文章编号:1671-4598(2018)04-0070-04 DOI:10.16526/j.cnki.11—4762/tp.2018.04.019 中图分类号:TP311.5 文献标识码:A

Java 自动化基本路径测试技术研究

冯俊池，赵 颖，连 尧，尹党辉，安丰亮
（后勤科学与技术研究所，北京 100071）

摘要：针对 Java 单元测试自动化程度和测试效率较低的问题，对基于 Java 程序的基本路径测试方法进行研究，提出了基于 Java 代码的基本路径生成方法和程序插桩方法，给出了插桩节点和控制流图节点的定义；首先，通过对 Java 源代码进行分析，构建程序的控制流图，进而对控制流图进行遍历生成基本路径集合；然后，对被测程序进行插桩，以获取程序的执行路径，插桩过程中保持节点和基本路径中的节点一致，使得插桩后的被测程序执行时得到的路径能够和基本路径集合进行自动化比对；最后，通过以测试数据为输入执行被测程序，对执行路径和基本路径进行比较，判断测试数据集对基本路径的覆盖度；通过实验，验证了所提出方法的有效性。

关键词：基本路径；控制流图；单元测试；程序插桩

Research on Automated Basis Path Testing of Java

Feng Junchi, Zhao Ying, Lian Yao, Yin Danghui, An Fengliang
(Institute of Logistic Science and Technology, Beijing 100071, China)

Abstract: The level of Java unit testing automation is low, and so is the testing efficiency. The basis path testing method based on Java program is studied. The basic path generation method based on Java source code and program instrumentation method is provided. The definition of instrumentation node and control flow graph node is given. Firstly, the control flow graph of program under test is built based on the analysis of Java source code, and then the control flow graph is traversed to generate the basis path set. Secondly, the program under test is instrumented to obtain the program execution path. The instrumentation node is consistent with the node of basis path, so the path obtained during execution can be compared with the basic path set automatically. Lastly, run the program under test with the test data as input, and compare the execution path with basis path to compute the basis path coverage of test data set. The effectiveness of the proposed method is verified by experiments.

Keywords: basis path; control flow graph; unit testing; program instrumentation

0 引言

单元测试是保证软件质量的重要手段[1]，能够在前期有效发现代码中存在的问题，避免将缺陷引入软件。在单元测试中，主要的覆盖准则有语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、条件组合覆盖和路径覆盖，其中路径覆盖是最强的覆盖准则，通过设计测试用例覆盖程序中所有可能的路径，发现其中存在的缺陷和错误。但当路径数量很大，尤其是存在循环时，很难做到完全覆盖，需要把路径数量压缩到一定程度。基本路径测试方法[2]在一定程度上解决了这一问题，通过只针对一组合理的路径即基本路径集合进行覆盖测试，而不是穷举所有路径，有效避免了路径爆炸问题。

目前针对基本路径集求解已有很多研究[3-5]，如路径字符串组合算法[3]、基于图深度优先搜索算法等[4]，基于状态图的测试路径自动生成算法[5]等。现有方法主要以控制流图为输入来生成基本路径集合[6-8]，减少了人工分析得到基本路径集合的工作量。为了减少由代码生成控制流图以及执行路径与基本路径集合比对的工作量，本文提出了针对 Java 源码的路径测试自动化方法，首先通过对 Java 代码的分析构建控制流图和实现程序插桩，保持控制流图和代码插桩中节点的一致性，然后通过控制流图生成基本路径集合，最后执行插桩后被测程序判断测试数据对基本路径集合的覆盖程度。

收稿日期:2017-10-18； **修回日期**:2017-11-09。

作者简介:冯俊池(1990-),男,山东临清人,硕士,助理工程师,主要从事软件质量方向的研究。

1 控制流图定义与构造

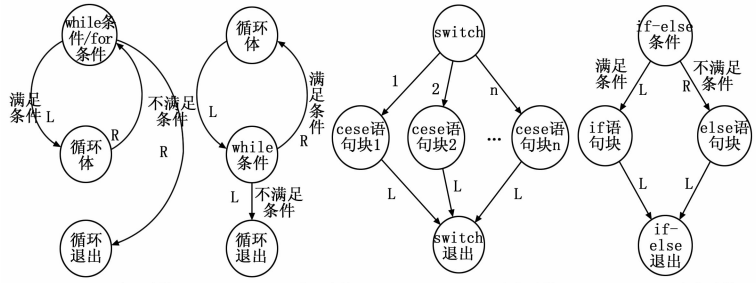
基本路径集通过分析被测程序的控制流图来得到，因此在基本路径测试过程中，首先要根据程序流程来绘制控制流图。控制流图是描述程序流程的一种图示方法，以结点和边的方式表示了一个程序执行过程中会遍历到的所有路径。

