂度都不高，但是其局限性比较明显。根据函数名称比较代码之间的相关性的一个重要前提是开发人员具有良好的函数命名习惯。然而实际情况中，很多编程人员的命名习惯多种多样，有的命名方式完全体现不了函数的功能和语义。在这种情形下，代码关联度挖掘的实例算法4就不能发挥实际作用。针对这一问题有很多替代的方案，比如采取不同的关联度挖掘的方法，下面将在4.3小节进行详细介绍。

4.3　代码关联度挖掘的其他实现

在4.2节中提出了一种简单的代码关联度挖掘算法。与之相关的代码挖掘算法还有很多，除了4.2节中提到的基于抽象语法树的特征抽取外，还有其他轻量级的代码关联度挖掘。但是所有方法从本质上看都是将源程序进行不同程度的解析，选取出特定的 token 进行统计形成特征，通过比较特征实现对源程序的关联度挖掘。此类方法大多需要对程序结构做解析，在处理小规模代码时预计效果较好。但是当代码规模增大时，抽象语法树的解析时间会相当大，因此在本工具中采用了上述4.2节中简单的方法进行处理。

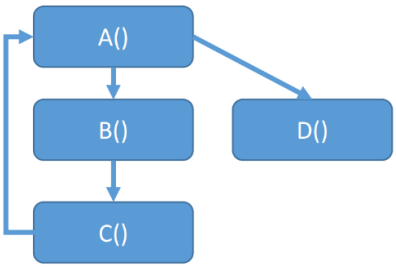

图 4.2　函数调用关系图

另外还有其他更加精细的代码挖掘算法。比如在图4.2中，函数 A、B、C、D 的调用关系如图所示。函数 B 与函数 D 都调用了函数 A，而函数 C 调用了函数 B，被函数 A 调用，因此倾向于相信函数 D 与函数 B 的关联度高于函数 D 与函数 C 的关联度。因此若正在编写函数 D，在进行 n-gram 模型训练时，函数 B 中的 n-gram 权重大于函数 C 中的 n-gram 权重。类似的方法还有，统计每个函数的调用者和被

调用者，通过比较两个函数的调用者列表和被调用者列表的重合程度对两个函数的相似性进行度量。这种挖掘算法的优势在于不需要生成完整的抽象语法树，只需要得到函数的调用关系图即可。

4.4　本章小结

本章首先介绍了代码关联度挖掘的动机，其本质是对不同文件中的 n-gram 训练权重的二次分配，然后给出了 API 补全中代码关联度挖掘的实现算法，并分析了算法的时间复杂度、空间复杂度以及其局限性，最后提出了几种其他形式的关联度挖掘的方案。

第 5 章　工具实现、应用与评估

第3章和第4章分别介绍了基于 BFS 的模糊搜索算法和代码关联度挖掘算法，两者为实现高精度、高鲁棒性的 API 补全提供了必要原料。本章介绍 API 补全工具 CRMAC 的架构以及在实际工程项目中的测试结果，并与传统的 n-gram 模型、缓存模型进行比较，分析模糊搜索算法和代码关联度挖掘算法在提升 API 补全性能中发挥的作用。

5.1　代码补全工具架构与实现

本小节将介绍 API 补全工具 CRMAC 的总体架构与补全流程。5.1.1节结合 CRMAC 工具的架构图介绍工具进行 API 补全时的主要步骤；5.1.2节将第3章和第4章中提出的模糊搜索算法以及代码关联度挖掘算法进行整合，将补全流程进行详细分析。

5.1.1　总体架构

代码补全工具的架构分成两个部分，分别为模型训练部分和补全部分，工具架构图如图5.1所示：

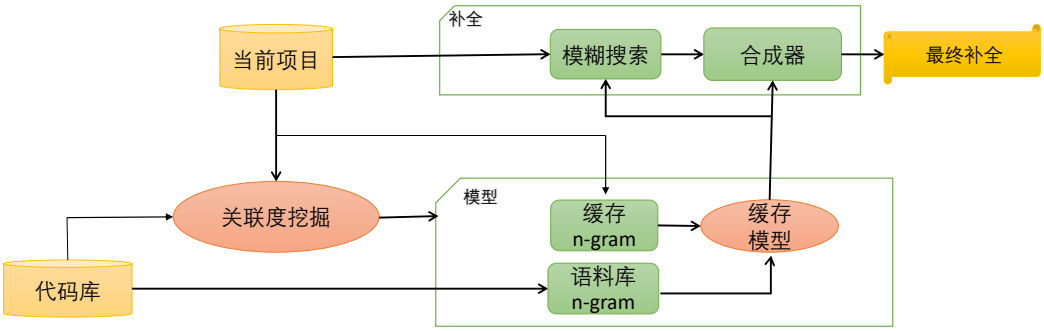

图 5.1　代码补全工具 CRMAC 架构图

当用户提交代码补全请求时，关联度挖掘模块首先获取当前编辑项目中的文件与其他文件（包括代码库中的文件和当前项目中的其他文件）之间的关联度，将计算结果输入到模型训练部分；在模型训练部分中，两个 n-gram 训练器基于关联度挖掘模块计算得到的关联度分别训练语料库和当前项目中的代码，得到两个 n-gram 模型，组合成为缓存模型。至此，模型训练模块的动作结束。

在补全部分，模糊搜索模块抓取当前项目的上文信息，调用基于 BFS 的模糊搜索算法，得到所有可能的代码上文作为合成器的输入。合成器逐条处理可能的上文，从模型训练的缓存模型中获取可能的 API，最终得到 API 列表返回给用户。用户选择后完成最终的补全。

5.1.2 补全流程

在工具的设计中，尝试将 CACHECA 中的缓存模型与模糊搜索、代码关联度挖掘相结合，在提高 API 补全的鲁棒性的基础上，充分利用不同文件与当前文件关联度的差异，赋予不同的文件中的 n-gram 不同的权重，提高 API 补全的准确率。其中缓存模型的 API 补全流程如下所示，根据代码语料库和当前工程训练得到缓存模型，获取当前上文信息后进行 API 推荐。

Algorithm 5 FindAPI

Require: $model_{corpus}$: n-gram model trained in corpus
$model_{cache}$: n-gram model trained in current project
γ: concentration factor
text: code context

