计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2022,31(11):60−67 [doi: 10.15888/j.cnki.csa.008783]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于 Token 语义构建的代码克隆检测①

王文杰[1,2], 徐 云[1,2]

[1](中国科学技术大学 计算机科学与技术学院, 合肥 230027)
[2](中国科学技术大学 高性能计算安徽省重点实验室, 合肥 230027)
通信作者: 徐 云, E-mail: xuyun@ustc.edu.cn

摘 要: 传统的基于 Token 的克隆检测方法利用代码字符串的序列化特性, 可以在大型代码仓中快速检测克隆. 但是与基于抽象语法树 (AST)、程序依赖图 (PDG) 的方法相比, 由于缺少语法及语义信息, 针对文本有较大差异的克隆代码检测困难. 为此, 提出一种赋予语义信息的 Token 克隆检测方法. 首先, 分析抽象语法树, 使用 AST 路径抽象位于叶子节点的 Token 的语义信息; 然后, 在函数名和类型名角色的 Token 上建立低成本索引, 达到快速并有效地筛选候选克隆片段的目的. 最后, 使用赋予语义信息的 Token 判定代码块之间的相似性. 在公开的大规模数据集 BigCloneBench 实验结果表明, 该方法在文本相似度较低的 Moderately Type-3 和 Weakly Type-3/Type-4 类型克隆上显著优于主流方法, 包括 NiCad、Deckard、CCAligner 等, 同时在大型代码仓上需要更少的检测时间.
关键词: 代码克隆检测; 抽象语法树; 语义信息; 高效索引; 源代码

引用格式: 王文杰,徐云.基于 Token 语义构建的代码克隆检测.计算机系统应用,2022,31(11):60–67. http://www.c-s-a.org.cn/1003-3254/8783.html

Code Clone Detection Based on Token Semantics

WANG Wen-Jie[1,2], XU Yun[1,2]

[1](School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)
[2](Key Laboratory of High Performance Computing of Anhui Province, University of Science and Technology of China, Hefei 230027, China)

Abstract: Traditional token-based clone detection methods utilize the serialization characteristics of code strings to quickly detect clones in large code repositories. However, compared with the methods based on the abstract syntax tree (AST) and program dependency graph (PDG), traditional methods can hardly detect code clones with large text differences due to the lack of syntax and semantic information. Therefore, this study proposes a token-based clone detection method with semantic information. First, AST is analyzed, and the semantic information of tokens located at the leaf nodes is abstracted using the AST path. Then, a low-cost index is established on the tokens for function names and type roles to quickly filter valid candidate clone fragments. Finally, the similarity between code blocks is judged using the tokens with semantic information. The experimental results on the public large-scale dataset BigCloneBench reveal that this method significantly outperforms the mainstream methods, including NiCad, Deckard, and CCAligner in Moderately Type-3 and Weakly Type-3/Type-4 clones with low text similarity while requiring less detection time on large code repositories.
Key words: code clone detection; abstract syntax tree; semantic information; efficient index; source code

软件开发过程中, 开发人员通过复制代码片段, 并修改或不修改部分内容来实现新的功能, 这种方式被称为代码克隆[1]. 虽然复用代码可以加速软件的开发 , 但是也会导致软件维护困难的问题, 对软件质量产生

① 基金项目: 国家自然科学基金 (61672480); 国家外专局 111 引智计划 (BP0719016)
收稿时间: 2022-02-24; 修改时间: 2022-03-28; 采用时间: 2022-04-02; csa 在线出版时间: 2022-07-14

© 中国科学院软件研究所 http://www.c-s-a.org.cn

影响[2]. 代码克隆检测作为一项基础研究, 被应用到代码剽窃检测[3,4]、演化分析[5]、Bug 检测[6]、软件质量评估[7] 等领域. 当前研究表明, 克隆代码广泛存在于大型代码仓中, Kamiya 等人[8] 报告 JDK 中存在 29% 的克隆代码, Baker[9] 在 Linux 源码中检测到 22.3% 的代码属于克隆. 从实用性和易用性方面出发, 基于 Token 的克隆检测方法更适合大型代码仓的检测.

目前在学术界, 根据代码的不同表征形式, 克隆检测方法主要分为基于 Token[10–13]、抽象语法树 (AST)[14]、程序依赖图 (PDG)[15]、深度学习[16,17] 等. 基于 AST、PDG、深度学习的方法通过分析程序的语法及语义信息, 建模程序代码的高级别特征, 达到检测文本差异较大的克隆代码的目的, 但是这些方法通常需要消耗大量的时间进行建模及检测, 不适用于大型代码仓, 如主流的 AST 方法 Deckard[14]、PDG 方法 CCGraph[15] 分别需要消耗 1 h 12 min、15 min 10 s 检测 1 M 行代码仓, 与之相比, Token 方法 CCAligner[12] 只需 1 min 13 s. 然而, 基于 Token 的克隆检测方法由于缺少使用程序的语法及语义信息, 在检测文本差异较大的代码上效果较差, 如 CCAligner 方法在公开数据集 BigCloneBench[18] 中的 Moderately Type-3 的召回率只有 10%. 因此, 通过在 Token 方法中加入程序的语义信息, 以提高 Token 方法的检测能力, 同时维持其检测快速的特点, 是在大型代码仓中进行克隆检测的关键.

