第 36 卷第 1 期 通 信 学 报 Vol.36 No. 1
2015 年 1 月 Journal on Communications January 2015
doi:10.11959/j.issn.1000-436x.2015003

空指针异常的自动故障定位方法

姜淑娟[1]，王兴亚[1]，张艳梅[1]，李威[1]，鞠小林[1,2]，刘颖祺[1]
（1. 中国矿业大学 计算机科学与技术学院，江苏 徐州 221116；2. 南通大学 计算机科学与技术学院，江苏 南通 226019）

摘 要：提出一种空指针异常自动定位方法。该方法首先结合程序的静态分析技术，利用程序运行时的堆栈信息指导程序切片，然后对得到的切片进行空指针分析及别名分析,得出引发空指针异常的可疑语句集合，最终给出错误定位报告。实验结果表明，所提方法虽然因收集堆栈信息和别名分析增加了时间开销，但是利用堆栈信息可以缩小问题搜索空间，别名分析可以发现隐含的值传递过程，从而克服单独使用静态方法分析结果引起误报和漏报的缺陷，使最终的错误定位结果更精确。

关键词：自动故障定位；空指针异常；实时堆栈；程序切片；别名分析
中图分类号：TP311 **文献标识码**：A

Fault localization approach for null pointer exception

JIANG Shu-juan[1], WANG Xing-ya[1], ZHANG Yan-mei[1], LI Wei[1], JU Xiao-lin[1,2], LIU Ying-qi[1]
(1. School of Computer Science and Technology, China University of Mining and Technology, Xuzhou 221116, China;
2. School of Computer Science and Technology, Nantong University, Nantong 226019, China)

Abstract: A novel approach to locate the fault for null pointer exception is presented. The approach first computes the static slice guided by the runtime stack, then conducts a null-pointer analysis and aliases analysis on the sliced program and obtains the suspicious statements that throw null-pointer exceptions, and finally generates a fault localization report. The experimental study indicates that, although time spent in runtime stack information collecting and aliases transferring, the proposed approach can narrow the searching space of the problem with runtime stack, and discover the value transfer process with aliases analysis, thereby eliminating false negative and false positive, and improving the effectiveness of fault localization.

Key words: automatic fault localization; null pointer exception; runtime stack; program slicing; alias analysis

1 引言

多数高可信领域对软件系统的稳定性、健壮性和可靠性都有严格的要求。异常处理机制是 Java、C++等高级程序设计语言提供的一种用来保障软件可靠性的重要措施。程序异常一般可以分为 2 类：第一类是应用异常，由程序员在应用程序中显式定义异常的抛出条件及异常类型，并在异常条件满足时引发；第二类是运行时异常，指在程序运行时由于违反系统的约束条件而隐式地引发。例如，数组越界访问异常、空指针引用异常等就属于运行时异常。异常若不能被程序正确捕获并处理，则可能导致程序崩溃。第一类异常是由于程序员自定义的，因而比较容易处理。目前，针对应用异常的研究有很多，如对应用异常进行分析，在程序设计、测试和维护等方面为开发人员提供有价值的信息[1~3]。

收稿日期：2013-06-03；**修回日期**：2014-03-10
基金项目：国家自然科学基金资助项目(60970032, 61202006)；江苏省“333”基金资助项目；中央高校基本科研业务费专项基金资助项目(2013QNB17)；江苏省高校自然科学研究基金资助项目(12KJB520014)；江苏省研究生培养创新工程基金资助项目(CXZZ12-0935)；南京大学计算机软件新技术国家重点实验(KFKT2014B19)
Foundation Items: The National Natural Science Foundation of China (60970032, 61202006); 333 Project of Jiangsu Province; The Fundamental Research Funds for the Central Universities (2013QNB17); The University Natural Science Research Projects of Jiangsu Province(12KJB520014); The Graduate Training Innovative Projects Foundation of Jiangsu Province (CXZZ12-0935); State Key Laboratory for Movel Software Technology at Nanjing University (KFKT2014B19)

然而第二类异常由于其引发条件是在程序的运行过程中隐式满足，因此，在设计程序时很难发现程序的缺陷。由于运行时异常的引发不像应用异常的引发那样可以预测，为此开发人员很少为运行时异常设计处理程序。因而运行时异常一旦发生，程序很难通过自身的异常处理机制来处理，经常需要人工的干预来检查并定位引发异常的根源。运行时异常由于其引发条件的隐蔽性和不确定性，定位该类异常并排除程序相关缺陷存在一定的困难。针对运行时异常的研究是当前故障定位研究热点之一。以 Java 程序为例，由于 Java 编译器并不对运行时异常进行检测，即在一个方法中异常引发时只有 2 种处理方式：一是在方法内指定与之匹配的处理程序；二是在方法的异常界面中指定该方法可能传播出该类异常。因此，当在程序的执行过程中引发运行异常时，如果没有匹配的异常处理程序来处理，将导致程序崩溃。