Ensure: new_items

token_list1 ← getMapValue($model_{corpus}$, text)
token_list2 ← getMapValue($model_{cache}$, text)
candidates_map ← ∅
for token : token_list1 **do**
    prob ← GetProbFromCM($model_{corpus}$, $model_{cache}$, γ)
    items_map.put(token, prob)
end for
for token : token_list2 **do**
    prob ← GetProbFromCM($model_{corpus}$, $model_{cache}$, γ)
    items_map.put(token, prob)
end for
new_items ← ExtractItemsByProb(items_map)
return new_items

在工具 CRMAC 中，我们对缓存模型做进一步的改进。根据训练得到的缓存模型和可能的上文列表，工具通过调用算法5获得可能的 API 列表。算法5的

第 5 行和第 9 行计算出所有可能的 API 对应的概率，在第 12 行通过调用函数 ExtractItemsByProb 按照概率对得到的 API 进行排序。

工具的整体工作流程如算法6所示。当工具接收到一个 API 补全的请求时，工具抓取上文信息，获得一组可能的补全上文 contexts。代码关联度挖掘算法计算当前文件与语料库、当前项目中每个文件的关联度，得到训练的权重 weight，依次训练每个文件。最后根据 contexts 和训练得到的缓存模型，调用算法5中定义的 FindAPI 函数生成推荐的 API。

Algorithm 6 CompleteAPI

Require: context: current context
curfile: current file
corpfiles: the list of Java files in the corpus
projfiles: the list of Java files in the current project
γ: concentration factor

Ensure: token_list

1: $model_{corpus} \leftarrow \emptyset$
2: $model_{cache} \leftarrow \emptyset$
3: token_list $\leftarrow \emptyset$
4: **for** file : corpfiles **do**
5: weight ← GetTrainingWeight(file, curfile)
6: $model_{corpus} \leftarrow model_{corpus} \cup$ TrainNGram(file, weight)
7: **end for**
8: **for** file : projfiles **do**
9: weight ← GetTrainingWeight(file, curfile)
10: $model_{cache} \leftarrow model_{cache} \cup$ TrainNGram(file, weight)
11: **end for**
12: contexts ← SearchForContexts(context)
13: **for** text : contexts **do**
14: new_items ← FindAPI($model_{corpus}$, $model_{cache}$, γ, text)
15: token_list ← token_list ∪ new_items
16: **end for**
17: **return** token_list

5.2　实验设计与结果分析

本小节将对 CRMAC 工具的实验评估部分进行系统的分析。5.2.1节介绍评估实验的整体设计，包括测试用例的选取、测试方法的选择、评价指标的定义以及实验环境。5.2.2节将分析在不同的配置参数下 CRMAC 工具在不同测试集中的 API 补全准确率，通过比较 CRMAC 工具与传统的 n-gram 模型、传统的缓存模型的 API 补全准确率论证模糊搜索算法和代码关联度挖掘算法的有效性，并通过一个具体的补全结果对算法效果进行详细说明。

5.2.1　实验设计

在实验评估部分，选取了 6 个开源项目做 API 补全测试：Ant、Batik、Cassandra、Log、Maven 和 Xlan。为了模拟真实的 API 补全环境，假定项目开发到一半后发出代码补全请求，因此将每个开源项目分成两部分，一部分作为已经开发完成的部分，另一半作为 API 补全的测试文件，另外五个项目作为语料库。

为了比较不同的关注因子对 API 补全正确率的影响，实验中我们为关注因子 γ 设置了三种不同的取值，分别为 0、10、100。当关注因子取值为 0 时，缓存模型退化成为传统的 n-gram 模型。同时为了验证代码关联度挖掘模块能够有效地提高 API 补全的准确率，每次实验同时测试并比较了缓存模型与 CRMAC 工具的 API 补全的准确率。

在进行数据分析时，统计了正确的 API 在补全结果中前 3、5、10 位中的百分比，分别为 Top 3, Top 5 和 Top 10 的准确率。另外，我们参考其他相关的工作中的评价指标[6]，计算了每次实验的 MRR，即 API 在列表中排序序号的倒数平均值。

$$MRR = \frac{1}{|T|}\sum_{i=1}^{|T|}\frac{1}{rank_i} \tag{5-1}$$

其中 $|T|$ 为 API 补全请求总数，$rank_i$ 为第 i 次 API 补全中正确的 API 在推荐列表中的次序。当 MRR 接近 1 时，API 补全的准确率接近 100%。

所有的实验在如下硬件环境下进行：

- CPU Intel i5-4210M 2.6GHZ
- 128GB SSD
- 12GB RAM

5.2.2 结果分析

表 5.1　API 补全的准确率: $\gamma = 0$

Test Project	Top3		Top5		Top10		MRR	
	cache model	CRMAC	cache model	CRMAC	cache model	CRMAC	cache model	CRMAC
Ant 1	42.3%	47.8%	54.6%	59.4%	62.5%	69.1%	40.2%	45.8%
Ant 2	44.7%	49.0%	55.9%	60.1%	64.9%	71.1%	41.8%	46.9%
Batik 1	46.1%	51.0%	60.3%	65.9%	71.0%	72.9%	44.2%	49.0%
Batik 2	44.2%	49.9%	58.2%	63.3%	68.1%	70.1%	42.1%	47.5%
Cassandra 1	45.9%	48.8%	54.1%	58.9%	63.9%	69.7%	44.0%	47.2%
Cassandra 2	46.1%	49.0%	55.2%	59.7%	64.1%	70.2%	44.1%	47.7%
Log4J 1	43.5%	47.0%	51.0%	56.2%	57.4%	62.1%	40.3%	45.1%
Log4J 2	42.9%	45.9%	49.5%	54.0%	56.2%	62.0%	39.8%	43.9%
Maven2 1	50.1%	55.2%	56.1%	62.0%	68.1%	71.0%	47.3%	53.0%
Maven2 2	52.0%	56.0%	57.9%	63.6%	70.5%	71.2%	49.7%	53.6%
Xalan 1	43.8%	50.5%	51.7%	62.4%	65.7%	68.1%	40.5%	47.6%
Xalan 2	43.7%	51.7%	50.1%	63.6%	64.9%	69.9%	41.8%	48.7%