根据检测过程使用的粒度, 基于 Token 的检测方法可以分为行粒度和词粒度两类. 基于 Token 的行粒度检测方法将代码行作为检测的基本单元, 通用的流程包括统一程序的代码风格, 归一化变量等标识符, 并匹配代码片段中行之间的相似性以验证是否为克隆. CCFinder[8] 对 C 和 Java 语言分别提出 6 项代码行的转换规则, 并使用后缀树算法检测克隆, 但是 CCFinder 只支持检测 Type-1 和 Type-2 克隆. NiCad[11] 提出一种灵活的代码行切割策略, 将一行代码分为多行, 以增强对于代码行修改行为的容错性, 然后以切割后的行作为基本单元, 计算代码片段的最长公共子序列衡量相似性, 但是切割策略需要人为设定, 会引入误差. CCAligner[12] 通过建立 e-误配索引表检测存在连续代码行失配的 large-gap 克隆, 同时在大型代码仓中达到更短的检测时间. LVMapper[13] 借鉴生物信息学中序列比对的方法, 检测存在分散的失配行的 large-variance 克隆, 且能够检测 250 M 行的代码仓. 但是, 这些行粒度的检测方法在面对克隆片段的多个代码行都存在细微修改, 以及行顺序调整而不影响整体功能的情况时, 会出现检出效果不足的问题. 基于 Token 的词粒度检测方法将单词作为基本检测单元, 如 SourcererCC[10] 将代码片段通过词法解析程序转为词袋, 通过判断词袋之间的相似性验证是否为克隆. 但是 SourcererCC 没有对用户自定义的标识符 (如变量名等) 归一化, 不能检测出对标识符统一修改的克隆情况.

综上所述, 传统的基于 Token 的行粒度和词粒度克隆检测方法在大型代码仓中有较好的性能, 但是大都从文本角度出发, 没有考虑代码片段的语法及语义信息, 而限制其检测能力. 为此, 本文提出一种赋予语义信息的 Token 词粒度检测方法, 从程序的抽象语法树中获取 Token 的语义信息, 并基于赋予语义信息的 Token 增强对于文本差异较大克隆的检测能力. 另外, 使用函数名和类型名角色的 Token 建立高效索引, 达到快速检测克隆的目的. 与主流方法相比 (包括 NiCad, Deckard, CCAligner 等), 本文方法显著提升在 Moderately Type-3 克隆上的召回率, 同时在大型代码仓中需要更少的检测时间.

1 基于 Token 语义构建的代码克隆检测

本文的基于 Token 语义构建的克隆检测方法主要由两个核心步骤组成, 包括赋予 Token 语义信息和基于语义 Token 的克隆检测. 检测整体流程如图 1 所示.

赋予 Token 语义信息过程主要目的是通过分析程序的抽象语法树, 抽象 Token 的语法及语义信息, 增强 Token 的特征描述. 首先, 使用语法分析工具对每个代码块构建抽象语法树; 然后, 根据语法规则分析每个 Token 对应的角色, 将其标记为变量名、函数名、类型名等角色; 最后, 对代码块的每个 Token 从其抽象语法树的路径中获取语义信息, 将每个 Token 编码为一个固定维度的特征向量.

克隆检测过程的主要目的是通过在代码块的函数名和类型名 Token 上建立的全局索引表加速有效的候选代码块的筛选, 同时利用赋予语义的 Token 判断代码块之间的相似性. 首先, 利用所有代码块的函数名和类型名角色的 Token 建立全局索引表; 然后, 对于每个代码块, 通过查询全局索引表快速定位出候选克隆块; 其次, 针对定位得到的候选克隆块, 通过一系列的过滤策略, 去除高度不相似的代码块, 防止后续验证过程不必要的时间开销; 最后, 根据赋予语义的 Token 计算代码块之间的相似性, 从而得到克隆对.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