1.1 控制流图结构

将控制流图涉及到的分支结构主要划分为 4 种：while/for 循环，do-while 循环，if-else 分支，switch-case 分支，如图 1 所示。其中每条有向边由父节点指向子节点，边上标注的“L”和“R”分别代表子节点是父节点的左子节点和右子节点，由于 switch 节点可以有多个子节点，因此其子节点按 1 到 n 的顺序进行标注。

1.2 数据结构

控制流图中的节点主要分为三类，数据结构如图 2 所示。基本节点类型 BranchNode 代表了普通节点，属性 id 的值为正整数，每个节点的 id 是唯一的，属性 visited 为布尔值，用来在路径遍历时标记节点是否被访问过，属性 leftChild 和 rightChild 分别记录了节点的左右子节点。WhileForBranchNode 为 while/for 循环结构和 do-while 循环结构的条件判断节点，相比普通节点增加了 exitNode 属性，记录了循环的退出节点，用于在循环遍历过程中退出循环。SwitchBranchNode 为



(1) while/for循环结构　(2) do-while循环结构　(3) switch-case分支结构　(4) if-else分支结构

图 1　分支结构

switch－case 分支结构的 switch 节点，与普通节点不同，switch 节点可能包含不止两个子节点，因此采用属性 children 记录子节点列表。

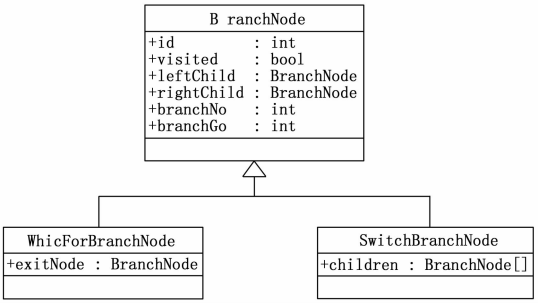


图 2　控制流图节点的数据结构

每个节点具有两个属性 branchNo 和 branchGo，分别记录了该节点所属上一层分支节点和自分支节点后的分支编号，对于非分支节点的子节点，该处值均取－1，假设 id 为 1 的节点为分支节点，下面有两个分支，分别有节点 2 和 3，则节点 2 和 3 的（branchNo，branchGo）属性值为（1，0）和（1，1），若节点 1 下不止两个分支，则子节点的 branchGo 值依次取 0，1，2 等整数。二元组（branchNo，branchGo）记录了节点在程序流图所处的位置，一条路径即可由（branchNo，branchGo）的列表标识。

1.3　控制流图生成

由被测程序生成控制流图主要分为 3 个步骤：词法分析、语法分析和控制流图构造，如图 3 所示。

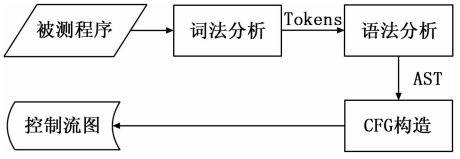


图 3　控制流图生成

词法分析和语法分析采用 Antlr 工具辅助实现，Antlr 可以根据用户提供的语法文件自动生成相应的词法/语法分析器。通过词法分析，将被测程序解析为离散的字符组，即 Tokens，包括关键字、标识符、符号和操作符等，供语法分析器使用。语法分析器将 Tokens 组织起来，转换为 AST（Abstract Syntax Tree，抽象语法树）。树上的每个节点都表示源代码中的一种结构，为下一步程序控制流图的构造提供信息。

控制流图构造通过对 AST 进行遍历实现，将程序中的语句划分为基本块，基本块是顺序执行的语句序列，其中只有一个入口和出口，针对 Java 程序，基本块的入口语句只能是程序的第一条语句，转移语句（break、continue 等）的目标语句或紧跟在条件转移语句（if－else、while、for、switch－case 等）之后的语句。一条入口语句到下一条入口语句间的所有语句构成一个基本块。基本块即对应于控制流图中的节点，基本块之间的转移关系为控制流图中的边，通过基本块的划分和转移关系的确定，即可得到被测程序的控制流图。

