文章编号：2095-2163(2019)03-0146-05 中图分类号：TP311.5 文献标志码：A

基于代码变更块和抽象语法树的两种重构模式识别

张志浩，杨春花
（齐鲁工业大学（山东省科学院）计算机科学与技术学院，济南 250353）

摘 要： 内联函数（Inline method）和替换算法（Substitute algorithm）是 2 种在代码重构中常用的重构手法，本文提出一种基于代码变更块和抽象语法树的重构模式识别算法，首先筛选出变更前后 2 个文件的代码变更块，找到可能属于这 2 种重构模式的代码变更块，再建立抽象语法树对这些变更块中的代码进行准确的语法分析，对其是否属于此 2 种模式进行判定。该算法在 4 个开源项目上进行了实验验证，表明了其具有较高的准确率。

关键词： 重构模式；抽象语法树；代码变更块；内联函数；替换算法

Identifying two refactoring patterns based on hunk and abstract syntax tree

ZHANG Zhihao, YANG Chunhua
(School of Computer Science and Technology, Qilu University of Technology (Shandong Academy of Sciences), Jinan 250353, China)

【Abstract】 Inline method and Substitute algorithm are two kinds of refactoring methods which are used in code refactoring frequently. This paper proposes a refactoring pattern recognition algorithm based on hunk and abstract syntax tree. First filter out all the hunks of the two files, find the specific hunks that may contain these two kinds of refactoring patterns, then establish an abstract syntax tree to analyze the code in this hunks more accurately and verdict whether it matches these two refactoring patterns. The algorithm has been experimentally verified on four open source projects, indicating that it has a high accuracy.

【Key words】 refactoring pattern; abstract syntax tree; hunk; Inline method; Substitute algorithm

0 引 言

代码重构[1]是在软件开发的过程中一种常用的在结构层次上的代码整理手段，其能在不改变软件可观察行为的前提下，对软件的结构进行调整，提高其可拓展性、可维护性、可读性，从而提升其质量，并降低其维护成本。

目前对重构的研究主要集中在代码自动重构这一方面，相关研究主要有：使用抽象语法树和静态代码分析的克隆代码自重构方法[2]、基于 K-近邻的 C 克隆代码重构方法[3]、单例模式导向的源代码自动重构研究[4]、面向 Java 锁机制的字节码自动重构框架[5]等等。

重构模式识别指的是对比变更前原代码和变更后代码以寻找符合某种重构模式代码段。在代码更新中，往往包含着对老版本代码的 bug 的修复、功能的添加以及重构的代码变更，这 3 种种类各异的代码变更方式的掺杂增加了阅读其代码、理解其内容的难度，若能对其中的重构代码进行自动识别，则可使重构与其他种类的变更相互分离，利于代码的阅读和理解。对代码重构模式的识别的研究也在继续深入之中，Fokaefs 等人[6]开发了一种 eclipse 插件、Weissgerber 等人[7]提出了基于签名分析的方法、Prete 等人[8]开发了 REF-FINDER，这种工具可以对几种重构模式进行识别，刘阳等人[9]提出了一种方法可以识别函数抽取。钟林辉等人[10]提出了一种基于版本的多重软件重构自动检测技术。以上研究中没有涉及到有关内联函数和替换算法这 2 种重构模式的识别，本文即对此实现了相应的识别算法。

1 内联函数与替换算法重构模式的识别

1.1 变更代码块

目前对代码变更的提取方法有 2 种。一种是语法分析法，通过对比 2 个文本的抽象语法树获取差异部分，另一种基于文本差异的对比方法，直接比较目标文本的差异，获取差异部分。

本文考虑到算法效率，采用第二种方式，通过基于文本差异的对比方法获取其基本输出单位变更代

基金项目：国家自然科学基金（61502259）。

作者简介：张志浩（1994-），男，硕士研究生，主要研究方向：代码变更分析、软件演化；杨春花（1974-），女，博士，教授，主要研究方向：代码变更分析、软件演化、软件开发。

收稿日期：2019-03-02

码块(hunk),一个 hunk 可以由删除行和添加行这 2 个部分组成,也可以只包含删除行部分或添加行部分,研究中列出了同属于项目 google_guice(https://github.com/google/guice/)的 3 个 hunk 即如图 1 所示。其中,hunk1 中的 67~76 行是删除行,行首为减号,71~73 行为添加行,行首为加号;hunk2 只包含添加行;hunk3 只包括删除行。