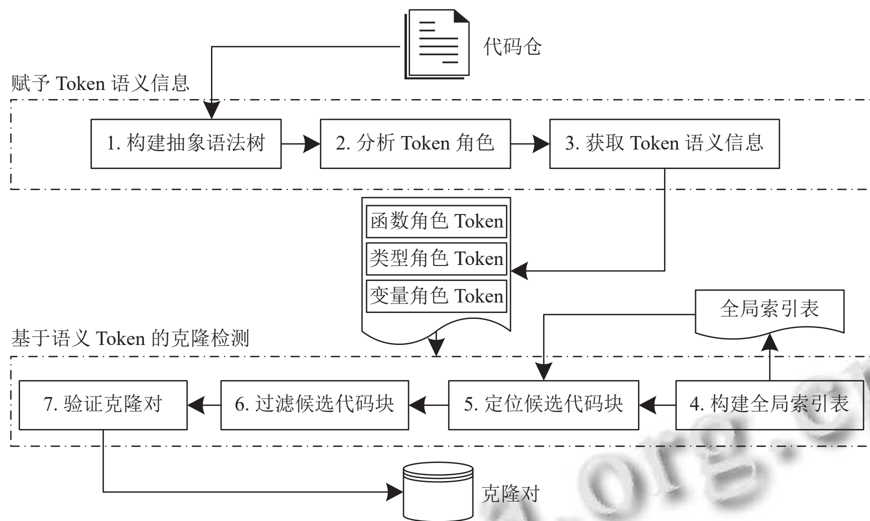

图 1 代码克隆检测整体流程

1.1 构建抽象语法树

基于开源工具 Javalang[19] 为代码块生成抽象语法树. 现有的工作[20] 表明了抽象语法树可以有效地表示代码片段的语法及语义信息. 抽象语法树通过树形结构描述源代码的结构特征, 不依赖于源代码语言的具体细节, 而对语法和语义信息进行抽象描述[21]. 抽象语法树的内部节点描述了源代码的基本语义结构, 叶子节点描述了具体的 Token 内容. 本文参考 Java 语法规范[22], 对 Javalang 生成的 AST 的内部节点进行整理, 共汇总得到 25 种内部节点类型, 如表 1 所示. 其中, 对 for 循环、do-while 循环和 while 循环 3 种结构进行等价处理, 将他们转换为 loop condition 和 loop body 两类语法节点; 同时, 将一元及二元表达式抽象为 3 类语法节点, 分别是逻辑表达式 (Logical Expression, 如>, <, ==, !=等), 数值表达式 (Numeric Expression, 如+, –, *, >>, &等) 和状态表达式 (Condition Expression, 如&&, ||等).

图 2 为 BigCloneBench 数据集[10] 中的存在克隆情况的代码块示例, 该代码块读取特定编码的文件内容, 然后将文件内容以字符串的形式返回. 图 3 为代码块生成的抽象语法树, 为简化展示的结果, 只对第 2 行和第 8 行绘制树结构. 其中, 内部节点为表 1 描述的结构信息, 叶子节点为源码具体的 Token 值.

1.2 分析 Token 的角色

Token 是组成源程序的基本词法单元, 并且 Token 可以分为很多角色, 包括关键字、标识符、常量、运算符和标点符号等, 其中标识符可以细分为变量名、函数名、类型名等角色. 受 Oreo 方法[23] 的启发, 如果两个代码块实现相似的功能 (或者存在克隆行为), 它们有很大概率调用相同的函数并定义相同的类对象[23]. 基于这种观察, 本文对 Token 的角色进行分析, 并将在第 1.4 节中使用函数名角色和类型名角色的 Token 建立全局索引表, 基于索引表查询与目标代码块存在相同的函数名和变量名 Token 的代码块, 将其作为候选代码块, 从而完成克隆对的定位操作.

表 1 抽象语法树节点类型

ID	Node type	ID	Node type
N1	If Condition	N14	Switch Body
N2	If/Else Body	N15	Switch Condition
N3	Method Definition	N16	Variable Declaration
N4	Loop Condition	N17	Assert Condition
N5	Loop Body	N18	Assert Body
N6	Array Selector	N19	Throw Body
N7	Logical Expression	N20	Try Body
N8	Numeric Expression	N21	Catch Body
N9	Condition Expression	N22	Finally Body
N10	Assign Expression	N23	Lambda Expression
N11	Method Invocation	N24	Constructor Invocation
N12	Return Statement	N25	Class/Array Creator
N13	Case Body		

本文通过分析 Javalang 中的语法规则标记 Token 的角色, 以图 4 中变量定义的语法规则为例, 赋值符号“:=”前的部分为规则名, 符号“:=”后的部分为该规则具体的定义内容. 从该规则中可以分析出, 变量定义 (图 4 第 1 行) 主要分为 3 个部分, 包括变量修饰符 (如 final 等), 变量类型 (如 int, 自定类型名等) 和变量声明

© 中国科学院软件研究所 http://www.c-s-a.org.cn

列表. 变量声明列表 (图 4 第 6 行) 由一系列逗号分隔的变量定义体构成. 变量定义体 (图 4 第 7 行) 由两部分构成, 包括变量名 (即 VariableDeclaratorId) 和初始化列表 (即 VariableInitializer), 位于 VariableDeclaratorId 部分的标识符即为变量名角色. 如图 2 第 2 行, *fis* 的角色为变量名, FileInputStream 的角色为类型名.

```
1 public static String load (File file, String encoding) throws IOException {
2    FileInputStream fis=new FileInputStream (file);
3    InputStreamReader reader=new InputStreamReader (fis, encoding);
4    StringBuffer strBuf=new StringBuffer ();
5    char [] buffer=new char [4096];
6    int i=0;
7    try {
8      while ((i=reader. read (buffer)) != −1) strBuf. append (buffer, 0, i);
9    } finally {
10     fis. close ();
11   }
12   return strBuf. toString ();
13 }
```