空指针异常是一类常见的运行时异常。例如 Java 中调用值为 null 对象的任何方法，都会引发空指针异常，而 Java 语言中的函数调用、别名引用等机制使在程序中查找和定位可能为空的指针对象非常困难。

仅使用静态方法来检查程序潜在故障的方法有很多，典型的如文献[4~7]所述。这些方法大多要求用户提供程序的注解且无法解决静态分析结果不精确的问题。采用静态分析与动态信息相结合的方法是解决静态分析结果不精确问题的一个很有效的方法[8~10]，但已有这些方法用于收集动态信息需要花费很大的代价。如果首先通过空指针分析确定程序中所有的值可能为空的对象，便可以缩小查找空指针异常的错误值来源的范围。Sinha 等[11]提出一种利用堆栈信息从当前异常引发位置后向遍历程序，查找赋空值语句，进而检查这些空值语句以确定空指针异常的引发原因的方法。这种方法需要遍历整个程序，当程序规模很大时，需耗费大量的时间。而且，他们的方法没有考虑对象别名对分析结果的影响。别名的存在影响了错误定位结果的准确度。别名分析可以帮助获取更为准确的引用型变量指向信息、提高静态分析精度。

程序切片通过分析特定语句的依赖关系，抽取控制依赖与数据依赖的语句。通过计算空指针异常引发点的切片，可以排除与空指针异常引发无关的程序语句，因而程序切片具有简化问题、缩小分析范围的优势[12,13]。

基于上述分析，在对程序切片、程序的数据流和依赖性分析进行了较为深入研究的基础上[14~16]，提出一种定位当前运行时空指针异常引发根源的方法。该方法将静态分析技术与实时堆栈信息相结合，针对异常引发点的程序切片开展空指针分析和别名分析以定位空指针异常。具体而言，方法分为 2 个阶段：1) 在实时堆栈信息的指导下对程序进行约减，在此基础上进行程序切片；2) 对切片后的程序进行空指针分析和别名分析。本方法所使用的动态信息是已有的实时堆栈信息，不需要额外的工作量。将该方法应用到开源项目的空指针异常的定位中，实验结果表明该方法可提高空指针异常的定位效率和精度。

2　问题描述

空指针异常的定位与检测问题可以从理论上描述为对问题 $D=\{P, M, A\}$的求解。这里 P 是待检测的程序，M 是空指针异常的模式集合，A 是定位空指针异常的算法。P 可以描述为程序语句的集合；空指针异常模式用状态机描述为 $M=<N, T, C>$。状态集合 $N=\{N_{start}, N_{exception}, N_{end}, N_{not}, N_{maybe}\}$，表示空指针异常模式可能到达的状态，状态变迁 $T=\{<n_i, n_j>|n_i, n_j\in N\}$，状态变迁条件为 $C:N\times C\rightarrow N$。对象 X 引发空指针异常是指对象的状态从初始状态 N_{start} 经过一系列的状态变迁，最终到达状态 $N_{exception}$。此外，空指针异常检测算法描述为 $A=\{\rho(L, X), \Psi\}$，其中，ρ计算程序 P 在执行到 L 处时变量 X 的程序切片，Ψ为当前实时堆栈中方法调用的摘要信息。

如果已知程序的空指针异常引发点并获得异常引发时的实时堆栈信息，那么，依据检测算法 A 可以检测引发空指针异常的原因。但传统的基于静态分析的程序切片方法因为考虑程序所有可能的执行情况，从而容易产生误报，即

$$\exists x, x\in \rho(L, X)\wedge \sim D(P, M, A)\Rightarrow FP(X)$$

其中，$\sim D(P, M, A)$表示变量 X 的取值 x 不会引发异常模式 M。$FP(X)$表示对变量 X 的误报。

此外，由于别名机制存在，导致以上分析结果有漏报的可能，即

$$\exists x, x\notin \rho(L, X)\wedge D(P, M, A)\Rightarrow FN(X)$$

其中，$D(P, M, A)$表示变量 X 取值 x 会引发异常模式 M。$FN(X)$表示对变量 X 的漏报。

以图1中的一段Java程序为例说明上述问题。图1中的程序包含4个简单方法：method1, method2, method3, method4，一个构造方法foo及一个main方法。运行时输入“1”将会在语句9引发一个空指针异常，程序将会终止，此时实时堆栈信息如下所示。

exception in thread "main" java.lang.nullpointer exception

at cn.edu.cumt.slicer4j.test.foo.method1 (foo.java:9)

at cn.edu.cumt.slicer4j.test.foo.method4 (foo.java:18)

at cn.edu.cumt.slicer4j.test.foo.main (foo.java:25)

实时堆栈信息表明程序执行触发一个空指针异常。当前堆栈中依次存在3个方法：method 1、method 4和main。main方法在语句25调用了method 4，method 4在语句19调用了method 1。由于method 1在执行语句9时，对象obj1是一个null值，从而引发了一个空指针异常。