(a)hunk 1

(b)hunk2

(c)hunk3

图 1　hunk 示例

Fig. 1　hunk example

1.2　抽象语法树

抽象语法树是在编程语言源代码的翻译和编译过程中使用的一种中间表示形式,其结构类似于树状结构,树上的每一个节点都表示源代码中的一种结构,但是抽象语法树的节点并不是对应真实语法的每一个细节,比如对于嵌套括号等没有对应的节点,故称为抽象语法树。利用抽象语法树可以对代码进行一定的语法分析。

1.3　内联函数重构模式的识别

1.3.1　内联函数模式

内联函数模式在代码重构中经常出现,当遇见某个函数的内部代码和其函数名同样清晰易读的小粒度函数时,可以使用这个重构模式。一般情况下,其运行过程是将这一个或几个小粒度函数的函数体提取到对此函数的调用语句处,并删除调用语句和此小粒度原函数。

继而,研究中列出了 2 个 hunk 如图 2 所示,且均属于 google_guice(https://github.com/google/guice/)项目中版本为 315ee3fe10 的 InjectorImpl.java 文件,hunk4 中(729~734 行)中的 *injectMembers*( ) 函数整体被移除,函数体被转移到 hunk5 的(756~758 行) *call*( ) 函数中,并且 *call*( ) 函数中的对 *injectMembers*( ) 函数的引用(776 行)也被移除,这就是一种典型的内联函数重构模式。

(a)hunk4

(b)hunk5

图 2　内联函数模式示例

Fig. 2　Inline method example

1.3.2　内联函数模式识别

假设原代码中有 A,B 两个函数,设为 A_1,B_1,变更后代码中也有 A 函数,设为 A_r,若此次变更为内联函数重构模式,则此代码所具有特点可概述为:

(1)B_1 函数的函数体代码被添加到变更后代码中的 A_r 函数里,之后 B_1 函数被整体移除。

(2)A_1 中有对 B_1 的引用,但 A_r 函数中没有对 B_1 函数的引用语句。

对这 2 个特点的判断是识别内联函数模式的重点,为了方便起见,称可能符合上述 2 个特点的函数,如 A_1,A_r 为 A 型节点,可能符合上述 2 个特点的函数,如 B_1 为 B 型节点,下文对这 2 个特点的识别方法分别进行研究阐述。

1.3.3　基于相似度检测对函数级代码移动进行判断

由于要单独使用到 hunk 中的删除行或添加行,方便起见,按照添加行和删除行将各个 hunk 分为 2 个部分,即添加部分(addpart)和删除部分(delpart),如图 3 所示,图 1 中的 hunk1 拆分为 addpart 和 delpart。

(a) delpart

(b) addpart

图 3　hunk1 的分离

Fig. 3　separation of hunk1

第一个特点本质是检测是否有函数级代码的移动,设 B_1 函数包含在某个 hunk(设为 h_1)的删除行中,而 A_r 中增加的代码则包含在某个 hunk(设为 h_2)的添加行中,因此对第一个特点的判断就是判

断 h_1 中的删除行和 h_2 中的添加行是否有相似关系，即 h_1.delpart 是否和 h_2.addpart 存在相似关系。

1.3.4 基于抽象语法树对函数引用情况进行判断

对第二个特点的判断需要考察 A_1 函数代码中是否有对 B_1 函数的引用语句，以及 A_r 函数中是否已经移除对 B_1 函数的引用语句，对此研究过程可阐释如下。

(1)在符合第一个特点的情况下，对原代码文件和变更后代码文件分别生成抽象语法树，并且获取 2 棵语法树的所有函数节点(Method node)，函数节点有行范围，h_1.delpart 和 h_2.addpart 同样有行范围，对比匹配两者的行范围可以找出 B_1 和 A_r 函数节点，因为函数节点包含函数头的所有信息，故可获取 A_r 函数节点的函数名，并根据此函数名遍历原代码文件的所有函数节点，找到同名同参数的 A_1 函数节点。

(2) 遍历 A_r 函数节点的所有子节点，确定是否存在一个对 B_1 的方法调用节点，并于 A_1 函数节点中确认此方法调用节点是否已经被删除。

1.3.5 内联函数重构模式识别算法

此算法输入 2 个版本的源文件 $file_l$ 和 $file_r$ 与第一步输出 hunk 集合 $H = \{ < h_1, h_2 > \}$，其中 h_1 的删除部分和 h_2 的添加部分有相似关系，输出存在内联函数模式的函数节点(Method node) 集合 $O = \{ < n_1, n_2, n_3 > \}$，其中 n_1，n_2 是 $file_l$ 的 A 型和 B 型节点，n_3 是 $file_r$ 的 A 型节点，n_1，n_2，n_3 之间存在内联函数重构模式关系。算法的伪代码详见如下。