表 5.2　API 补全的准确率: $\gamma = 10$

Test Project	Top3		Top5		Top10		MRR	
	cache model	CRMAC	cache model	CRMAC	cache model	CRMAC	cache model	CRMAC
Ant 1	50.2%	54.3%	60.5%	64.6%	67.1%	73.2%	48.0%	52.1%
Ant 2	51.8%	54.7%	62.5%	65.4%	66.8%	74.3%	48.2%	52.3%
Batik 1	49.8%	53.1%	65.7%	69.3%	73.6%	76.7%	47.3%	51.2%
Batik 2	48.4%	52.3%	63.1%	68.1%	71.7%	74.2%	46.8%	50.5%
Cassandra 1	50.7%	56.8%	60.2%	66.4%	69.8%	75.5%	48.1%	54.9%
Cassandra 2	51.2%	57.5%	62.2%	69.8%	70.8%	77.5%	48.9%	55.7%
Log4J 1	45.8%	50.1%	54.9%	59.4%	60.1%	68.4%	44.1%	47.4%
Log4J 2	43.4%	49.8%	51.6%	58.3%	59.5%	67.6%	41.8%	46.9%
Maven2 1	53.5%	59.6%	58.7%	65.9%	72.6%	77.3%	51.0%	57.3%
Maven2 2	54.9%	62.3%	59.1%	67.0%	73.4%	78.3%	52.3%	59.0%
Xalan 1	46.9%	54.7%	56.1%	66.0%	71.4%	74.9%	45.5%	52.8%
Xalan 2	45.0%	53.9%	54.9%	68.9%	69.8%	72.0%	44.3%	51.9%

表 5.3　API 补全的准确率: $\gamma = 100$

Test Project	Top3		Top5		Top10		MRR	
	cache model	CRMAC	cache model	CRMAC	cache model	CRMAC	cache model	CRMAC
Ant 1	51.7%	53.2%	58.9%	65.7%	66.9%	74.3%	47.3%	50.2%
Ant 2	50.1%	55.8%	63.1%	66.2%	68.6%	75.1%	47.1%	52.0%
Batik 1	48.1%	52.9%	66.2%	70.4%	72.5%	75.9%	45.2%	49.8%
Batik 2	47.6%	51.2%	65.8%	69.6%	72.9%	74.9%	43.8%	48.2%
Cassandra 1	51.9%	57.1%	61.7%	67.0%	70.3%	74.2%	48.0%	54.9%
Cassandra 2	52.4%	56.8%	62.9%	68.9%	71.1%	76.7%	49.8%	53.6%
Log4J 1	44.9%	51.2%	55.6%	60.1%	62.4%	68.0%	42.1%	48.0%
Log4J 2	43.7%	50.2%	53.5%	58.9%	61.0%	69.1%	41.5%	47.8%
Maven2 1	53.7%	59.6%	58.4%	66.4%	72.9%	76.8%	51.0%	57.3%
Maven2 2	54.5%	61.7%	60.1%	68.2%	73.3%	77.9%	52.3%	58.8%
Xalan 1	47.3%	55.1%	55.2%	66.7%	71.9%	75.1%	44.1%	52.8%
Xalan 2	45.8%	54.9%	55.5%	67.5%	70.8%	74.2%	43.6%	52.1%

选取三个不同的关注因子，分别为 0，10, 100。实验结果分别见表5.1、5.2、5.3。在第一组实验中，设置关注因子为 0，此时缓存模型和 CRMAC 中的缓存 n-gram 部分失效。由表5.1中的数据可知，CRMAC 的 Top 10 补全的正确率比未经过代码关联度挖掘的缓存模型高，且不同项目之间性能提高的程度具有明显的差异性。比如在 Ant、Cassandra、Log4J 和 Xalan 这四个项目中，代码关联度挖掘对 API 补全正确率的提高效果明显，能提高 5% 到 7%。然而在 Bantik 和 Maven 这两个项目上效果并不显著，只提高 1% 左右。这种差异的原因在于不同项目之间的源代码在本工作中定义的关联度意义下差异程度不同。若训练集中其他文件与当前文件的关联度相差较小，在 n-gram 训练时，各个文件中的 n-gram 被赋予近似相等的权重，这与传统的缓存模型几乎完全相同，因此提升效果不明显。

然而另一方面，在另四个项目中，关联度挖掘有效地提高了代码补全的准确性。如图5.2所示，LineContains.java 文件与 LineContainsRegExp.java 中的函数名称相似程度高。

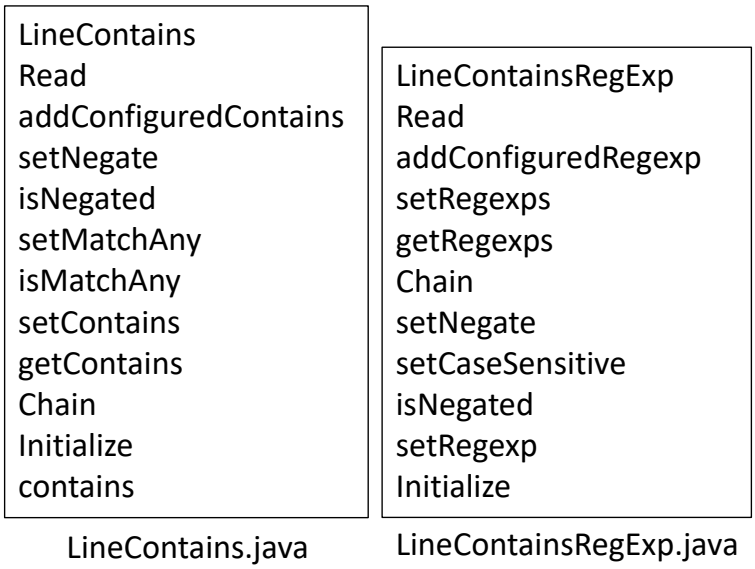

图 5.2　两个 Java 文件的方法列表

```
if (line.length() == 1) {
   line = null;
} else {
   line = line.substring(1);
}
```

```
int ch = line.charAt(linePos);
linePos++;
if (linePos == line.length()) {
   line = null;
}
```