图 2　BigCloneBench 数据集中的克隆代码块示例

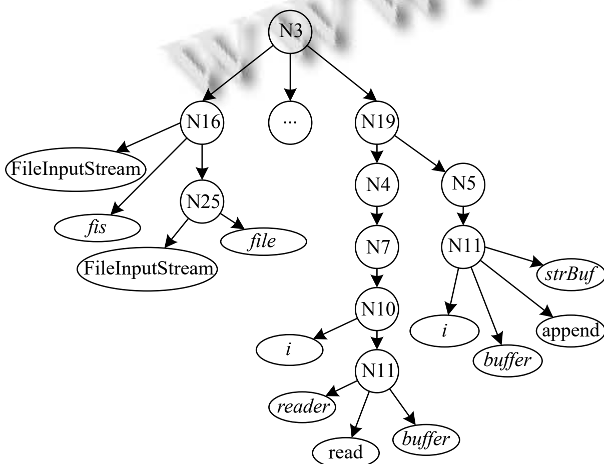

图 3　图 2 代码生成的抽象语法树

```
1 VariableDeclaration: =VariableModifier* Type DeclaratorList;
2 VariableModifier: ='final' | Annotation
3 Type: =BasicType | ReferType
4 ReferType: =Identifier
5 BasicType: ='float' | 'int' | 'double' | 'short' | 'long' | 'char' | 'byte'
6 DeclaratorList: =VariableDeclarator (',' VariableDeclarator)*
7 VariableDeclarator: =VariableDeclaratorId ('=' VariableInitializer)?
8 VariableDeclaratorId: =Identifier dims?
```

图 4　Javalang 中的变量定义语法规则

1.3　获取 Token 语义信息

本文从代码块的抽象语法树出发, 分析 Token 的语义信息, 通过 Token 的高维特征, 而不是仅从文本角度来验证 Token 之间的相似性, 达到检测结构相似的克隆对的目的. 抽象语法树的内部节点构成了程序的骨架结构, 叶子节点填充了具体的 Token 信息, 从根节点至叶子节点的路径描述了 Token 在抽象语法树中的结构特征. 如果两个 Token 有相似的路径结构特征, 那么这两个 Token 的功能存在相似性, 进一步地, 如果两个代码块中存在众多路径结构特征相似的 Token, 那两个代码块很可能存在克隆行为. 标识符角色的 Token 表达了程序的功能, 其他角色的 Token (如关键字、常量、运算符等) 主要用于修饰程序的结构, 所以本文通过衡量两个代码块的标识符 Token 的相似性, 从而衡量代码块的相似性. 对于函数名和类型名角色的 Token, 本文使用其文本字符串的形式, 对于变量名角色的 Token, 本文编码其语义信息, 将其抽象为一个特征向量.

为了编码 Token 在抽象语法树中的结构特征信息, 本文先序遍历抽象语法树, 记录叶子节点 Token 的路径信息, 路径信息是由表 1 的节点构成的节点序列. 本文将节点序列构造为一个 25 维的整型向量, 向量中的每一维代表一种类型的节点, 每一维的值代表节点的频度, 另外, 如果 Token 在代码块中出现多次, 则将其对应的所有路径向量加和作为其最终的特征向量. 例如, 图 2 第 8 行的 *strBuf* 在图 3 的路径序列为 N3 (Method Definition)-N19 (Throw Body)-N5 (Loop Body)-N11 (Method Invocation), 编码的 25 维特征向量为 [0, 0, 1, 0, 1, ⋯, 1, ⋯, 1, ⋯, 0], 其中第 3、5、11、19 维的值为 1, 其他位置的值为 0, 另外 *strBuf* 出现在第 4 和 12 行, 分析对应位置的特征向量, 将它们与第 8 行的特征向量加和之后作为 *strBuf* 的最终特征向量.

1.4　建立全局索引表

为了防止在代码仓中成对地匹配代码块来验证克隆对的相似性, 本文借鉴 CCAligner[12] 和 LVMapper[13] 方法, 构造索引表, 通过查询索引表获得指定代码块的候选克隆代码块. 由于 CCAligner 和 LVMapper 是行粒度的 Token 检测方法, 本文对构建过程进行改造, 使用代码块的函数名和类型名角色的 Token, 对其建立 n-gram 序列, 生成索引表.

具体的操作步骤包括: 首先, 对每个代码块的函数名和类型名 Token 按照字典序进行升序排序; 然后, 对于每个排好序的长度为 m 的 Token 序列, 按照窗口大小为 k, 步长为 1 的方式进行滑动, 获得 $m–k+1$ 个 k-gram 字符串序列; 最后, 对 k-gram 字符串序列做哈希操作得到哈希值, 将哈希值作为索引项, 代码块的 ID 作为值, 插入到全局索引表中. 全局索引表的键的数据结构为 k-gram 序列的哈希值, 值的数据结构为包含该 k-gram

© 中国科学院软件研究所 http://www.c-s-a.org.cn

序列的代码块 ID 数组. 例如, 如果代码块 B_1 的函数名和类型名 Token 为 A, B, C, D, E, 设定窗口 k=3, 则索引表的内容为{hash(A, B, C):[B_1], hash(B, C, D):[B_1], hash(C, D, E): [B_1]}.