输入：2 个版本的源文件 $file_l$ 和 $file_r$ 与第一步输出的 hunk 集合 H

输出：存在内联函数模式的函数节点(Method node)集合 O

```
(H_l, H_r) ← getAllHunks(file_l, file_r)
H ← filter(H_l, H_r)
(H_d, H_a) ← split(H)
R_d ← gethunkrange(H_d)
R_a ← gethunkrange(H_a)
t_l ← generateAST(file_l)
t_r ← generateAST(file_r)
M_1 ← getAllMethodNodes(t_l)
M_2 ← getAllMethodNodes(t_r)
M_del ← ∅, M_add ← ∅
for each r_d ∈ R_d, r_a ∈ R_a
  for each m_1 ∈ M_1, m_2 ∈ M_2
    r_m1 ← getMethodRange(m_1)
    r_m2 ← getMethodRange(m_2)
    if r_m1 ⊆ r_d
      M_del ← M_del ∪ m_1
    end if
    if r_a ⊆ r_m2
      M_add ← M_add ∪ m_2
    end if
  end for
end for
M_A ← getLeftAMethod(M_add)
for each (m_add, m_del, m_A) ∈ (M_add, M_del, M_A)
  if checkInvoc((m_add, m_del, m_A))
    O ← O ∪ (m_add, m_del, m_A)
  end if
end for
```

算法第 1 行使用 *getAllHunks*( ) 函数获取 2 个版本的源文件 $file_l$，$file_r$ 的所有 hunk 集合(H_l, H_r)；算法第 2 行使用 *filter*( ) 函数选取 H_l 中的 hunk 的删除部分和 H_r 中的 hunk 的添加部分相似的 hunk 集合 $H = \{ < h_1, h_2 > \}$；算法第 3 行使用 *split*( ) 函数将 H 中属于 $file_l$ 的 hunk.delpart 取出来放进 H_d，将属于 filer 的 hunk.addpart 取出来放进 H_a；第 4，5 行获取行范围 R_d，R_a；第 11 ~ 22 行对 R_d，R_a 和 M_1，M_2 的行范围 r_{m1}，r_{m2} 进行对比匹配，挑选出 H_d 的行范围包含的函数节点集合 M_{del} 的行范围和行范围包含 r_a 的函数节点集合 M_{add}；第 23 行通过 *getLeftAMethod*( ) 函数获取属于 $file_l$ 的和 M_{add} 同名函数节点集合 M_A；第 24 ~ 28 行对属于 $file_l$ 的 A 型节点 m_A，B 型节点 m_{del}，属于 $file_r$ 的 A 型节点 m_{add} 通过 *checkInvoc*( ) 函数判断 m_A 中是否有对 m_{del} 的调用语句和 m_{add} 中是否已经移除对 m_{del} 的调用语句。

1.4 替换算法重构模式的识别

1.4.1 替换算法模式介绍

在对代码的更新的过程中，随着对问题有了更多的理解，或者相关代码有所改动，往往需要对某个函数进行替换，这种做法在重构模式中即可称为替换算法(Substitute algorithm)。一般情况下，其运行过程是在某个函数的函数头不变的情况下，用新函数体替换此原函数的函数体。

图 4 的代码来自于 maven(https://github.com/apache/maven/) 版本为 01ae529 的 RemoteSnapshotMetadata.java 文件，可以看到函数 *getExpandedVersion*(*Artifact artifact*) 的原函数体被移除(行 94)，被替换成了新的函数体(94~95 行)，

这就是一个典型的替换算法重构模式实例。

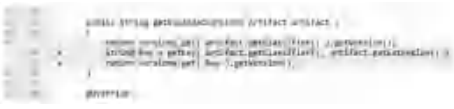

图 4　替换算法重构模式示例

Fig. 4　Substitute algorithm refactoring pattern example

1.4.2　基于抽象语法树和代码变更块对替换算法模式进行识别

对于替换算法模式而言，可归纳为 2 个特点，对此可概述为：

(1)原函数和变更后函数的函数头相同，即原函数的函数头没有发生改变。

(2)原函数的函数体和变更后函数的函数体发生了改变。

设有原文件 $file_l$ 变更后文件 $file_r$，先使用文本型代码差异化分析工具获取这 2 个文件的变更代码块集，替换算法发生在一个 hunk 之中，故对此 hunk 集的所有 hunk 进行筛选，筛掉只有添加部分或删除部分的 hunk，获取 hunk 集(设为 H)，用同样的方法获取 H 的删除部分和添加部分集(设为 H_a，H_d)。