图 5.3　两个代码片段

比如在 LineContains.java 文件中定义了函数 addConfiguredContains，在 LineContainsRegExp.java 文件中定义了函数 addConfiguredRegexp，经过关联度挖掘模块处理时分别被处理成三个 token，其中都包含 add 和 Configured 这两个 token。因此在度量代码关联度时，这两个文件被关联度挖掘模块判断为关联度高，因此补全其中一个文件时，另一个文件中的 n-gram 训练权重要大于其他文件。两个 java 文件中都包含图5.3中左侧的代码片段，HeadContains.java 中包含右侧的代码片段。由于 LineContainsRegExp.java 与 LineContains.java 的关联度更高，因此 line=line.substring(1) 赋予了更高的权重，从而在进行补全时，CRMAC 给出了正确的补全结果 substring，而不是缓存模型给出的 charAt。根据上述分析，当文件之间的差异程度相近时，代码关联度挖掘对 API 补全的准确率的提升效果不明显。

在后两组实验中，设置关注因子分别为 10 和 100。比较第一组和第二组实验数据能够发现缓存机制的引入能够很有效地提升 API 补全的准确性，Top 3 准确率平均提高 4.79%，Top 10 准确率平均提高 5.28%。然而，当关注因子达到某个值后，继续增大关注因子对补全结果的影响程度较小。通过比较第二组和第三组实验可以发现，将关注因子从 10 提高到 100 并没有从统计的意义上提升补全的正确率，原因在于较小的关注因子已能够有效地捕捉到缓存文件中的 n-gram。

5.3　本章小结

本章首先介绍了代码补全工具 CRMAC 的架构和工作流程，接着介绍了 CRMAC 的测试环境和测试用例配置，最后分析多组测试数据结果，论证了本文的方法的有效性。实验结果表明，模糊搜索和代码关联度挖掘对 API 补全工具 CRMAC 的 API 补全正确率的提升和鲁棒性的提高都起到关键作用。在选取合适的实验参数的情况下，将现有典型工具的 Top 3 准确率平均提高 4.79%，Top 10 准确率平均提高 5.28%，平均 Top 10 准确率达到 74.4%。

第 6 章　总结展望

本章对整个研究工作进行系统地总结，并探讨其中具有研究价值的重要问题，并对今后的研究方向进行展望。

6.1　工作总结

代码自动补全是编程语言中的一个传统且富有挑战的问题，近 20 年来各种形式的代码补全工具层出不穷，其中最有代表性的为 Eclipse 和 IntelliJ IDEA 中 API 补全插件，但是它们各自存在一些局限。近年来，基于自然语言处理的代码补全成为代码补全领域常用的研究方式。考虑到编程语言与自然语言具有的共性，两者同时具有局部重复性，研究人员使用各种统计语言模型对语料库和待补全的代码进行建模处理。但是如何捕捉项目自身的特异性、提高补全的鲁棒性一直是待探索的问题。

本文第2章在传统的 n-gram 模型的基础上，引入了缓存机制，将传统的 n-gram 模型拆分成两个部分，分别对应外部语料库与当前项目训练得到的两个 n-gram 模型。同时为了应对 n-gram 模型的数据稀疏现象，在工具开发的过程中，我们引入了 Lidstone 平滑处理概率计算中的每个乘积项。

本文第3章提出了基于 BFS 的模糊搜索算法。在实际 API 补全场景中，常常出现当前补全上文在语料库和当前项目其他位置并未出现过的情况，从而经典的 n-gram 模型无法进行 API 推荐。为了解决这个问题，可以尝试搜索所有与当前上文类似的 n-gram。这里引入了一些启发式搜索的想法，比如通过 BFS 得到所有与当前上文有公共子序列的 n-gram，通过最长公共子序列的长度计算当前上文与该 n-gram 之间的相似度，选出最有可能替代当前上文的 n-gram 进行 API 补全。这样的改进有很重要的意义，因为在实际编码中，开发人员往往会因为疏忽输错某些字符导致无法整体匹配上正确的 n-gram。模糊搜索算法就可以较为精准地推荐出可能的 API。

本文第4章提出了代码关联度挖掘算法，依次阐述了算法设计的动机、具体过程以及其他可能的实现。本质上说，代码关联度挖掘是对缓存模型的进一步推广：缓存模型将原本为一体的代码库与当前项目拆分成两个部分分别训练，等同于赋予两个部分中的 n-gram 以不同的训练权重；而代码关联度挖掘是将拆分的层次细化到文件级别，通过比较当前待补全的文件与其他每个文件的相似程度赋予不同文件内的 n-gram 不同的训练权重。代码关联度挖掘是一个通用的算法框架。度量

两个源码之间的关联度的算法有很多，可以根据实际问题的需要分别做个性化设计。在第四章后半部分提出了多种度量算法，可以选择合适的度量策略替代补全框架中的度量模块。

第5章将前三章设计的算法流程实现成完整的 API 补全工具 CRMAC。首先介绍了 CRMAC 的体系架构以及 API 补全的工作流程，接着详细介绍了实验的设计、环境以及实验结果，并分析了不同参数下的结果差异，阐述了引起准确率差异的原因。实验数据表明第3章和第4章提出的模糊搜索算法和代码关联度挖掘算法在提升 API 补全正确性上的贡献，并为之后的研究探索出新的方向。

6.2　研究展望

本文提出了基于统计语言模型的 API 补全算法，并实现了原型工具。其中存在很多有意义、值得深入研究的问题。在本工作中，选取 n-gram 模型作为基本的统计语言模型对程序语言的统计规律进行刻画，此外仍然存在其他具有很强表现能力的语言模型。第4章提出了一个代码关联度挖掘的框架，除了代码关联度挖掘，其他代码挖掘问题也同样具有很强的研究价值。模糊搜索为 API 补全工具增强了鲁棒性，也是一个值得深入探索的问题。由于本工作研究对象为 API 补全，更广泛的表达式补全依赖于中间语言表示的生成，因此中间语言表示成为了代码补全的一个核心问题。下面将对这些研究问题进行详细阐述。

6.2.1　语言模型

本文通过 n-gram 模型对代码进行建模，能够捕捉相邻的 token 之间的相关性。但是无论是自然语言还是编程语言，跨空间的 token 之间仍然存在依赖性，因此用 n-gram 模型不能捕捉这一特性。

另一方面，尽管 n-gram 模型中采用了回退和平滑方法保证了概率计算中每一项均非 0，但是当出现一个之前没有出现过的 token 时，模型对其赋予一个很小的值，从而整体 n-gram 出现的联合密度概率会非常低。这严重限制了 n-gram 模型的应用。尽管在第3章中提出了模糊搜索的算法找出所有类似的上文信息以尽可能地规避这个问题，但是这样的方法并没有从根本上解决这一问题。问题的根源在于 n-gram 模型对句子出现概率的估计机制具有瑕疵，因此要想更有效解决这一问题需要更换语言模型。以 LSTM 为代表的递归神经网络模型更能有效应对这一问题。