由此可见，当引发运行异常时，Java实时环境在实时堆栈中保存了当前程序执行时方法之间的调用关系及执行顺序。这些信息包括引发运行时异常的语句所在的行号、该语句所在的方法名、以及程序从main方法开始运行到该语句所调用并且没有正常退出的方法。如果一个方法被调用过但已经正常退出，则不会显示在实时堆栈中。实时堆栈中的信息可以辅助开发人员定位并修复引发该异常的故障。而无需考虑那些在当前运行中根本不可能被调用的方法，进而大大减少空指针分析的工作量，最终提高故障定位的效率和准确率。

在运行时异常引发时，若程序没有对应的异常处理程序，则程序会立即终止，并且与该异常相关的动态信息会保存在实时堆栈中，利用程序调试接口，将实时堆栈信息输出到外部文件，用作后续的切片程序输入。

然而由于实时堆栈信息粒度较粗糙，因此需要结合程序的静态分析得到的信息开展空指针异常定位。堆栈信息中仅仅包括本执行中所涉及到的方法以及方法调用语句，并不包括方法内部的控制流信息。另外，对于当前执行中被调用并且已经正常返回的方法并不出现在实时堆栈中。因此，只利用实时堆栈信息查找空指针异常引发的原因，会漏掉那些在本次执行过程中运行过但在实时堆栈中没有显示的方法，导致可能不能全面地理解程序执行时复杂的控制流，也无法进行全面的别名分析。

例如，图1实例中的实时堆栈信息表明main方法调用了method 4方法，而method 4方法调用了method 1方法，最终在method 1方法中(程序第9行)引发了空指针异常。在堆栈信息提示的3个方法中，只有method 1方法对obj1进行了赋值操作，而无法判断赋值操作一定会引发空指针异常。因此，仅利用实时堆栈信息查找空指针异常的引发原因是不够的，需要通过分析程序的执行过程，获取更多的运行时信息(如控制流、数据流等)来辅助空指针分析和别名分析，以降低故障定位的误报和漏报。

```
public class foo {
1)     private object obj1;
2)     private object obj2;
3)     private object obj3;
4)     public void method1(string mode) {
5)             if ("1" equals(mode)) {
6)                     obj1 = obj2;
7)             } else if (obj3 == null) {
8)                     obj1 = new object();
                }
9)         system.out.println( “obj1:” + obj1 to string());
10)    }
11)    public void method2() {
12)            this.obj2 = new object();
13)    }
14)    public void method 3() {
15)            this obj3 = new object();
16)    }
17)    public void method 4 (string mode) {
18)            this method 1(mode);
19)    }
20)    public static void main(string[] args) {
21)    foo foo = new foo();
22)    if (args length >= 1) {
23)                    if (args[0] length()==1) {
24)                            foo method 3();
                        }
25)                    foo method 4(args[0]);
                } else {
26)                    foo method 2();
                }
27)    }
28)  public foo() {
29)          obj1= null;
30)          obj2= null;
31)          obj3= null;
32)  }
33) }
```