在控制流图生成过程中，对分支节点的 id 从 1 进行编号，其子节点称为分支子节点，也即 branchNo 和 branchGo 属性非负值的节点，通过记录分支子节点的（branchNo，branchGo）属性值即可唯一标识一条路径。

2　基本路径生成

2.1　算法思想

通过深度优先搜索遍历控制流图，在不存在循环的情况下，每遇到一个分支节点，则复制当前的子路径，并针对后继的每个节点继续进行搜索；由于循环的存在，需要避免搜索算法陷入死循环无法结束，为每个节点设置访问标志，当再次遍历到已访问过的节点时，有如下 3 种情况：

1）while/for/do－while 的判断节点：直接访问退出节点，因已访问过该节点的分支，本次直接退出循环，遍历后面的节点。

2）分支节点：选择其中一个节点继续，出现该种情况原因是存在分支嵌套或分支并列，已访问过该节点的分支，本次遍历不再访问所有分支，避免产生冗余路径。

3）其他节点：选择其子节点继续，出现该种情况原因是存在分支嵌套或分支并列。由于 while/for 循环体的最后一个节点左侧子节点为空，右侧子节点指向循环判断语句，因此需要判断当前节点是否是 while/for 循环体，若是，则选择右侧节点继续遍历，否则，选择左侧节点继续遍历。

2.2　基本路径生成算法

算法为 visitPrimePath（currentNode，path），其中 currentNode 代表当前节点，初始输入为程序入口节点，path 用于记录路径，算法步骤如下：

步骤 1：如果 currentNode 是结束节点，终止，否则转到步骤 2；

步骤 2：如果 currentNode.visited 为 true，执行步骤 2.1，否则，转到步骤 3；

步骤 2.1：如果节点类型为 while/for/do－while 循环的判断节点，取循环的结束节点 currentNode.exitNode，执行 visitPrimePath（currentNode.exitNode，path），否则执行步骤 2.2；

步骤 2.2：如果类型为 switch，对第一个子节点 firstChildNode 执行 visitPrimePath（firstChildNode，path），否则执行步骤 2.3；

步骤 2.3：若 currentNode 是分支子节点则将其加入 path，如果节点的左子节点 leftChild 不为 null，执行 visitPrimePath（currentNode.leftChild，path），否则，执行 visitPrimePath（currentNode.rightChild，path）；

步骤 3：若 currentNode 是分支子节点则将其加入 path，令 currentNode. visited＝true，执行步骤 3.1；

步骤 3.1：如果是 switch 类型，依次对每个子节点 childenNode 执行 visitPrimePath（currentNode. exitNode，path），否则执行步骤 3.2；

步骤 3.2：判断左子节点是否为 null，如不为 null，执行 visitPrimePath（currentNode. leftChild，path）；判断右子节点是否为 null，如不为 null，执行 visitPrimePath（currentNode. rightChild，path）。

2.3 算法证明

算法功能正确性证明如下：

1）每条程序路径都是独立路径，即每条路径都至少包含一条其他路径未包含的边。

在程序流图遍历过程中，第一次遇到分支节点时，算法分别选择不同的分支继续遍历；当再次遇到访问过的节点时，则当前路径已包含了其他路径未包含的边，则在当前节点只选择其中一个继续进行遍历。因此能够保证生成的每一条路径包含独立的边，即每条程序路径都是独立路径。

2）程序中所有的边都被访问。

由算法 visitPrimePath（currentNode，path）第 3 步可以看出，针对 switch 节点，其所有的子节点都被访问，即所有分支均被访问；针对其他节点，如果子节点不为空，则继续遍历，即访问了当前节点所有分支。因此，程序中所有的边都被访问。

3）程序中的所有的、不属于基本路径集的路径都可由基本路径集中的路径经过线性运算得到。

对于一条需要通过基本路径线性运算得到的路径，称之为目标路径。分为两种情况来讨论：

第一种情况，目标路径中不包含重复片段，如图 4 所示，左侧为目标路径，右侧路径 1 到路径 $2n-1$ 来自生成的路径集合。目标路径由 n 条边组成，路径 1 包含了片段 a，路径 3 包含了片段 b，由第 1）、2）的证明可知，路径 1 和 3 是来自于生成的路径集合（这里假设不存在一条路径包含 a 和 b 两个片段的情况，即路径 1 和 3 不同），则存在路径 2 和路径 3 有着相同的子路径 x 且节点 1 后的路径不同，即节点 1 为分支节点，由算法可知，分支节点前的所有不同子路径必经过同一条子路径到达结束节点，对于路径 1 和路径 2，节点 1 前的子路径不同，必然会存在一条路径 2 使得节点 1 后的子路径相同，路径 1、2、3 计算可得到路径 $a-b-z$，依次推导，可得路径 1、2、3…$2n-1$ 均在生成的路径集中且可以通过线性计算得到目标路径。

