文章编号：2095-2163(2020)04-0013-05 中图分类号：TP391 文献标志码：A

语句分裂变更模式的定义及识别

段卫华，杨春花
（齐鲁工业大学 计算机科学与技术学院，济南 250353）

摘 要：替换算法是一种常见的函数层面的重构手法，而日常代码变更中更为多见的是一种将替换算法变更思想应用于更为广泛的代码语句层面的代码变更模式。本文将这种代码变更模式命名为语句分裂变更模式，在对存在该模式的代码变更实例进行人工分析的基础上，给出了该模式的定义，并设计了一种对该模式的识别算法。该算法根据语句分裂变更模式的语法特征对其进行识别，并在 4 个开源项目上进行了实验验证，实验结果表明了该算法具有较高的识别准确率。

关键词：软件演化；模式识别；抽象语法树；代码变更块

Definition and recognition of statement splitting change pattern

DUAN Weihua, YANG Chunhua
(School of Computer Science and Technology, Qilu University of Technology, Jinan 250353, China)

【Abstract】 Substitute algorithm is a common method of function level refactoring, but in daily code change, it is more common to apply the idea of substitute algorithm change to a broader code statement level code change pattern. In this paper, the code change pattern is named statement splitting change pattern. Based on the analysis of the code change cases with the pattern, the definition of the pattern is given, and an algorithm for identifying the pattern is designed. The algorithm is based on the syntactic characteristics of the split change pattern, and is verified by experiments on four open source projects. The experimental results show that the algorithm has a high recognition accuracy.

【Key words】 Software evolution; Pattern recognition; Abstract syntax tree; Hunk

0 引 言

现代软件的开发一般基于版本管理系统设计实现，软件工程师为了实现开发和维护的任务，每天都会提交大量的代码变更，而这些变更往往会遵循一定的修改模式，如重构（Refactoring）、缺陷修复（bug-fixing）和一些装饰型（cosmetic）的修改模式等等。由于变更的日志描述往往反映不了变更的真正行为[1]，因此对代码变更的理解还需要人工审查变更代码来确定。日常代码修改往往存在多种变更模式的重叠，如果将代码变更模式进行识别，则可将各种模式的变更代码区别开来，减少这些代码之间的耦合，从而方便分析和理解软件的演化过程。

对于典型的重构模式，M.Fowler[2] 对其进行了系统的研究。重构的目的是优化代码中存在的一些不合理结构（即代码异味），而对代码异味的检测主要有人工检测[3-4]和自动化检测[5]2 种方式。对于缺陷修复（bug-fixing）较为重要的研究方向就是对缺陷的检测。缺陷的检测方法有：基于信息检索技术的缺陷定位方法[6]和利用项目的历史信息改进的缺陷检测方法[7]等。装饰型修改的目的一般是规范代码的编写，所以不会影响整个软件的作用，对其研究包括标识符重命名[8]和编程风格[9]的研究等。

替换算法是将函数体中复杂的功能块进行封装，从而使函数体结构更为简洁清晰的重构手法。但在开发实践中，常见将一条语句分裂成多条语句的代码变更模式，即语句分裂变更模式。该模式在代码语句层面利用封装思想优化代码语句结构。然而有些语句分裂变更类似于重构，不会改变语句行为，而有时又会像缺陷修复模式一样改变语句行为。因此，不能将该模式简单归类为重构模式或是缺陷修复模式。

本文对语句分裂变更模式进行了定义及识别研究，通过对大量的语句分裂变更模式实例进行人工观察，分析总结出了 2 种该模式常见的变更形式，并将其定义命名为拆分形式和替换形式，同时还设计并实现了 2 种形式的识别算法。

1 语句分裂变更模式

代码变更是指一个源代码文件修改前后的两个版本的差异。目前，对变更的提取基本是借助于差

基金项目：山东省自然科学基金面上项目（ZR2017MF056）。

作者简介：段卫华（1993-），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：2020-03-02

异化分析工具来实现的,对语句分裂变更模式的定义是基于差异分析工具的输出结果而制定的。

1.1 差异分析工具

目前使用的差异分析工具主要有:文本差异分析(Textual - Differencing)和树差异分析(Tree - Differencing)。

1.1.1 文本差异分析工具

文本差异分析是将更改前后的源文件视为字符串,通过计算公共子序列来判别发生变动的文本,并将这些文本信息以及对应的行号以代码变更块(hunk)为单位输出。所以,hunk 实际上就是不同行的集合,其包括旧版本源文件的删除行和新版本源文件的增加行,也可以只包含删除行部分或增加行部分。