1.5 定位候选代码块

定位过程主要是尽可能多地找到潜在克隆代码块, 不需要保证克隆代码块的真伪. 如果两个代码块存在克隆行为, 那么它们的函数名和类型名角色的 Token 存在重叠的情况. 在第 1.4 节中定义的全局索引表的索引项是对函数名和类型名角色 Token 信息的聚合, 而互为克隆的代码块之间存在相同的聚合信息. 因此, 可以对指定代码块的函数名和类型名角色的 Token 构造 k-gram 序列后, 查找全局哈希表, 得到的代码块 ID 数组即为候选代码块, 具体的步骤如算法 1 所示.

算法 1. 定位与指定代码块存在克隆的候选代码块法

输入: 全局索引表 M, 查询的代码块 B.
输出: 候选克隆代码块集合 Q.

1 对序列 S 按照窗口大小为 k, 步长为 1 的方式进行滑动, 生成 $m-k+1$ 个 k-gram 列表, 即 [$T_1, T_2, \cdots, T_{|S|-k+1}$].
2 初始化候选克隆代码块集合 Q 为空; 获取代码块 B 的函数名和类型名角色的 Token 序列, 并按照字典序升序排序, 得到序列 S.
3 对于列表 [$T_1, T_2, \cdots, T_{|S|-k+1}$] 的每个 k-gram 序列, 进行 hash 操作得到其哈希值 H_i=hash(T_i).
4 将哈希值 H_i 作为关键字查询全局索引表 M, 得到对应的值列表 V_i=get(M, H_i); 然后合并列表 V_i 到集合 Q 中.
5 返回候选克隆代码块集合 Q.

1.6 过滤候选代码块

过滤过程主要目的是通过低时间复杂度的计算方式, 删去定位过程中得到的候选代码块中的伪克隆代码块, 避免这些伪克隆在后续验证过程中被再次计算. 本文通过两种策略进行过滤操作, 也就是代码块的规模差距和代码块中相同函数名和类型名角色 Token 的覆盖率. 如果两个代码块的 Token 个数差距过大, 那么它们互相之间不太可能存在克隆关系, 其计算公式如式 (1) 所示, 并限制个数的比例不小于阈值 α. 另外, 如果两个代码块的相同函数名和类型名角色的 Token 的比例太小, 那么它们可能主要是实现不同的功能, 而不存在克隆关系, 其计算公式如式 (2) 所示, 并限制比例不小于阈值 β.

$$SR = \frac{\min(|B_1|, |B_2|)}{\max(|B_1|, |B_2|)} \tag{1}$$

其中, B_1, B_2 分别为指定代码块和候选代码块; $|B_1|$, $|B_2|$分别表示代码块 B_1, B_2 中标识符 Token 的总数.

$$TR = \frac{count_same_token(T_1, T_2)}{\min(|T_1|, |T_2|)} \tag{2}$$

其中, T_1, T_2 分别为代码块 B_1, B_2 的函数名和类型名 Token, 函数 *count_same_token* 用于计数相同函数名和类型名 Token 的个数.

1.7 验证克隆对

在过滤阶段完成之后, 候选代码块中包含一些潜在的代码块与指定代码块存在克隆关系. 验证阶段需要计算指定代码块与候选代码块的相似度, 从而得到克隆对. 克隆对的相似性计算包含两个方面, 一方面是函数名和类型名角色在文本上的相似性, 另一方面是赋予语义信息的变量名角色 Token 的相似性.

本文提出一种多轮匹配的近似算法计算两个代码块变量名角色 Token 的相似性. 每个变量名角色的 Token 编码为 25 维的向量, 则代码块的变量名角色的 Token 可形成一个向量集合, 两个代码块变量名角色 Token 相似度的计算转化为两个向量集合相似度的计算, 考虑到精确的向量集合相似度计算的时间复杂度为O(n!), 本文提出一种近似的计算方式, 将时间复杂度优化到O(kn^2), 具体的步骤如算法 2 所示.

算法 2. 计算两个变量名角色 Token 向量集合的相似度

输入: S_1, S_2 are vector collection of variables for block B_1 and B_2, respectively. η is decreasing ratio.
输出: VR is the the similarity between S_1 and S_2.

```
1 sim_sum = 0; // sum of matched vectors
2 initialize array M1 to size |S1| with default value 0;
3 initialize array M2 to size |S2| with default value 0;
4 for t=1.0 to 0.0 step η do
5    for i=0 to |S1| do
6       if M1[i]==1 then continue;
7       for j=0 to |S2| do
8          if M2[j]==1 then continue;
9          s=cosine(S1[i], S2[j]);
10           if s⩾t then
11              sim_sum=sim_sum+s;
12              M1[i]=1, M2[j]=1;
13           end if
15       end for
16    end for
17 end for
18 return sim_sum/max(|S1|, |S2|);
```

在算法 2 中, 对于给定的两个向量集合和一个预设的下降率 η (介于−1.0 和 0.0 之间的小数), 计算向量

© 中国科学院软件研究所 http://www.c-s-a.org.cn

集合的相似度. 在计算过程中, 每轮逐次减少匹配的阈值 t (第 4 行), 遍历向量集合 (第 5, 7 行), 计算选定的两个向量的余弦相似度 (第 9 行); 将余弦相似度大于等于本轮阈值 t 的两个向量进行绑定 (第 12 行), 并将这两个向量的余弦相似度计入总和 sim_sum 中; 另外, 如果选定的向量已经被绑定 (第 6, 8 行), 则跳过向量的匹配; 最后将计算的向量和除以两个向量集合规模的最大值 (第 18 行), 作为向量集合的相似度 VR.