图1　一个实例程序foo

图2显示了实例程序foo的过程间的控制流。对于输入“1”的执行路径如下：21a，28，29，30，

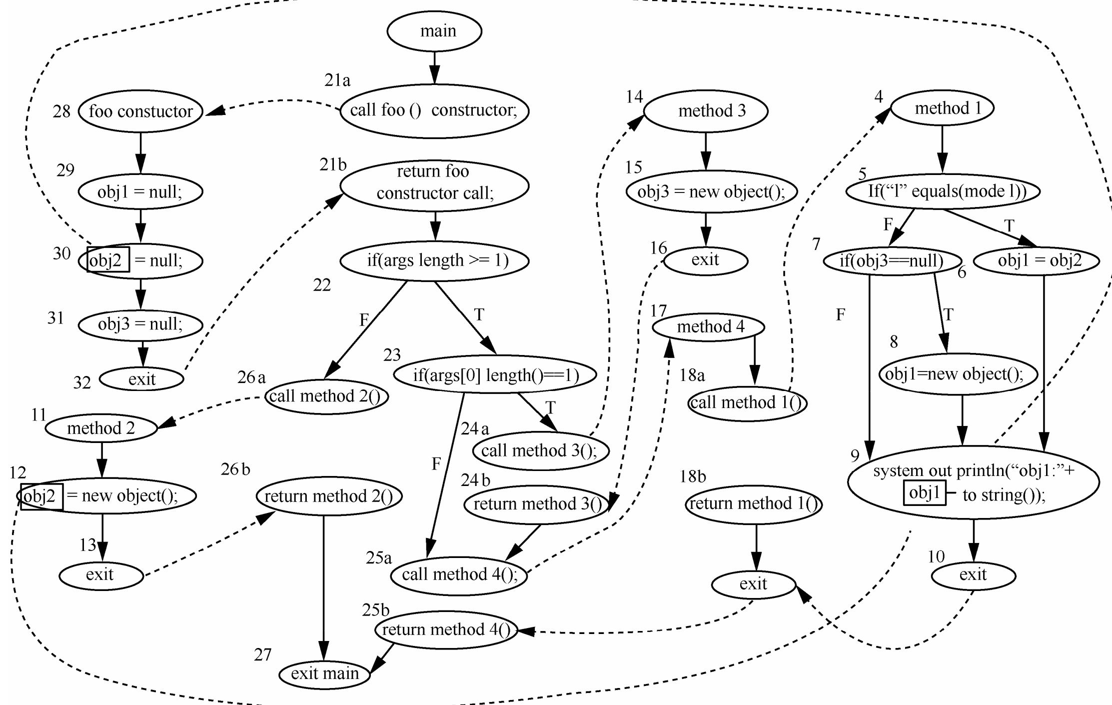


图 2　实例程序 foo 的控制流

31，32，21b，22，23，24a，14，15，16，24b，25a，17，18a，4，5，6，9。通过回溯分析当前程序的执行，可知引发语句 9 空指针异常的赋值空值语句在程序语句 30 中(obj2=null)，在语句 6 中将 obj2 的空值复制给 obj1，最后，该空值 obj1 在语句 9 中被引用，从而触发了空指针异常。然而当空指针异常引发时，程序构造方法(foo)中的语句 30 不在实时堆栈中。

此外，根据实时堆栈信息和程序结构可以断言：method 2 根本不可能执行。如果把那些根本不可能执行的方法和语句删除，则会大大降低程序分析的复杂度，减少进行故障定位的工作量。由于程序切片技术具有简化问题、缩小范围的优点[12,13]，因此，可以先在实时堆栈信息指导下，结合程序的静态结构，首先，将当前测试中肯定没有执行的方法排除，对剩下的程序利用实时堆栈进行切片，然后，再对切片后的程序进行静态分析，进而找到引发该空指针异常的根源。

虽然切片后的程序较切片前有所减少，但是仍不足以精确定位到引发空指针异常的错误值来源的语句。例如，由图 2 可知，如果在执行语句 9 之前先执行了语句 8，则语句 9 就不会引发空指针异常。因此，在进行空指针异常故障定位时应该从切片后的程序中删除那些不产生空引用的语句。

此外，别名分析也有助于进一步提升故障定位的精度，降低漏报的可能。例如分析图 1 中的实例程序，可以发现 foo 的构造函数中语句 30 中的 obj2 和引发空指针异常的语句 9 中的 obj1 互为别名，方法 method 2 中语句 12 中的 obj2 和引发空指针异常的语句 9 中的 obj1 也互为别名。另外，在程序中还有一些引用变量，它们也可能是空值，但与引发空指针异常的变量并不互为别名，它们与当前空指针异常中的空值也没有直接关系，则应该从切片集合中暂时去掉这些变量，如语句 31 中的 obj3。

综上分析，先在实时堆栈信息指导下进行程序切片，然后再进行空指针分析和别名分析，可以提高空指针异常故障定位的精度和效率。

3　本文方法框架

利用实时堆栈信息针对待分析源代码进行约减，排除当前不可能执行代码(方法)，进而在约减后的代码上进行切片。从前面的分析可知，在程序切片的基础上进行空指针分析和别名分析能够有效降低空指针异常定位的误报率和漏报率。本文方法定位的是在当前执行下产生空指针异常故障来源语句。具体而言，包含如下过程：1) 基于实时堆栈信息对程序中的方法进行分类，排除

在当前肯定未被执行的方法，即在发生空指针异常的这次运行中，这些方法没有被执行；2) 在实时堆栈信息的指导下进行程序切片；3) 对切片后的程序进行空指针分析和别名分析；4) 对空指针分析结果和别名分析结果进行合并(集合交运算)，进而得出故障定位结果报告。方法框架如图3 所示。

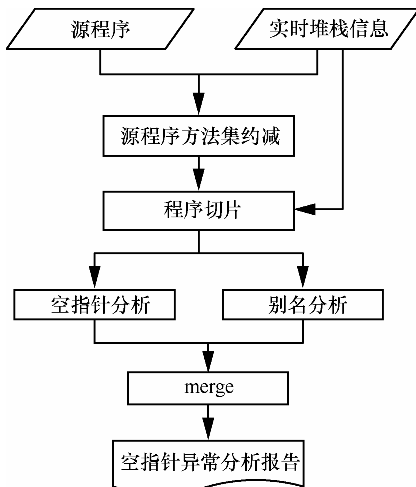


图3 方法的框架

步骤 1 依据实时堆栈中的信息，通过静态分析程序的控制流程图，对程序中的方法按其在当前执行中是否被执行进行分类。排除在当前运行中肯定不执行的方法。

步骤 2 在步骤 1 结果的基础上，以空指针异常的引发点为切片准则，结合实时堆栈信息进行后向静态切片，进一步缩小定位空指针引用异常的可疑范围。

步骤 3 在步骤 2 得到切片基础上，分别进行空指针分析和别名分析。其中空指针分析目的是分析程序中值可能为空的引用型变量，即显式存在的空指针赋值；别名分析目的是为了找出切片中所有引用型变量的指向信息，即空指针引发点依次关联的所有别名。空指针分析和别名分析后分别得到异常引发根源的语句集合。