常见的文本差异分析工具是 diff。其中, GNU-Diff[10] 推出了“合并格式”的 Diff,2 个文件的上下文合并在一起显示,并由“-”表示变动前的文件,“+”表示变动后的文件。hunk 实例如图 1 所示,hunk 中左边行号 88 行为旧版本的删除行,右边行号 88-89 则为新版本的增加行。

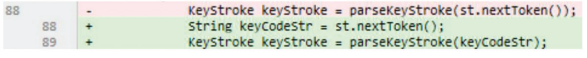

图 1 hunk 示例

Fig. 1 hunk example

1.1.2 树差异分析工具

抽象语法树(abstract syntax tree, AST)是一种源文件语法结构的树状表现形式,而树中的每一个节点都是由源代码中的语句映射而来的。通过对抽象语法树的遍历,可以准确获得源代码中的每一个节点信息。

树差异分析是通过对比修改前后源文件所对应的抽象语法树,来获取结构化的变更信息,这些信息更有利于变更的分析。最著名的基于抽象语法树差异分析工具是 Change-Distiller[11],其可以获得一个文件变更前后所有的代码变更类型。这是 Fluri 等人编写的一个 Tree differ 算法,对变更前后抽象语法树进行对比,获取分类的变更。同时,也可以区别多种方法类型的变化或类等级上的变化。

1.2 语句分裂变更模式的定义

语句分裂变更模式包含拆分和替换两种形式,而对于这 2 种形式的研究是以文本差异分析工具输出的代码变更块(hunk)为单位的,所以要对 hunk 进行定义。

定义 1 代码变更块 hunk 可以用四元组 h=<L-,L+,R-,R+>的形式来表示。其中, L-和 L+分别代表删除行和增加行的文本内容, R-和 R+分别代表删除行和增加行的行号范围。

基于上述 hunk 的定义并结合 2 种形式的文本特征,来定义拆分形式和替换形式的语句分裂变更模式。

定义 2 拆分形式的语句分裂变更模式可以用三元组<h,s,s'1~s'n>的形式表示。在 h 中,∃语句 s∈L-∧语句 s'1~s'n∈L+。在此需满足:

(1)s 是一条存在传递赋值的赋值语句。

(2)s'1~s'n 是由 s 拆分而来的多条赋值语句。拆分形式实例如图 2 所示。其中,删除行 872 行为传递赋值语句,增加行 872-873 行为拆分后得到的两个赋值语句。

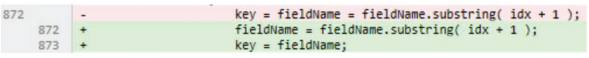

图 2 拆分形式

Fig. 2 Split form

定义 3 替换形式的语句分裂变更模式是一个四元组<h,s,v,s'>。在一个 h 中,∃语句 s∈L-∧语句 v∈L+∧语句 s'∈L+。此时需满足:

(1)v 是一条变量赋值语句。

(2)s'是由语句 v 中的变量替换语句 s 中部分内容或直接添加到语句 s 中而得来的。

替换形式实例如图 3 所示。其中:删除行 398 为语句 s;增加行 402 为变量赋值语句 v,404 行为语句 v 中的变量 extensionComponents,替换语句 s 中 getPlugExtensionComponents(plugin)而得到的语句 s'。

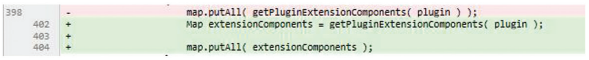

图 3 替换形式

Fig. 3 Replacement form

1.3 语句分裂变更模式的抽象语法树节点变化特征

语句分裂变更模式的识别,要借助于抽象语法树节点变化特征的分析。抽象语法树的每一层结构被称做节点(Node),一个 AST 可以由单一节点或多个节点构成,每个节点包含了 type(类型)和 location(位置信息)。抽象语法树的节点类型分为:语句(Statement)类、表达式(Expression)类、声明(Declaration)类等。节点的位置信息包括起始位置(被解析的源区域的第一个字符的位置)和结束位置(被解析的源区域之后的第一个字符的位置)。

假设在一个 hunk: <L-,L+,R-,R+>中,删除行 L-和增加行 L+所包含的全部节点集合分别为 N-与 N+。

根据定义 2 的拆分形式存在以下语法特征:

（1）N-中一定存在至少一个赋值语句类型的节点，且其包含赋值语句类型的子节点（对应上述定义 2 中的语句 s）。

（2）N+中一定存在至少 2 个赋值语句类型的节点（对应上述定义 2 中的语句 s'1～s'n），并且这些节点的子节点要包含（1）中赋值语句节点的所有子节点，以确保 N+中的多个赋值语句是由 N-中的赋值语句拆分得来的。