最后, 克隆对 B_1, B_2 的相似度的 PR 计算方法如式 (3) 所示, 将相似度大于阈值 φ 的克隆对输出.

$$PR = \mu \times TR + (1-\mu) \times VR \tag{3}$$

其中, TR 为克隆对的相同函数名和类型名 Token 的比例, VR 为克隆对的变量名 Token 的相似度, μ 表示代码块中函数名和类型名 Token 在代码块的全部标识符 Token 的占比, 不同代码块中的 μ 大小有所差异, 其计算方式如式 (4):

$$\begin{cases} \mu_1 = |T_1|/|B_1| \\ \mu_2 = |T_2|/|B_2| \\ \mu = (\mu_1+\mu_2)/2 \end{cases} \tag{4}$$

其中, T_1, T_2 分别为代码块 B_1, B_2 的函数名和类型名 Token, 取 μ_1, μ_2 的平均值作为克隆对的 μ 值.

2 实验结果与分析

2.1 实验数据与环境

本文采用公开克隆检测数据集 BigCloneBench[18] 进行实验. BigCloneBench 是由加拿大的 Svajlenko 团队建立的人造数据集, 它从 IJaDataset-2.0 中挖掘克隆, 并标记出 8 584 153 个真克隆对和 279 032 个假克隆对. 其中包含 55 499 个 Java 文件, 根据功能分类保存到 43 个子文件夹中. 本文在 BigCloneBench 数据集上进行克隆检测从而验证工具的有效性.

为了对比不同工具可扩展性的差别, 本文从 IJaDataset-2.0 中分别抽取出 1 M、10 M、20 M、30 M、250 M LOC (lines of code) 代码构成规模数据集, 在不同的规模数据集上进行实验, 比较不同工具的检测用时从而体现可扩展性的差异.

实验的运行环境为 Ubuntu 18.04 系统, Intel(R) Xeon(R) Gold 5120 2.20 GHz CPU, 503 GB 内存空间.

2.2 实验评价指标

本文采用精确率 P (precision)、召回率 R (recall) 和时间性能 3 个指标来评估克隆检测工具. 精确率 P 表示被检测出来的克隆对属于真克隆的概率, 如式 (5) 所示; 召回率 R 表示真克隆对被检测出来的概率, 如式 (6) 所示:

$$P = \frac{TP}{TP+FP} \tag{5}$$

$$R = \frac{TP}{TP+FN} \tag{6}$$

其中, TP 表示被预测为克隆且为真克隆对的数量; FP 表示被预测为克隆但为假克隆对的数量; FN 表示没有被检测出来的真克隆对的数量.

在评估工具的召回率时, 采用 BigCloneEval[24] 评估框架. BigCloneEval 是基于 BigCloneBench 设计的自动化评估工具, 它将标记出的真克隆对与克隆检测工具报告的克隆对进行比较, 根据被检测出的真克隆对数量报告克隆检测工具的召回率, 并将召回率分为 Type-1 (T1)、Type-2 (T2)、Very Strongly Type-3 (VST3, 克隆对相似度介于 0.9–1.0)、Strongly Type-3 (ST3, 克隆对相似度介于 0.7–0.9)、Moderately Type-3 (MT3, 克隆对相似度介于 0.5–0.7) 和 Weakly Type-3/Type-4 (WT3/T4, 克隆对相似度介于 0–0.5) 共 6 类.

在评估克隆检测工具的精确率时, 由于克隆检测结果数量庞大 (接近百万个克隆对), 难以逐个验证克隆对的真伪. 本文采用学术界广泛使用的方法[10,12], 从克隆检测结果中随机选出 400 个克隆对, 由 3 位超过 6 年编程经验的研究者人工验证克隆对的真伪, 根据验证的结果计算克隆检测工具的精确率.

2.3 检测方法有效性的实验结果及分析

为了对比本文克隆检测方法的有效性, 选取代表性较强的 5 类方法, 包括基于 Token 的方法: NiCad[11]、SourcererCC[10]、CCAligner[12] 和 LVMapper[13], 以及基于 AST 的检测方法: Deckard[14]. 通过对比不同检测工具在 BigCloneBench 数据集上的精确率和召回率验证方法的有效性, 结果如表 2 所示.

实验结果表明, 本文提出的克隆检测方法在精确率方面有较好的检测效果, 达到 91%, 表明检测出的克隆对基本上都是真克隆. 在召回率方面, 对 T1、T2 类型的克隆对检测达到接近 100% 的效果, 略低于 NiCad、CCAligner 和 LVMapper, 在后续分析中发现, 没有达到 100% 的原因是 Javalang 工具在解析部分源文件时出现错误, 无法获得程序的抽象语法树, 导致与该程序相关的克隆对无法被报告. 在 VST3 和 ST3 类

© 中国科学院软件研究所 http://www.c-s-a.org.cn

型的克隆上, 本文的方法略低于 NiCad 方法, 但仍处于领先水平.