图6.1为神经网络语言模型结构图。训练阶段中，根据极大似然原理估计出模

型的参数。似然函数如公式 (6-1) 所示:

$$L = \frac{1}{T}\sum_{t} \log f(w_t, w_{t-1},, w_{t-n+1}; \theta) + R(\theta) \tag{6-1}$$

其中 $R(\theta)$ 为模型中的正则化项，又称为惩罚项。与 n-gram 模型不同，神经网络模型不需要加入回退操作或平滑操作解决取值为 0 的概率项的问题。对于未出现过的词的特征向量，神经网络模型根据其在语料库中的分布进行初始化。

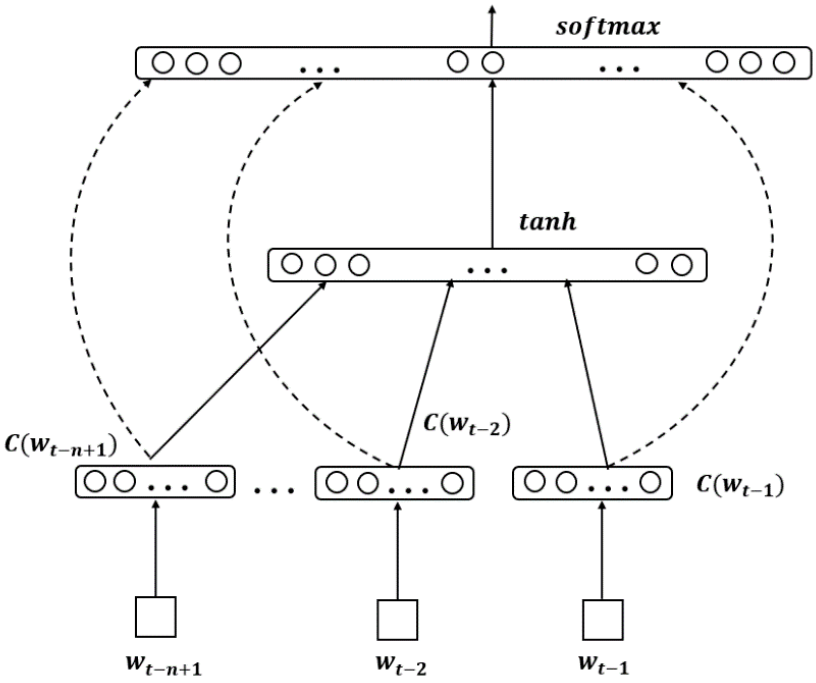

图 6.1　神经网络语言模型

其他的语言模型比如 skip-gram 模型、连续词袋模型等也具有很强的表现能力[33]。在实际使用时需要结合任务需要选择合适的语言模型，或者对已有的语言模型进行修改，使其更加适应当前任务。比如在代码补全工作中，需要修改模型结构或将其与其他模型组合[34][35]。

6.2.2　代码挖掘

代码挖掘除了在代码补全领域有重要应用，同样在代码克隆分析、漏洞模式挖掘[36][37] 等领域也扮演重要的角色。另一方面，代码挖掘的方法也十分丰富，本文第4章简略介绍了基于抽象语法树和函数调用图的代码挖掘算法，除此之外还有很多值得尝试的研究思路，包括将经典数据挖掘的算法应用于代码挖掘[38]。

比如在代码克隆领域，常常需要对代码的来源进行跟踪，即判断某段代码是否是通过复制粘贴后进行简单的重命名后得到的。这一技术可以在 bug 追踪与代码责任分配问题上起到关键作用。在图6.2中的两个代码片段中，fun2.java 是通过将 fun1.java 中的变量进行重命名后变换某些语句的顺序后得到，因此在通过代码挖掘分析过程后，可以将这两段代码视为同一份代码。当检测到 fun1.java 中有错误时，在一定程度上可以断言 fun2.java 中也存在类似的问题。

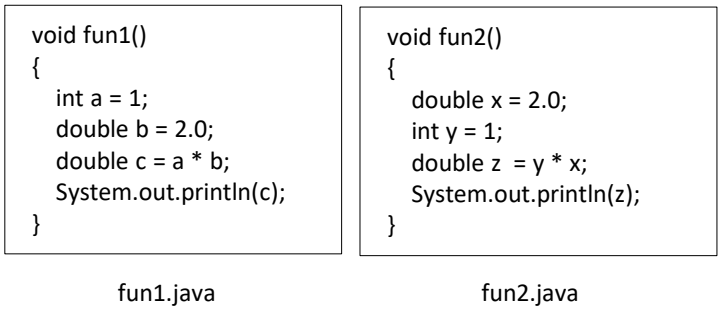

图 6.2　两个等价的代码片段

同样在程序分析等软件漏洞检测领域，代码挖掘同样扮演着重要角色。Fabian Yamaguchi 在他的博士期间发表了大量关于基于代码属性图的漏洞检测的相关工作[39]。他试图将传统的几种经典图表示合并成一种更加有效的表示，在合成后的图上做挖掘，建立起程序漏洞与图结构中的映射关系[40]。这一研究思路具有很好的推广价值。在代码补全和克隆分析领域，基于程序图表示的概率模型为提高补全、分析效果创造了又一种可能。

6.2.3　模糊搜索

代码模糊匹配是一个值得深入探索的领域。在代码补全中，代码库中可能不存在与当前上文完全字面匹配的序列，但是可能存在结构上完全同构的序列。因此，如何查找出在结构上相似或者相同的代码变为代码补全领域中的关键问题。本文采用了基于 BFS 的模糊搜索算法，但是在处理同构的代码的情形中，本文中提出的算法仍然具有较大的局限性。

模糊搜索在很多情况下需要与一些代码挖掘的算法组合使用，比如通过挖掘算法提取出的程序特征或者训练出的模型在代码仓库中选择出与当前代码序列类似的代码片段组成候选集合。启发式策略在模糊搜索中也能起到很好地效果，比如遗传算法等可以对代码片段进行评估，选择评分较高的代码片段构成候选集合[41]。本工作中基于 BFS 的模糊搜索是一种较为简单的版本，在处理实际代码中作用相