使用语法树分别获取 $file_l$ 和 $file_r$ 在 H 中包含的函数节点 M_l，M_r，若这 2 个节点的函数头相同，并且 2 个节点的函数体分别就是删除部分和添加部分，则满足替换算法模式的特点(1)。

在满足第一个特点的前提下，若 M_l 和 M_r 节点的函数体不同，则满足替换算法模式的特点(2)，可以使用代码相似度检测技术进行检测，判断 M_l 和 M_r 节点的函数体是否不同。

1.4.3　算法

此算法输入变更前后 2 个版本的源文件 $file_l$ 和 $file_r$，输出存在替换算法模式的函数节点(MethodNode)元组的集合 $M_s = \{ < M_1, M_2 > \}$，M_1，M_2 是 2 个函数节点，符合替换算法重构模式。研究给出算法的伪代码详见如下。

输入：变更前后 2 个版本的源文件 $file_l$ 和 $file_r$

输出：存在替换算法模式的函数节点集合 M_s

$H_s \leftarrow getAllHunks(file_l, file_r)$

$H \leftarrow \varnothing$

for each $h_s \in H_s$

　if (h_s.addpart $\neq \varnothing$)&&(h_s.delpart $\neq \varnothing$)

　　$H \leftarrow H \cup h_s$

　end if

end for

$(H_d, H_a) \leftarrow split(H)$

$t_l \leftarrow createAST(file_l)$

$t_r \leftarrow createAST(file_r)$

$M_1 \leftarrow getAllMethods(t_l)$

$M_2 \leftarrow getAllMethods(t_r)$

$(M_l, M_r) \leftarrow match(H, M_1, M_2)$

$M_s \leftarrow \varnothing$

for each $(m_l, m_r) \in (M_l, M_r)$

　if (! $checkSimilar((m_l, m_r))$

　&&$checkMethodHead((m_l, m_r)))$

　　$M_s \leftarrow M_s \cup (m_l, m_r)$

　end if

end for

算法第 1 行是通过文本型差异分析方法获取 2 个文件的所有 hunk 集 H_s，3 ~ 7 行将 H_s 里的所有存在添加部分和删除部分的 hunk 筛选出放入集合 H，第 8 行是将 H 按照删除部分、还是添加部分分离为 (H_d, H_a)，第 13 行 $Match(H, M_1, M_2)$ 函数的功能和上一算法伪代码的第 11 ~ 22 行的功能一致，即实现通过 hunk 获取相关函数节点；第 15 ~ 19 行实现对函数节点 m_l，m_r 函数体是否不同和对其函数头是否相同的判断，即是否是替换算法重构模式，并将符合替换算法重构模式的函数节点元组 (m_l, m_r) 放置入 M_s 中，其中 $checkSimilar()$ 函数用于检测 m_l，m_r 函数节点的函数体是否相似，若不相似说明函数体已经被替换，本文使用编辑距离算法来计算相似度；$checkMethodHead((m_l, m_r))$ 函数用于检测 m_l，m_r 这 2 个节点的函数头是否相同，其方法是比较函数节点中的数个属性(MODIFIERS，RETURN_TYPE，NAME，PARAMETER，THROW_EXCEPTIONS)是否都相同，至此得到最终结果。

2　算法的实现及验证

本文使用 Java 编程语言来实现这 2 个算法，采用面向行的文本差异化分析工具 Gun Differ (http://www.gnu.org/software/diffutils/) 获取 Hunk 集，采用 eclipse jdt parser 来获取源文件的抽象语法树，在 4 个开源项目上进行了实验验证。研究内容具体如下。

2.1　数据来源

获取了 4 个开源项目的数据集用于对算法的验证，对此拟做出阐析分述如下。

(1) eclipseJDTCore (https://github.com/eclipse/eclipse.jdt.core) 开源项目，这是一个针对 Java 的集

成开发环境,此次实验获取时间段为 2001/06/23 ~ 2013/10/16 的更新数据。

(2) maven(https://github.com/apache/maven/) 开源项目,这是一个通过信息描述来管理项目的构建、报告和文档的一个开源的管理工具,此次实验获取了时间段为 2003/09/02 ~ 2014/01/29 的更新数据。