表 2 不同方法在 BigCloneBench 上克隆检测结果的对比 (%)

方法	R						P
	T1	T2	VST3	ST3	MT3	WT3/T4	
本文方法	99	98	98	88	43	2.3 (176, 712)	91
NiCad	100	100	100	95	1	0 (12)	94
SourcererCC	100	98	93	61	5	0 (1, 892)	98
CCAligner	100	99	97	70	10	0.2 (12, 540)	77
LVMapper	100	99	98	82	19	0.3 (23, 923)	87
Deckard	60	58	58	31	12	1.0 (77, 293)	35

注: 由于克隆检测方法在WT3/T4的召回率接近0, 在括号中描述出BigCloneEval的报告中检出的克隆对数量, 体现各种方法效果的差异.

另外, 相比于 NiCad、SourcererCC、CCAligner、LVMapper 和 Deckard 方法, 本文方法在检测 MT3 和 WT3/T4 类型的克隆上有最好的检测效果, 在 MT3 类型上召回率达到 43%, 同时检测到 176 712 对 WT3/T4 类型的克隆. 这是由于 NiCad、SourcererCC、CCAligner 和 LVMapper 这些传统的基于 Token 方法单从程序的文本角度出发, 它们可以在 T1、T2 等不存在语句变异 (如插入、删除和修改等) 的克隆上呈现较好的效果, 但是对于语句行出现差别的克隆对 (也就是 MT3 和 WT3/T4 克隆), 这些方法出现检测不足的问题. 而基于 AST 的方法 Deckard, 由于其从 AST 子树匹配的角度分析, 对 AST 整体结构的建模能力不足, 在 MT3 和 WT3/T4 这样差异较大的克隆上效果不好. 本文方法从 AST 的路径上分析 Token 的语义信息, 同时考虑克隆过程中变化不大的函数名和类型名角色的 Token, 在检测文本差异较大的克隆对上呈现较好的检测能力.

2.4 时间性能的实验结果及分析

随着软件功能的丰富, 当前的代码仓的体量呈现逐渐增大的趋势, 克隆检测方法在大型代码仓的检测时间是一项需要重点关注的指标. 本实验将与 NiCad、SourcererCC、CCAligner 和 LVMapper 方法对比, 统计不同方法在 1 M、10 M、20 M、30 M、250 M 行规模数据集上的运行时间, 评估不同方法的可扩展性的差异. 由于 Deckard 方法无法运行在 10 M LOC 数据集, 所以不对其进行展示, 其他方法的结果如表 3 所示.

从表 3 中可以看出, 本文提出的克隆检测方法在 1 M、10 M、30 M 和 250 M LOC 的规模数据集上需要最少的检测时间, 达到最好的可扩展性效果, 在 20 M LOC 数据集上略差于 LVMapper 方法. 能够快速检测的原因有两方面, 一方面是在定位过程中快速地查询到与指定代码块可能存在克隆的候选代码块, 并且通过低成本的过滤阶段去掉部分伪候选; 另一方面是在验证阶段优化相似性比对方法, 降低了时间复杂度.

表 3 不同方法在规模数据集上克隆检测耗时的对比

方法	1 M	10 M	20 M	30 M	250 M
本文方法	34 s	5 min 10 s	27 min 40 s	1 h 2 min	25 h 43 min
NiCad	4 min 10 s	10 h 36 min	—	—	—
SourcererCC	5 min 40 s	27 min 52 s	1 h 2 min	1 h 32 min	56 h 41 min
CCAligner	1 min 8 s	48 min 36 s	—	—	—
LVMapper	38 s	7 min 44 s	19 min 24 s	1 h 35 min	52 h 25 min

3 结论与展望

为增强传统 Token 克隆检测方法在文本较大差异的克隆上的检测能力, 本文提出一种为 Token 赋予语义信息且适用于大型代码仓的方法. 该方法从程序的抽象语法树中分析 Token 的语义信息, 并从语义角度比较 Token 的相似性. 在检测过程中通过构建索引表、定位、过滤和验证步骤, 达到在大型代码仓中快速检测的目的. 实验结果表明, 相比于其他克隆检测方法 (NiCad, Deckard, CCAligner 等), 赋予语义的 Token 可以更好地表征语义信息, 显著提高在 MT3 克隆上的检测效果; 同时, 本方法在大型代码仓中表现出更加突出的时间性能. 在后续的研究中, 可以进一步优化 Javalang 生成抽象语法树的过程, 提高 T1、T2 类型的召回率, 另外可以尝试程序依赖图 (PDG) 中提取特征, 增强检测的能力.

参考文献

1 Roy CK, Cordy JR. A survey on software clone detection research. Technical Report, Ontario: Queen's University at Kingston, 2007. 64–68.