对有限，因此为了保证工具的实用性，对模糊搜索进行深入研究具有一定的必要性。

6.2.4　中间语言表示

本文探索了一种 API 自动补全算法，然而在实际的编程开发过程中，集成环境仅仅具有 API 自动补全功能是不够的，在很多情况下需要具有表达式自动补全的功能。由于表达式本身的复杂性与灵活性，n-gram 模型难以直接对其统计规律进行刻画，因此需要将代码转化成中间语言表示后进行序列化处理，再通过 n-gram 模型等统计语言模型进行中间语言补全，最后将补全结果转化成源程序。中间语言表示的生成大多数依赖于 AST 树，即抽象语法树。

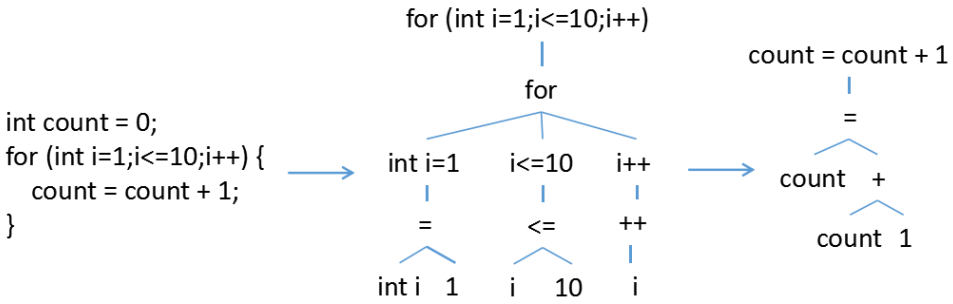

图 6.3　抽象语法树实例

比如图6.3的例子中，若要生成源代码片段的中间语言，需要按照某种方式对其对应的抽象语法树进行遍历，筛选出其中的重要的信息，过滤掉无用的程序 token，生成信息密度更大的中间语言。

两种相同的代码	先序遍历	中序遍历	后序遍历
A(B());	AB	AB	BA
int b = B(); A(b);	BA	BA	BA

不同的序列　　相同的序列

图 6.4　遍历方式比较

在中间语言生成的过程中，不可避免需要对原来程序的抽象语法树进行遍历，然而遍历的方式需要根据实际任务的需要进行精心设计。比如在图6.4中的函数调

用，实际上存在多种相同语义的源代码表示。若根据抽象语法树进行先序遍历和中序遍历，得到的结果不同。然而中间语言与程序语义之间应该存在一一对应的关系，即不同的代码，只要语义相同，生成的中间语言应该是相同的，这是保证代码补全正确性的必要条件。因此应该选取后序遍历作为基本的遍历策略。对于不同的代码补全任务，仍然需要对后序遍历做一些修改，保证能够用尽可能短的中间语言提取出所有的有效的程序语义信息。开源社区有丰富的工具为中间语言生成提供支持，其中最有代表性且广泛使用的工具 ANTLR 支持多语言的词法分析和语法分析[42]。

6.2.5　代码补全中的其他方法

以上讨论的代码补全，不论是 API 补全还是表达式补全，参考的数据均为历史代码[43][44]。此外程序的输入也可以为代码补全提供有用信息[45]。同时，源文件中除了包括源代码，还包括编程人员编写的注释。对于具有良好编程习惯的开发人员编写的代码，其中的注释能够提供很多有效的信息，为源代码的补全提供了很多辅助信息[46]。此外，程序中的噪音数据也可以为代码补全提供有益的信息[47]。

在编程语言的补全问题中，一般采用的模型为 sequence to token 的模型，即根据一段序列推断出一个 token，比如 API 等。然而如果结合代码周边的注释环境，采用自然语言模型中的序列到序列的语言模型能够更好地完成代码补全的任务[48][49]。类似的做法可以参考机器翻译中的技术[50]。

以上是对代码自动补全领域中相关技术问题的讨论。代码自动补全系统作为一个经典的推荐系统，其本质在于利用已有的信息对未来代码的内容进行预测，从而在一定程度上控制被开发的软件系统的行为。正如上述章节所述，这里的信息不仅仅包括当前项目的源代码，还包括注释等其他代码信息，当然还包括代码仓库中的代码以及开发、交互日志等代码产品等等。

作为编程语言方向的一个经典问题，代码自动补全与其他的编程语言方向研究问题一样，其核心在于理解代码和预知代码：理解代码的语义、预知代码的内容和行为。这与其他的编程语言问题，比如程序分析、程序验证、程序综合等都具有相同的特征。而代码补全也不仅仅停留在对开发人员的辅助编程这一层面。对于程序分析中检测出的程序漏洞，可以通过代码补全的手段，对其中某些易于修复的缺陷进行缺陷定位并自动修复[51][52]。因此这一研究问题具有很高的研究价值和应用前景。

6.3　本章小结

本章总结了基于 n-gram 模型的 API 补全的工作，分析了工作的核心点，并深入探讨了其中具有深入研究价值的若干方向。这些方法和方向为之后进行表达式补全提供了一定的参考。

参考文献

[1] Prescher D. A tutorial on the expectation-maximization algorithm including maximum-likelihood estimation and EM training of probabilistic context-free grammars[J/OL]. CoRR, 2004, abs/cs/0412015. http://arxiv.org/abs/cs/0412015.

[2] V.Raychev, P.Bielik, M.T.Vechev. Probabilistic model for code with decision trees[C/OL]// Visser E, Smaragdakis Y. OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016. ACM, 2016: 731–747. http://doi.acm.org/10.1145/2983990.2984041.

[3] Balog M, Gaunt A L, Brockschmidt M, et al. Deepcoder: Learning to write programs[C/OL]//5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. 2017. https://openreview.net/forum?id=ByldLrqlx.

[4] Dam H K, Tran T, Pham T. A deep language model for software code[J/OL]. CoRR, 2016, abs/1608.02715. http://arxiv.org/abs/1608.02715.

[5] Tu Z, Su Z, Devanbu P T. On the localness of software[C/OL]//Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014. 2014: 269–280. https://doi.org/10.1145/2635868.2635875.

[6] Franks C, Tu Z, Devanbu P T, et al. CACHECA: A cache language model based code suggestion tool[C/OL]//37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2. 2015: 705–708. https://doi.org/10.1109/ICSE.2015.228.