(3) jEdit (https://github. com/linzhp/jEdit - Clone),这是一个跨平台的文本编辑器,此次实验获取了时间段为 1998/09/27 ~ 2012/08/08 的更新数据。

(4) google _ guice (https://github. com/google/guice/)是一个轻量级的依赖注入容器。数据的获取时间段为:2006/08/23 ~ 2013/12/12。

2.2 实验结果和分析

本文对来自于上述 4 个开源项目的数据集进行了实验,同时列出了内联函数模式检测算法所检测到的内联函数模式的个数见表 1,并以人工检测的结果为基准获得了其查全率和查准率。

表 1 内联函数识别实验结果

Tab. 1 Result of Inline method recognition experiment

项目名称	检测到的内联函数数	其中为真的个数	人工检测内联函数数	查全率/%	查准率/%
eclipse	8	6	7	86	75
maven	17	12	15	80	71
google- guice	9	7	10	70	78
jEdit	10	8	11	73	80

可以看到,内联函数识别实验结果显示查全率在 70% ~ 86% 之间波动。查准率在 71% ~ 80% 之间波动。

接下来,研究列出了替换算法重构模式识别算法所检测到的替换算法模式的个数见表 2,同样以人工检测的结果为基准获得了其查全率和查准率。

表 2 替换算法模式识别结果

Tab. 2 Result of Substitute algorithm recognition experiment

项目名称	检测到的替换算法数	其中为真的个数	人工检测替换算法数	查全率/%	查准率/%
eclipse	70	59	80	74	84
maven	138	113	150	75	82
google- guice	93	70	98	71	75
jEdit	121	87	109	80	72

可以看到,在替换算法模式识别结果中,对这 4 个开源项目的识别。实验得出,查全率在 71% ~ 80% 之间波动。查准率在 72% ~ 84% 之间波动。

分析可知,查全率和查准率没有达到 100% 的一个原因在于 Gun Differ 对于大粒度的函数的改动容易划分为多个 hunk,使得各个 hunk 只包含这个大粒度函数的一部分,在算法的运行过程中被自动筛去,以后可以增加 hunk 合并机制,让某个大粒度函数的多个 hunk 合并为一个 hunk,让一个 hunk 就能够包含此大粒度函数,从而为这 2 个算法所识别。

3 结束语

本文提出了基于代码变更块和抽象语法树的对内联函数和替换算法重构模式自动识别的算法,在 4 个开源项目上进行了实验,表明其有较高的准确率,这种基于代码变更块和抽象语法树的算法也可以对其它有着代码移动情况的代码重构模式,如移动字段(Move field)、抽取类(Extract Class)等进行识别,后续工作仍需开发出针对大粒度函数的多个 hunk 合并机制以提高算法的准确率,在更多的数据集上验证其有效性,进而将其应用到其它类型的重构模式识别上。

参考文献

[1] FOWLER M. 重构—改善既有代码设计[M]. 北京:中国电力出版社,2003.

[2] 于冬琦,彭鑫,赵文耘. 使用抽象语法树和静态分析的克隆代码自动重构方法[J]. 小型微型计算机系统,2009,30(9):1752-1760.

[3] 冯江辉. 基于 K-最近邻的 C 克隆代码重构方法研究[D]. 哈尔滨:哈尔滨工业大学,2011.

[4] 刘伟,胡志刚,刘宏韬. 单例模式导向的源代码自动重构研究[J]. 小型微型计算机系统,2014,35(12):2664-2669.

[5] 张杨,张冬雯,仇晶. 面向 Java 锁机制的字节码自动重构框架[J]. 计算机科学,2015,42(11):84-89.

[6] FOKAEFS M,TSANTALIS N, STROULIA E, et al. Identification and application of extract class refactorings in object - oriented systems[J]. Journal of Systems and Software, 2012, 85(10): 2241-2260.

[7] WEISSGERBER P, DIEHL S. Identifying refactorings from source-code changes [C]// Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE´06). Tokyo, Japan:IEEE,2006: 231-240.

[8] PRETE K, RACHATASUMRIT N, SUDAN N, et al. Template-based reconstruction of complex refactorings [C]//ICSM ´10 Proceedings of the 2010 IEEE International Conference on Software Maintenance.Washington, DC, USA :IEEE, 2010:1-10.

[9] 刘阳,刘秋荣,刘辉 . 函数抽取重构的自动检测方法[J]. 计算机科学,2015,42(12):105-107.

[10] 钟林辉,黄小明,薛良波,等. 基于版本的多重软件重构自动检测技术研究[J]. 江西师范大学学报(自然科学版),2018,42(5):464-469,472.