步骤 4 把空指针分析的结果与别名分析的结果进行集合交运算，进一步排除可能误报的情形，从而得到更小的可疑语句集合，进而得到空指针异常的分析报告。

最终，基于上述框架，利用开源软件 soot 提供的程序静态分析接口，设计并实现一个用于空指针异常分析的故障定位工具。

4 定位空指针异常

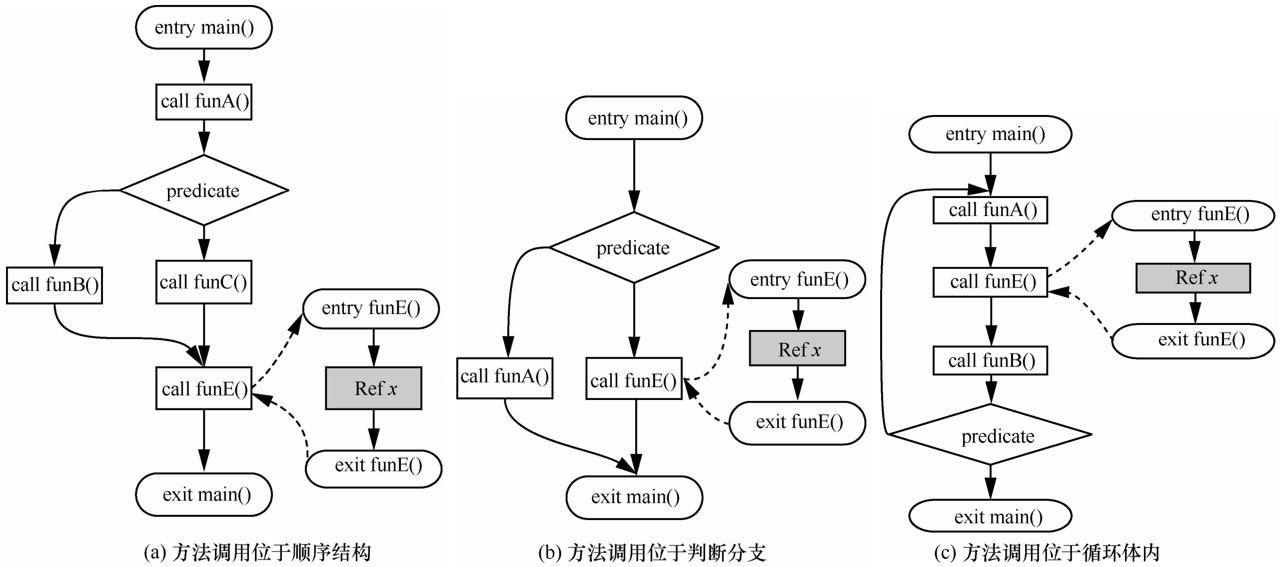
4.1 基于堆栈信息的程序约减

一般情况下，程序执行发生异常，一定是程序执行过的语句当中存在错误，而那些未被执行的语句则不会导致本次执行失败。因此，可以考虑排除程序中未被执行的语句。在空指针异常故障定位系统中，排除那些在当前运行中肯定不被执行的方法。具体方法如下。

对程序中的方法按其在当前运行时是否被执行过进行分类。可以分为：1) 肯定被执行的方法；2) 可能被执行的方法；3) 肯定不被执行的方法。

考虑现代程序语言的基本结构只有 3 种，顺序结构，选择结构和循环结构。由 3 种基本结构构成的程序中，方法调用情况对应可以分为 3 种，如图4 所示。图 4 中方法 FucE()中语句 Ref *x*(阴影部分)，表示对象 *x* 的引用，若执行到此处(阴影部分)时 *x* 为空，则必然引发空指针异常。分析过程为：1) 对于图 4(a)，若语句 Ref *x* 引发空指针异常，则在实时堆栈中保存的方法有 funE 和 main。由于 funA 位于程序主干上，与发生异常的 funE 是顺序结构，是 funE 的后向必经节点，可知 funA 必定被执行过，因此 funA 归为肯定执行的方法类别。另外，funB 和 funC 分别位于程序的 2 个分支上，在没有实时跟踪程序执行的情况下，无法断定它们是否一定被执行，因此，可以把 funB 和 funC 归为可能执行的方法类别；2) 对于图 4(b)，若语句 Ref *x* 引发空指针异常，则在实时堆栈中保存的方法有 funE 和 main。由于 funA 与 funE 分别处于 2 个程序控制流分支，因此 funA 是肯定不执行的方法；3) 对于图 4(c)，若语句 Ref *x* 引发空指针异常，在实时堆栈中保存的方法有 funE 和 main。由于 funE 肯定被执行了，而 funA 处于 funE 后向必经节点，因此 funA 是肯定被执行的方法。但是如果在 funE 第一次执行时就引发了空指针异常，则 funB 不会被执行；如果 funE 在循环发生后引发空指针异常，则 funB 一定被执行。因此，图 4(c)中的 funA 归类为肯定被执行的方法，funB 归类为可能被执行的方法。