根据定义 3 的替换形式存在以下语法特征：

（1）N+比 N-多一个变量赋值节点（对应上述定义 3 中的语句 v）；

（2）N-与 N+存在相同类型的语句节点（对应上述定义 3 中的语句 s 与 s'），并且 N+中的语句节点的子节点信息中要包含（1）中变量赋值节点的变量名信息。

2　语句分裂变更模式识别

本文根据语句分裂变更模式所呈现出的语法特征，设计了该模式的识别算法。该算法首先在文本层面以 hunk 为单位，提取出一次更改前后两个版本源文件之间所有的变更信息；经初步筛选后，再根据 hunk 所包含的语法树节点信息逐一分析是否包含了语句分裂变更模式的 2 种常见形式；最后将包含语句分裂变更模式拆分形式的 hunk 以三元组<h，s，s'1～s'n>的形式存入集合 *PS*，将包含替换形式的 hunk 以四元组<h，o-，m，c>的形式存入集合 PR。算法部分伪代码如下所示：

输入：同一文件的一次更改前后 2 个版本的源文件 Fileold 和 Filenew。

输出：拆分形式语句分裂变更模式的 hunk 集 PS 与替换形式的语句分裂变更模式的 hunk 集 *PR*。

```
1   H←diff(Fileold, Filenew)
2   H'←filter(H)
3   (A,A')←parser(Fileold, Filenew)
4   for eachh ∈ H'
5     (N-,N+)←fetchNodes(h,A,A')
6     if true←checkSplit(N-,N+)  then
7       PS←<h,s,s'1～s'n>
8     else V←findVarDefine(N-,N+)
9         if V≠Ø
10            M←getVarName(V)
11            (O-,O+)←getSameTypeNode
          (N-,N+)
12            C←getChildNodes(O+)
13            if ∃m∈M,c∈C true←contains
          (c,m)
14              PR←<h,o-,m,c>
15          end if
16        end if
17      end if
18  end for
```

算法第 1 行使用 diff 工具获得两个源文件的 hunk 集合 *H*；第 2 行使用 *filter*（）函数得到整体范围较小的 hunk 的集合 *H'*；第 3 行使用 parser 工具分别构造两个目标文件的抽象语法树（*A*，*A'*）；第 5 行使用 *fetchNodes*（）函数获得 hunk 的删除行和增加行包含的节点信息集合 *N* - 和 *N* +；第 6、7 行使用 *checkSplit*（）函数识别拆分形式，并以三元组<h，s，s'1～s'n>的形式存入集合 *PS* 中；第 8～16 行判断替换形式。对判断替换形式伪代码部分的说明：首先利用 *findVarDefine*（）函数获取 hunk 中变量声明节点 *V*，并通过 *getVarName*（）函数获得新声明变量的变量名集合 *M*，使用 *getSameTypeNode*（）函数找出与 *N* - 与 *N* + 中相同类型节点的集合（*O* -，*O* +）；第 12 行使用 *getChildNodesAsString*（）函数获得集合 *O* + 中每个节点的子节点信息集合 *C*；第 13 行使用 *contains*（）函数判断字符串 *C* 是否至少包含一个集合 *M* 中的字符序列，以确定分裂后的原句中存在新变量的变量名；第 14 行满足上述条件的 hunk 以四元组<h，o-，m，c>的形式存入集合 *PR* 中。此算法的输出结果中，变量 o-、m 和 c 分别替代了定义 3 中的语句 s、语句 v 和语句 s'。变量 *h* 表示一个四元组<L-，L+，R-，R+>；变量 *o* - 是集合 *O* - 中的元素，代表语句 s 的节点信息；变量 *m* 是集合 *M* 中的元素，代表语句 v 的新变量变量名；语句 s 被 m 替换部分内容后变为语句 s'；变量 *c* 是集合 *C* 中的元素，代表语句 s'的节点信息。

3　算法实现与验证

该算法基于 Java 编程语言实现，算法使用文本差异分析工具 Gnu Differ 来提取变更，并使用 Java Parser 工具构造 2 个源文件的抽象语法树，最后在 4 个开源项目上进行了验证。具体研究内容如下。

3.1　数据源

利用 Minigit3 工具从 Git Hub 项目托管平台中抓取一定时间内 4 个开源项目的版本变更历史数据作为源数据集。

（1）eclipseJDTCore4 开源项目，是针对 Java 的集成开发环境。

（2）mavens 开源项目，是通过信息描述来管理项目的构建、报告和文档的开源管理工具。

（3）jEdit6 开源项目，是跨平台的文本编辑器。

(4) google-guice7 是轻量级的依赖注入容器。

表 1 列出了上述 4 个源数据集各自所包含更改提交的次数和 4 个项目的文件数,每次更改的提交会涉及到对多个文件的更改。

表 1 4 个源数据集的提交更改的次数和文件数

Tab. 1 Number of commit changes and files for four source datasets

项目名称	提取时间段	提交次数	文件个数
J Edit	1998/09~2012/08	6 275	1 416
maven	2003/09~2014/01	8 845	3 051
guice	2006/08~2013/12	1 004	1 172
Eclipse	2001/06~2013/10	19 123	7 346