第二种情况，目标路径中存在重复片段，如图 5 所示，当程序中存在循环时，目标路径中可能会存在重复片段，以图 5（a）为例，假设目标路径为 $x-a-b-a-b-a-c-y$，由路径生成算法可知，生成的路径集合中包括 $x-a-c-y$ 和 $x-a-b-a-c-y$，经过简单线性运算可得到子路径 $b-a$，与路径 $x-a-b-a-c-y$ 相加即可得到目标路径，图 5（b）中的程序控制流图亦是如此，因此程序中存在循环时，重复片段也可通过生成路径的线性运算得到。即程序中的其他路径都可由算法生成的路径集合中的路径经过线性运算得到。

综合以上三点证明，生成的路径集合为基本路径集合。

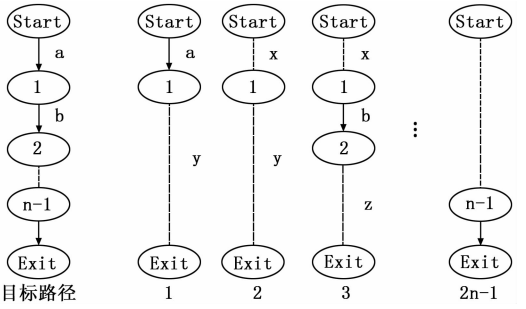


图 4 不存在重复片段的路径

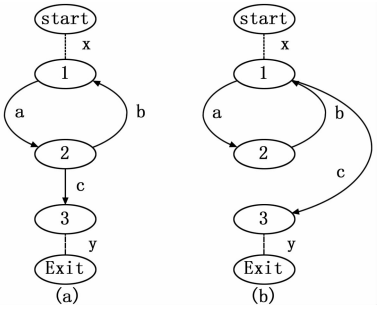


图 5 存在重复片段的路径

3 程序插桩

为了记录程序实际执行过程中的路径，需要在程序中插入记录执行路径信息的语句，即程序插桩。主要思路是在程序的分支节点处插入语句记录程序执行时的分支走向，执行结束后，每个节点处的分支走向即为此次执行的路径。为了便于与基本路径集合中的路径进行比对，在插桩时采用与程序控制流图节点一致的数据结构。

遍历 AST 时，在 Token 流中指定位置插入语句，最后输出 Token 流即为插桩后的被测程序。在每个可能的分支处插入节点，与控制流图中的节点保持一致。

插桩语句形式如下：

```
PathRec path = new PathRec();
path.addBranchNode(BranchNode node);
```

采用 PathRec 记录节点序列，在程序执行过程中，通过 addBranchNode 方法收集执行路径上的节点，即可得到执行路径。

4 基本路径覆盖测试

基本路径覆盖测试的目标是使程序的每一条基本路径都被执行。通过基本路径集合的获取和程序插桩，可以判断在测试过程中是否覆盖了基本路径集合。测试流程如图 6 所示。以测试数据[9-10]为输入，执行插桩后的被测程序，并获取执行路径，将执行路径并与基本路径集合进行比对，若相同则从基本路径集合中删除该路径，直至路径集合为空，则实现了对被测程序的路径覆盖。针对特定的测试数据集合，也可以判断其对基本路径的覆盖程度。

5 实验与分析

以被测程序 Sample 为例进行实验，对基本路径测试方法进行验证，Sample 程序如下：

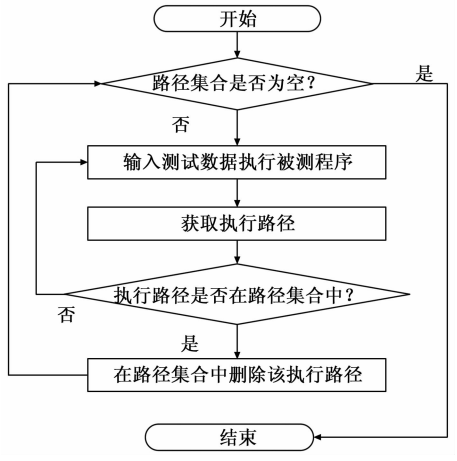


图 6　测试流程图

```
public class Sample{
  public int test(int array[], int length){
  int sum = 0, average = 0;
  for(int i=0;i<length;i++){
  if(array[i] >= 0)
  sum += array[i];
  else
  sum -= array[i];
  }
  average = sum/length;
  return average;
  }
}
```