例如，图 2 中的实例程序的控制流图，当程序在语句 9 异常终止时，main 函数在节点 25a 处终止。由控制流图可知，节点 21、22 和 23 都是从 main 的入口到达节点 25a 的必经节点，并且节点 21 和 23 又各自分别调用了类 foo 的构造方法和 method 3



(a) 方法调用位于顺序结构　(b) 方法调用位于判断分支　(c) 方法调用位于循环体内

图 4　3 种基本结构对应的方法调用组合

方法，因此，可以确定它们都被调用过并且已正常返回。对于节点 24，由于实时堆栈信息并没有包含方法内控制流信息，因此，无法判断从节点 23 到达节点 25 是否经过了节点 24，则方法 method 3 是可能执行的方法。

再如，图 2 的 main 方法中语句 26 调用了方法 method 2，而 main 方法中语句 26 在当前执行过程中是不可能执行的语句，因此，可以断定方法 method 2 在当前执行过程中没有被调用过，则其对 obj2 的修改在当前执行中也没有任何意义。

综上所述，当空指针异常引发时，可以根据实时堆栈信息，结合静态分析程序的结构，把程序中的所有方法按图 4 中的 3 种模式进行分类。在设计的系统中，通过前向遍历程序控制流程图完成对方法的分类。

4.2　基于实时堆栈的静态切片算法

由上一节的分析可知，程序中存在一些没有执行过的方法和语句，并且这些方法和语句对本次执行不产生任何影响，则切片时可以不对其进行跟踪，由此也就不需要分析其依赖关系。

考虑图 1 中的实例程序，肯定被执行的方法有 main、method 4 和 method 1，肯定没有执行的方法有 method 2，可能被执行的方法有 method 3。因此，在构建系统依赖图时可以忽略掉方法 method 2，约减之后的系统依赖图较约减之前有明显简化。

本文改进了静态切片算法，其切片准则为<P, V, CS>，其中 CS 为一个实时堆栈信息，可以是空指针异常引发时的调用序列，或人工添加的一条调用序列(前提是该调用序列必须是可行的)。改进的静态程序切片算法的主要思想是：在约减的系统依赖图的基础上进行反向遍历，查找切片准则中的数据依赖和控制依赖语句，同时不去分析当前执行中一定不会执行的那些方法。具体如算法 1 所示。

算法 1　SliceUnderStaticTrace

Input　*G*: simplified system dependency graph

Output　*Slice*: slice Based on real-time stack

begin

1) markreachingvertice(*G*, *S*, { parameter_out });

2) *S′* = all marked vertices in *G*;

3) markreachingvertice(*G*, *S′*, {paramter_in, call});

end

function markreachingvertice(*G*, *V*, *Kinds*)

Input　*G*: simplified system dependency graph
V: collection of the initial node
Kinds: edges that are not tracked when reversed traversal

Output　*S*: all nodes that can be reached from the initial node

Declare　*Mark*(*v*): marking for *v*
Pred(*G*, *v*): all of the direct precursor of *v* in *G*

Method(v): method in which node v is referenced

E(m): exceptional terminal node of method m

Fetch(*worklist*, v): fetch a node v form worklist

add(*worklist*, w): add a node v to worklist

begin

1) *worklist* = v

2) **while** *worklist* $\neq \varnothing$ **do**

3) *Fetch*(*worklist*, v)

4) *Mark*(v)

5) **for** each node w that is not marked in *Pred*(G, v) **do**

6) **if** edge w->v belongs to the type of *Kinds* then continue **end if**

7) **if** *Method*(v) is marked as not executed definitely then continue end if

8) **end if**

9) *add*(*worklist*, w)

10) **end for**

11) **end while**

end

算法 1 是一个典型的 2 步遍历的切片算法[13]。从切片准则语句开始，第一步为不沿参数输出边对系统依赖图(SDG)进行反向遍历，第二步为不沿参数输入边和调用边对 SDG 进行反向遍历。遍历的节点构成的集合即为切片。算法 1 中的子函数 markreachingvertice 的第 7)行和第 8)行针对待分析程序的方法执行情况进行处理，首先对跟踪的每条边 $w\rightarrow v$ 进行判断：如果起点 w 所在的方法 M 被标记为“肯定不被执行”的时候，切片算法就不需要跟随这条边进入一个不可能执行的方法中。