2 Choi E, Yoshida N, Ishio T, *et al*. Extracting code clones for refactoring using combinations of clone metrics. Proceedings of the 5th International Workshop on Software Clones. Honolulu: ACM, 2011. 7–13. [doi: 10.1145/1985404.1985407]

3 Cosma G, Joy M. An approach to source-code plagiarism detection and investigation using latent semantic analysis. IEEE Transactions on Computers, 2012, 61(3): 379–394. [doi:

© 中国科学院软件研究所 http://www.c-s-a.org.cn

10.1109/TC.2011.223]

4 Liu C, Chen C, Han JW, *et al*. GPLAG: Detection of software plagiarism by program dependence graph analysis. Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Philadelphia: ACM, 2006. 872–881. [doi: 10.1145/1150402.1150522]

5 Saha RK, Asaduzzaman M, Zibran MF, *et al*. Evaluating code clone genealogies at release level: An empirical study. Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation. Timisoara: IEEE, 2010. 87–96. [doi: 10.1109/SCAM.2010.32]

6 Li JY, Ernst MD. CBCD: Cloned buggy code detector. Proceedings of the 34th International Conference on Software Engineering. Zurich: IEEE, 2012. 310–320. [doi: 10.1109/ICSE.2012.6227183]

7 Thummalapenta S, Cerulo L, Aversano L, *et al*. An empirical study on the maintenance of source code clones. Empirical Software Engineering, 2010, 15(1): 1–34. [doi: 10.1007/s10664-009-9108-x]

8 Kamiya T, Kusumoto S, Inoue K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering. 2002, 28(7): 654–670. [doi: 10.1109/TSE.2002.1019480]

9 Baker BS. On finding duplication and near-duplication in large software systems. Proceedings of 2nd Working Conference on Reverse Engineering. Toronto: IEEE, 1995. 86–95. [doi: 10.1109/WCRE.1995.514697]

10 Sajnani H, Saini V, Svajlenko J, *et al*. SourcererCC: Scaling code clone detection to big-code. Proceedings of the 38th International Conference on Software Engineering. Austin: IEEE, 2016. 1157–1168. [doi: 10.1145/2884781.2884877]

11 Roy CK, Cordy JR. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. Proceedings of the 16th IEEE International Conference on Program Comprehension. Amsterdam: IEEE, 2008. 172–181. [doi: 10.1109/ICPC.2008.41]

12 Wang PC, Svajlenko J, Wu YZ, *et al*. CCAligner: A token based large-gap clone detector. Proceedings of the 40th International Conference on Software Engineering. Gothenburg: IEEE, 2018. 1066–1077. [doi: 10.1145/3180155.3180179]

13 Wu M, Wang PC, Yin KQ, *et al*. LVMapper: A large-variance clone detector using sequencing alignment approach. IEEE Access, 2020, 8: 27986–27997. [doi: 10.1109/ACCESS.2020.2971545]

14 Jiang LX, Misherghi G, Su ZD, *et al*. DECKARD: Scalable and accurate tree-based detection of code clones. Proceedings of the 29th International Conference on Software Engineering. Minneapolis: IEEE, 2007. 96–105. [doi: 10.1109/ICSE.2007.30]

15 Zou Y, Ban BH, Xue YX, *et al*. CCGraph: A PDG-based code clone detector with approximate graph matching. Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. Melbourne: IEEE, 2020. 931–942. [doi: 10.1145/3324884.3416541]

16 Fang CR, Liu ZX, Shi YY, *et al*. Functional code clone detection with syntax and semantics fusion learning. Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. Online: ACM, 2020. 516–527. [doi: 10.1145/3395363.3397362.]

17 Yu H, Lam W, Chen L, *et al*. Neural detection of semantic code clones via tree-based convolution. Proceedings of the 27th International Conference on Program Comprehension. Montreal: IEEE, 2019. 70–80. [doi: 10.1109/ICPC.2019.00021]

18 Svajlenko J, Islam JF, Keivanloo I, *et al*. Towards a big data curated benchmark of inter-project code clones. Proceedings of the IEEE International Conference on Software Maintenance and Evolution. Victoria: IEEE, 2014. 476–480. [doi: 10.1109/ICSME.2014.77]

19 Thunes C. Javalang. https://github.com/c2nes/javalang. (2021-10-18)[2021-12-10].

20 Ullah F, Jabbar S, Al-Turjman F. Programmers' de-anonymization using a hybrid approach of abstract syntax tree and deep learning. Technological Forecasting and Social Change, 2020, 159: 120186. [doi: 10.1016/j.techfore.2020.120186]

21 许健, 陈平华, 熊建斌. 融合滑动窗口和哈希函数的代码漏洞检测模型. 计算机应用研究, 2021, 38(8): 2394–2400.

22 ORACLE. Java language and virtual machine specifications. https://docs.oracle.com/javase/specs/index.html[2021-12-11].

23 Saini V, Farmahinifarahani F, Lu YD, *et al*. Oreo: Detection of clones in the twilight zone. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Lake Buena Vista: ACM, 2018. 354–365. [doi: 10.1145/3236024.3236026]

24 Svajlenko J, Roy CK. BigCloneeval: A clone detection tool evaluation framework with BigCloneBench. Proceedings of IEEE International Conference on Software Maintenance and Evolution. Raleigh: IEEE, 2016. 596–600. [doi: 10.1109/ICSME.2016.62]

(校对责编: 孙君艳)

© 中国科学院软件研究所 http://www.c-s-a.org.cn