首先，构造被测程序的控制流图，图 7 为生成的控制流图。

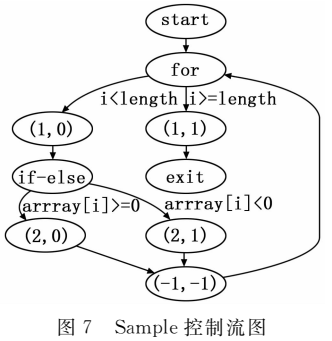


图 7　Sample 控制流图

然后，对程序进行插桩，插桩后被测程序为：

```
import path.PathRec;
public class Sample {
  public static PathRec path = new PathRec();
  public int test(int array[], int length) {
  int sum = 0, average = 0;
  for (int i = 0; i < length; i++) {
  path.addBranchNode(new BranchNode(1, 0));
  if (array[i] >= 0) {
  path.addBranchNode(new BranchNode(2, 0));
  sum += array[i];
  } else {
  path.addBranchNode(new BranchNode(2, 1) );
  sum -= array[i];
  }
  }
  path.addBranchNode(new BranchNode(1, 1));
  average = sum / length;
  return average;
  }
}
```

根据生成的控制流图生成基本路径集合，基本路径集合为：

(1, 0) -> (2, 0) -> (1, 1),

(1, 0) -> (2, 1) -> (1, 1),

(1, 1)。

最后，采用人工或自动方式生成测试数据，执行插桩后的被测程序，获取测试数据对应的执行路径，验证测试数据集合对基本路径的满足程度。针对 Sample 程序，当测试数据为 { [2, 1], 1}, { [−2, 1], 1}, { [2, 1], 0} 时，即可实现对基本路径的覆盖。

6　结论

本文通过对 Java 源码进行分析，构建程序的控制流图，同时对被测程序进行插桩；根据控制流图生成基本路径集合，在插桩后的被测程序执行过程中记录执行路径；通过执行路径和基本路径的自动化比对，判断测试数据对基本路径集合的覆盖程度。通过实验证明了该方法在 Java 程序基本路径测试中的有效性。

下一步将在路径生成的过程中对分支条件进行分析处理，采用符号执行、约束求解等方法来筛选出不可行路径，从而减少后期测试工作量，提高测试效率。

参考文献：

[1] 张小松，等译. 软件测试 [M]. 北京：机械工业出版社，2006.

[2] 施冬梅. 嵌入式软件路径覆盖测试的研究 [J]. 计算机测量与控制，2010，18 (10)：2236 - 2237.

[3] 王　敏，陈亚光. 用于基本路径测试的路径字符串组合算法 [J]. 计算机工程与科学，2013，35 (12)：134 - 140.

[4] 吴取劲，阳小华，鹿江春，等. 一种基于图深度优先搜索的基本路径集自动生成优算法 [J]. 南华大学学报：自然科学版，2012，26 (12)：87 - 90.

[5] 李　鹏，彭祥伟，周　喜，等. 基于状态图的测试路径自动生成 [J]. 计算机工程，2011，37 (1)：25 - 29.

[6] 张广梅，李晓维，韩丛英. 路径测试中基本路径集的自动生成 [J]. 计算机工程，2007，33 (22)：195 - 197.

[7] 韩　寒，姜淑娟. 路径测试中基本路径集自动生成方法的研究 [J]. 微电子学与计算机，2013，30 (1)：104 - 109.

[8] 王　冠，景小宁，王彦军. 基本路径测试中的 McCabe 算法改进与应用 [J]. 哈尔滨理工大学学报，2010，15 (1)：48 - 51.

[9] 冯俊池，于　磊. 测试数据生成中遗传算法的改进 [J]. 计算机辅助设计与图形学学报，2015，27 (10)：2008 - 2014.

[10] Anand S, Burke E K, Chen T Y, et al. An orchestrated survey of methodologies for automated software test case generation [J]. Journal of Systems and Software, 2013, 86 (8): 1978 - 2001.