3.2 实验结果分析

3.2.1 算法对人工筛选 hunk 集的识别

经人工筛选 4 个源数据集,并且得到包含语句分裂变更模式的 hunk 集后,用识别算法去识别人工筛选出的 hunk 集,目的是验证识别算法的准确率,实验结果见表 2。

表 2 识别算法识别人工数据集结果

Tab. 2 Experimental results of recognition algorithm identifying artificial data set

项目	人工筛选个数		算法识别个数		准确率/%
	拆分	替换	拆分	替换	
J Edit	0	101	0	88	87
maven	1	64	1	53	83
guice	0	21	0	16	81
Eclipse	5	310	3	278	89

由表 2 可见,该识别算法在识别人工筛选数据集时的准确率在 81%~89%之间波动。由于在人工筛选过程中,包含语句分裂变更模式的 hunk 行数一般较少,所以识别算法中筛掉了一些行数较多的 hunk,这样做可提升算法的效率,但导致识别算法的准确率没有达到最高水平。

3.2.2 算法对 4 个源数据集的识别

在上述实验结果的基础上,利用识别算法对 4 个源数据集进行识别,并将输出的 hunk 集进行人工筛查,去除其中可能存在的非语句分裂变更模式的 hunk。表 3 列出了识别算法所检测到的 hunk 的个数,以及人工筛查后留下的真正包含语句分裂变更模式的 hunk 的个数及准确率。

表 3 识别算法识别 4 个源数据集的实验结果

Tab. 3 Experimental results of identification algorithm identifying four source data sets

项目	算法识别个数		为真的个数		准确率/%
	拆分	替换	拆分	替换	
J Edit	3	124	3	98	80
maven	4	75	4	52	71
guice	0	37	0	29	78
Eclipse	12	386	10	284	76

由表 3 可见,识别算法识别 4 个源数据集的准确率在 71%~80%之间波动。

对结果分析可知,识别算法的输出结果中存在非语句分裂变更模式的 hunk 的原因主要是:在识别替换模式时,要查找 hunk 的删除行和增加行中相类型的语句,并且增加行中的语句要包含新声明的变量,但类型相同并不一定能保证增加行的语句是由删除行的语句变化得来的。即使增加行的语句包含了新变量,也不能说明其与删除行语句本质上是同一条语句,所以并不能保证一定是替换形式,这种情况多发生于 return 语句的变更中。

图 4 是一个算法输出的 hunk。其中删除行和增加行包含了同种类型的 return 语句,且 1 841 行的 return 语句包含了 1 839 行新声明的变量 clone。但从 clone 的赋值内容来看,显然与 1 832 行 return 语句的内容不同。因而算法识别出的这两个相同类型的 return 语句,从本质上并不是同一条语句。所以,此 hunk 不能作为语句分裂变更模式的实例。

```
1832  -       return new MavenProject( this );
1839  +       MavenProject clone = (MavenProject) super.clone();
1840  +       clone.deepCopy( this );
1841  +       return clone;
```

图 4 特例展示

Fig. 4 Special case display

3.3 实验结论

(1) 由语句分裂变更模式的 2 个形式进行比较可知,该模式大多情况都是以替换形式出现的,拆分形式只为极少数。

(2) 由各项目所包含的语句分裂变更模式的 hunk 个数对比可知,变更提交次数和文件数越多的项目,包含该模式的 hunk 就越多。说明语句分裂变更模式的出现次数,与项目属性和更改提交者的个人习惯等因素关系较小。项目规模越是庞大,该模式出现的次数就越多。

4 结束语

本文通过人工筛选的数据归纳,总结了语句分裂变更模式的特征,制定了该模式的定义,并且设计了基于该模式语法特征和以 hunk 为识别单位的语句分裂变更模式的识别算法。分别在人工筛选的数据集和 4 个开源项目的源数据集上进行了算法的验证,并由实验结果得到了一些结论。后续工作还需要对识别出来的语句分裂变更模式进行更为细致的分类,即对语句分裂变更过程中各个部分伴随的一些变化进行分析,以便更加深入的了解语句分裂变更模式;另外还需要考虑到行数较多的 hunk 或多个

(下转第 21 页)