4.3　空指针分析

空指针分析是分析并找出代码中可能为空的引用型变量，在空指针异常故障定位中发挥着十分关键的作用。在 Java 中，值为 null 的对象调用任何方法都会引发空指针异常，空指针引用在程序中较为隐蔽，因而空指针异常是 Java 中最难查找和调试的一种异常。如果能找出程序中所有的值可能为空的对象，则会大大缩小查找空指针异常的错误值来源的范围。利用开源软件 soot[17]中提供的编程接口实现的空指针分析功能。具体而言，在程序切片后的程序上，找出代码中可能为空值的引用型变量并将其标出，以便于找到引发空指针异常的错误值来源。实际上，对于一个包含引用型变量 v 的语句，可利用 soot 提供的编程接口，实现对变量 v 取值的判断，即 v 是否可能为空。

算法 2 给出了基于 soot 实现针对对象的空指针分析过程。算法的基本思想是：首先，根据前面的切片结果获取待分析的代码；然后，分析这些代码中的引用型变量的取值情况；最后，根据分析结果生成 xml 文件，结果中标签<alwaysNullList>内的行号表示对应的代码行中存在空变量；标签<NullUnknown>内的行号表示对应的代码行中存在可能为空的变量。具体分析过程如算法 2 所示。

算法 2　doNullnessAnalysis

Input　resultList：the program slice

Output　alwaysNullList: code lines which contain the variable with null value
nullUnknownList: code lines which contains the variable that maybe null object

Declare　is UnknownNull (statement, local)：identify whether the variable *local* can be null
isAlwaysNull (statement, local)：identify whether the variable *local* is always null

begin

1) **for** each class c in resultList **do**

2) stmtList ← all statements of class c;

3) **for** each statement in stmtList **do**

4) refLocalList ← refVariables in statement;

5) **for** each local in RefLocalList **do**

6) **if** isAlwaysNull (statement,local) then

7) mark statement as null; **end if**

8) alwaysNullList ← statement;

9) **if** isUnknownNull (statement,local) then

10) mark statement as unknown; **end if**

11) nullUnknownList ← statement;

12) **end for**

13) **end for**

14) **end for**

end

算法 2 的关键过程为第 5 行至第 12 行。先根据切片结果获取待分析的所有引用型变量，然后结合数据流方程和它们所在的代码行分析这些引用变量是否为空。算法中方法 is AlwaysNull 和 is UnknownNull 是采用 soot 的空指针分析原理实现的。最后的分析结果以 xml 格式保存到外部存储器。

4.4 别名分析

别名分析指的是找出引用型变量的指向信息，根据指向信息判断 2 个引用型变量是否互为别名。为了进一步提高空指针异常故障定位的精度并减少误报，本文方法在完成切片的基础上，利用开源软件 soot 提供的编程接口，针对引发空指针异常的对象进行了别名分析，充分考虑了别名对引用型变量指向信息的影响，便于能进一步锁定引发空指针异常的错误语句，从而提高空指针异常故障定位的精度并降低误报率。具体而言，别名分析算法利用 soot 编程接口，4.2 节中切片分析结果，依次抽取需要检测代码行，然后，检测这些代码中的引用型变量和引发空指针异常的变量是否存在别名关系。提取与指定代码行变量存在别名关系的变量所在行，并按照空指针分析结果的 xml 文件格式保存别名分析的结果，具体过程如算法 3 所示。

算法 3　doAliasAnalysis

Input sliceResultList: the result of SliceUnderStaticTrace

className: the class in which the runtime exception occured

lineNo: the line number of code which thrown the runtime exception

Output a xml file which contains analysis result

Declare ePointsToSetList: the alias info list of the object that resulted in runtime exception

pointsToSetMap:the alias information map of all the reftype variables

pointsToSet: a class which contains alias information

pta: the instance of class points to analysis

begin

1) fetch the list of reftype variables according to className and lineNo,noted as localList ；

2) **for** each local in localList **do**

3) pointsToSet pointsToSet = pta reaching-objects(local);

4) ePointsToSetList ← pointsToSet ;

5) **end for**

6) fetch all locals for analyzing according to sliceResultList as gLocalList；

7) **for** each glocal in gLocalList **do**

8) pointsToSet pointsToSet = pta reaching-objects (glocal);

9) pointsToSetMap ← pointsToSet;

10) **end for**

11) **while** pointsToSetMap *is not empty* **do**

12) fetch pointsToSet from resultMap;

13) flag=hasNonEmptyIntersection (pointsToSet, ePointsToSetList) ;

14) **if** flag == true ;

15) resultList ←value paris <className, lineno>; **end if**

16) **end while**

17) output resultList into the xml file ;

end

算法 3 展现了别名分析算法(doAliasAnalysis)的详细流程。首先，根据类名和行号，获取可能引发空指针异常的所有变量，分析这些变量的指针信息(1)~5)行)；其次，根据切片分析过程的结果获取有待检测的引用型变量，依次分析这些变量的指针信息(6)~10)行)；第三，根据各个变量的指针信息，判断变量间的别名关系(11)~16)行)；最后，将生成别名分析结果以 xml 文件(17)行)格式输出到外部存储器。