[7] A.Hindle, E.T.Barr, Z.Su, et al. On the naturalness of software[C/OL]//ICSE 2012, June 2-9, 2012, Zurich, Switzerland. 2012: 837–847. http://dx.doi.org/10.1109/ICSE.2012.6227135.

[8] V.Raychev, M.T.Vechev, E.Yahav. Code completion with statistical language models[C/OL]// PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014. 2014: 44. http://doi.acm.org/10.1145/2594291.2594321.

[9] Mooty M, Faulring A, Stylos J, et al. Calcite: Completing code completion for constructors using crowds[C/OL]//IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2010, Leganés-Madrid, Spain, 21-25 September 2010, Proceedings. 2010: 15–22. https://doi.org/10.1109/VLHCC.2010.12.

[10] Yang Y, Jiang Y, Gu M, et al. A language model for statements of software code[C/OL]// Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017. 2017: 682–687. https://doi.org/10.1109/ASE.2017.8115678.

[11] Zhong H, Xie T, Zhang L, et al. MAPO: mining and recommending API usage patterns[C/OL]// ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings. 2009: 318–343. https://doi.org/10.1007/978-3-642-03013-0_15.

[12] Zhang C, Yang J, Zhang Y, et al. Automatic parameter recommendation for practical API usage [C/OL]//34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland. 2012: 826–836. https://doi.org/10.1109/ICSE.2012.6227136.

[13] Wang X, McCallum A, Wei X. Topical n-grams: Phrase and topic discovery, with an application to information retrieval[C/OL]//Proceedings of the 7th IEEE International Conference on Data Mining (ICDM 2007), October 28-31, 2007, Omaha, Nebraska, USA. 2007: 697–702. https://doi.org/10.1109/ICDM.2007.86.

[14] C W B, M T. N-gram-based text categorization[J]. Ann Arbor MI, 1994, 48113(2): 161–175.

[15] Siddharthan A. Christopher d. manning and hinrich schutze. *Foundations of Statistical Natural Language Processing*. MIT press, 2000. ISBN 0-262-13360-1, 620 pp. $64.95/£44.95 (cloth)[J/OL]. Natural Language Engineering, 2002, 8(1): 91–92. https://doi.org/10.1017/S1351324902212851.

[16] P.F.Brown, P.V.deSouza, R.L.Mercer, et al. Class-based n-gram models of natural language[J]. Computational Linguistics, 16(2), 1992: 467–479.

[17] Bielik P, Raychev V, Vechev M T. PHOG: probabilistic model for code[C/OL]//Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016. 2016: 2933–2942. http://jmlr.org/proceedings/papers/v48/bielik16.html.

[18] Allamanis M, Tarlow D, Gordon A D, et al. Bimodal modelling of source code and natural language[C/OL]//Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015. 2015: 2123–2132. http://jmlr.org/proceedings/papers/v37/allamanis15.html.

[19] Stolcke A. SRILM - an extensible language modeling toolkit[C/OL]//7th International Conference on Spoken Language Processing, ICSLP2002 - INTERSPEECH 2002, Denver, Colorado, USA, September 16-20, 2002. 2002. http://www.isca-speech.org/archive/icslp_2002/i02_0901.html.

[20] Gabel M, Su Z. A study of the uniqueness of source code[C/OL]//Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010. 2010: 147–156. https://doi.org/10.1145/1882291.1882315.

[21] Kuhn R, de Mori R. A cache-based natural language model for speech recognition[J/OL]. IEEE Trans. Pattern Anal. Mach. Intell., 1990, 12(6): 570–583. https://doi.org/10.1109/34.56193.

[22] Wikipedia. AC automaton algorithm[EB/OL]. https://en.wikipedia.org/wiki/Aho%E2%80%93Corasick_algorithm.

[23] Wikipedia. Boyer-moore algorithm[EB/OL]. https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string-search_algorithm.

[24] Wikipedia. Hamming distance[EB/OL]. https://en.wikipedia.org/wiki/Hamming_distance.

[25] Wikipedia. Jaccard index[EB/OL]. https://en.wikipedia.org/wiki/Jaccard_index.

[26] Wikipedia. Simple matching coefficient[EB/OL]. https://en.wikipedia.org/wiki/Simple_matching_coefficient.

[27] Wikipedia. Lee distance[EB/OL]. https://en.wikipedia.org/wiki/Lee_distance.

[28] Wikipedia. Edit distance[EB/OL]. https://en.wikipedia.org/wiki/Edit_distance.

[29] Wikipedia. Longest common subsequence problem[EB/OL]. https://en.wikipedia.org/wiki/Longest_common_subsequence_problem.

[30] Gao J, Yang X, Fu Y, et al. Vulseeker-pro: enhanced semantic learning based binary vulnerability seeker with emulation[C/OL]//Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018. 2018: 803–808. https://doi.org/10.1145/3236024.3275524.

[31] Gao J, Yang X, Fu Y, et al. Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary[C/OL]//Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. 2018: 896–899. https://doi.org/10.1145/3238147.3240480.

[32] Holmes R, Walker R J, Murphy G C. Approximate structural context matching: An approach to recommend relevant examples[J/OL]. IEEE Trans. Software Eng., 2006, 32(12): 952–970. https://doi.org/10.1109/TSE.2006.117.

[33] M.Andriy, Whye Y. A fast and simple algorithm for training neural probabilistic language models[J]. Computer Science, 2012: 1751–1758.

[34] A.T.Nguyen, T.N.Nguyen. Graph-based statistical language model for code[C/OL]//ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1. 2015: 858–868. http://dx.doi.org/10.1109/ICSE.2015.336.

[35] T.T.Nguyen, A.T.Nguyen, H.A.Nguyen, et al. A statistical semantic language model for source code[C/OL]//ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013. 2013: 532–542. http://doi.acm.org/10.1145/2491411.2491458.

[36] Livshits V B, Zimmermann T. Dynamine: finding common error patterns by mining software revision histories[C/OL]//Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005. 2005: 296–305. https://doi.org/10.1145/1081706.1081754.

[37] Gabel M, Su Z. Javert: fully automatic mining of general temporal properties from dynamic traces[C/OL]//Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008. 2008: 339–349. https://doi.org/10.1145/1453101.1453150.

[38] Xie T, Thummalapenta S, Lo D, et al. Data mining for software engineering[J/OL]. IEEE Computer, 2009, 42(8): 55–62. https://doi.org/10.1109/MC.2009.256.

[39] Yamaguchi F. Pattern-based vulnerability discovery[D/OL]. University of Göttingen, 2015. http://nbn-resolving.de/urn:nbn:de:gbv:7-11858/00-1735-0000-0023-9682-0-2.

[40] Yamaguchi F, Wressnegger C, Gascon H, et al. Chucky: exposing missing checks in source code for vulnerability discovery[C/OL]//2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013. 2013: 499–510. https://doi.org/10.1145/2508859.2516665.

[41] W.Weimer, T.Nguyen. Automatically finding patches using genetic programming[C/OL]//ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings. 2009: 364–374. http://dx.doi.org/10.1109/ICSE.2009.5070536.

[42] Parr T J, Quong R W. ANTLR: A predicated- *LL(k)* parser generator[J/OL]. Softw., Pract. Exper., 1995, 25(7): 789–810. https://doi.org/10.1002/spe.4380250705.

[43] R.Robbes, M.Lanza. Improving code completion with program history[J]. Automated Software Engineering 17(2), 2010: 257–277.

[44] White M, Vendome C, Vásquez M L, et al. Toward deep learning software repositories[C/OL]//12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015. 2015: 334–345. https://doi.org/10.1109/MSR.2015.38.

[45] Han S, Wallace D R, Miller R C. Code completion of multiple keywords from abbreviated input[J/OL]. Autom. Softw. Eng., 2011, 18(3-4): 363–398. https://doi.org/10.1007/s10515-011-0083-2.

[46] Movshovitz-Attias D, Cohen W W. Natural language models for predicting programming comments[C/OL]//Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4-9 August 2013, Sofia, Bulgaria, Volume 2: Short Papers. 2013: 35–40. http://aclweb.org/anthology/P/P13/P13-2007.pdf.

[47] V.Raychev, P.Bielik, M.T.Vechev, et al. Learning programs from noisy data[C/OL]//POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. 2016: 761–774. http://doi.acm.org/10.1145/2837614.2837671.

[48] Long F, Rinard M. Automatic patch generation by learning correct code[C/OL]//Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. 2016: 298–312. http://doi.acm.org/10.1145/2837614.2837617.

[49] Bruch M, Monperrus M, Mezini M. Learning from examples to improve code completion systems[C/OL]//Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009. 2009: 213–222. https://doi.org/10.1145/1595696.1595728.

[50] Osborne M. Statistical machine translation[M/OL]//Encyclopedia of Machine Learning and Data Mining. 2017: 1173–1177. https://doi.org/10.1007/978-1-4899-7687-1_783

[51] Monperrus M. A critical review of "automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair [C/OL]//36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014. 2014: 234–242. https://doi.org/10.1145/2568225.2568324.

[52] Le Goues C, Forrest S, Weimer W. Current challenges in automatic software repair[J/OL]. Software Quality Journal, 2013, 21(3): 421–443. https://doi.org/10.1007/s11219-013-9208-0.

致 谢

清华的七年时光进入尾声，然而一切仿佛都发生在昨天。还记得本科进入清华的第一堂课上，陈吉宁校长对我们“立德、立言、立行”的教诲；还记得本科最后一次班会课上，张奥千辅导员给我们的寄语“做个好学生、好家人、好公民”；还记得姜宇老师、Min Zhou 老师在硕士三年来给我的无私帮助；还记得数学系的好友刘杨帮我解决的一个个数学上的问题；还记得清华学堂的桌椅、大礼堂的穹顶和六教答疑坊的点点灯光。

对我而言，清华的学习生活充满了坎坷和曲折，正是有了很多好心的老师和同学的帮助，我才得以走到今天。站在学堂路的终点，回望漫长的七年时光，内心只有无尽的感激。在我的学习生活中，父母一直是我无穷的动力，面对人生的重大抉择时一直尊重我的选择，始终相信我的能力和眼光。

三年的硕士生涯中，实验室的老师为我创造了很好的研究环境，即便不是直接指导我的老师也给予我很多帮助。我的导师姜宇老师为我提供了宽松的研究环境，使我能够短时间内从困境中走出来，重新回到研究的正途。罗贵明老师在我陷入低谷时鼓励我勇敢探索，给我以勇气。Min Zhou 老师和张荷花老师也为我提供了很多帮助，感谢他们的无私付出。

每天接触最多的实验室同学更是我砥砺前行的原动力。看到优秀的学长，我渴望自己之后能够像他们一样优秀；看到积极努力的学弟，我仿佛看到了曾经的自己而不敢懈怠。感谢一直以来为我提供帮助的谷哥、兴哥和小杨哥，还有相处仅一年的学弟妹们。还有很多现在仍在联系的高中好友，有的就在这片园子里，有的在地球的另一端，但都在我最需要帮助的时候照亮了我眼前的道路。在此特别感谢为这篇毕业论文成型提供过帮助好友王曼嫣，在我毕业事务繁忙的时候帮助我绘制表格、将初稿转化成 LaTex 代码，为我争取了大量的时间。

路尚未走完，但清华的烙印已深深刻在我的心里，永不磨灭。路仍在脚下，此刻不仅仅是过去的延长线，更是未来的生长点。还记得之前一位优秀的学弟在本科生特等奖学金答辩时的一句话：“我能想到的最浪漫的事便是在学术之路上远行，无须伪装，忘怀天地。”从此，这句话便成为了我的座右铭和对未来的憧憬。在未来的生活中，我愿一直追寻心中的梦想，哪怕天寒地冻，路远马亡。

感谢所有帮助我的人。

感谢我自己。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

个人简历、在学期间发表的学术论文与研究成果

个人简历

1994 年 11 月 2 日出生于安徽省安庆市。

2012 年 9 月考入清华大学软件学院计算机软件专业，2016 年 7 月本科毕业并获得工学学士学位。

2013 年 9 月进入清华大学数学系数学与应用数学专业学习，2016 年 7 月获得理学学士二学位。

2016 年 9 月免试进入清华大学软件学院攻读软件工程专业工学硕士学位至今。

发表的学术论文

[1] **Chengpeng Wang**, Yixiao Yang, Han Liu, Le Kang. Statistical API Completion Based on Code Relevance Mining. International Workshop on Mining and Analyzing Interaction Histories, 2019:7-13.（EI 收录，检索号:20191606793303）