5 实证研究

5.1 实验设计

为了验证提出的空指针异常故障定位方法的有效性，依据所提出的方法设计并实现了一个原型工具。并在一组开源程序上开展空指针异常故障定位实验。实验对象如表 1 所示。实验选取开源软件 ant 的 7 个不同版本，源程序代码行数为 77 980 行至 179 889 行。以空指针异常作为研究目标，从相应的缺陷数据库中获取了各自的空指针异常，包括 BugID、空指针异常引发的文件及所在异常语句行号。实验目的是验证所提方法能否有效定位这些异常，是否存在误报和漏报的情形。

表 1 实验对象介绍

程序名	代码行数	BugID	发生空指针异常的文件名(.java)	异常语句行号
ant 1.4	77 980	5 652	UnknownElement	148
ant 1.5.1	131 419	15 465	CBZip2InputStream	327
ant 1.6.0	172 445	25 826	GenericDeploymentTool	788
ant 1.6.2	152 483	32 200	CBZip2InputStream	285
ant 1.6.3	157 646	34 878	DirectoryScanner	875
ant 1.6.5	157 678	38 622	ImportTask	96
ant 1.7.1	179 889	46 236	DefaultLogger	339

5.2 实验结果及分析

为了验证方法的有效性,依据第4节所述方法,首先,对程序进行约简,排除不可能执行的方法,接着,在实时堆栈信息基础上,计算切片准则,并采取一种后向切片方法计算程序切片;然后基于程序切片分别进行空指针分析和别名分析,并将两者的计算得到的可疑语句集合进行交运算。最终定位空指针异常引发的根源。

具体而言,基于表1中的程序分别设计了2组实验:第一组实验中不利用实时堆栈信息,直接针对所有源代码进行切片,然后再进行空指针分析和别名分析;第二组实验在实时堆栈信息指导下进行故障定位,即先利用实施堆栈信息对源程序进行约简(通过排除不可能执行的方法缩减怀疑的范围),进而在约简后的程序上切片,然后再进行空指针分析和别名分析。以上2组实验的最终结果分别如表2和表3所示。

表 2 无实时堆栈信息指导的定位结果

BugID	切片后所剩行数	空指针分析后所剩行数	别名分析后所剩行数	最终定位行数
5 652	1 394	873	53	44
15 465	1 968	1 018	14	14
25 826	16 602	11 179	122	117
32 200	17 574	11 852	23	22
34 878	18 091	12 224	132	124
38 622	18 111	12 244	5	5
46 236	18 425	12 444	23	6

由表2可见,基于切片后的空指针分析和别名分析可以有效地缩小可疑语句范围。例如,定位编号5 652的空指针异常,通过切片计算把可疑范围从77 980行缩减到1 394行,进而对切片进行空指针分析和别名分析,分别将可疑范围缩小到873行和53行。最后对空指针分析和别名分析的结果进行集合交运算,得到可疑语句数为44行。

表 3 基于实时堆栈信息指导下定位结果

BugID	切片后所剩行数	空指针分析后所剩行数	别名分析后所剩行数	最终定位行数
5 652	1 390	868	53	44
15 465	263	44	10	10
2 5826	15 525	10 284	120	115
32 200	16 372	10 862	23	22
34 878	17 051	11 342	128	120
38 622	16 885	11 218	5	5
46 236	8 541	4 954	7	6

由表3可见,在实时堆栈信息指导下,先对程序进行缩减,然后再进行程序切片同样可以大幅度缩小故障定位的范围。例如,编号为32 200的空指针异常,先在实时堆栈信息指导下,用4.1节阐述的方法,对152 483行源代码进行约简,并进行切片,得到16 372行,这比表2中不进行源代码缩减情况下的切片(17 574行)要少很多行,这意味着,在缩减后的程序上进行程序切片可以减少切片计算的工作量,并得到更小的可疑语句集合。

针对表2和表3中的数据,采用Wilcoxon配对检验,结果表明,在实时堆栈信息指导下对程序约简后的切片规模比直接在源程序上所得切片规模有比较显著的缩小($p\text{-}value = 0.064\,1$),且空指针分析的规模也有较显著降低($p\text{-}value = 0.064\,1$),但是在表3中的别名分析结果较表2略有减少($p\text{-}value = 0.034\,98$),而最终故障定位结果则差别不大($p\text{-}value = 0.448\,9$)。

综合表2和表3的分析,虽然最终定位程序空指针异常的结果2种方法比较接近,但是,由于提出的程序缩减方法可以显著减少(置信度水平为93.6%)计算切片分析程序规模,因此,可以减少切片计算时分析程序、计算程序依赖关系的时间。总地来看,本文方法增加了程序缩减所需时间,减少了计算程序切片所需的时间。

由4.1节分析可知,程序中所有的方法均可通过程序一次失败执行时产生的堆栈信息分为肯定被执行的方法、可能被执行的方法和肯定不被执行的方法3类。在随后的切片过程中,只对程序中上述前2类方法进行依赖关系分析,并在此基础上进行程序切片,以用于后续的错误定位。因此,可以

获得当前执行过程中所有与状态异常语句有依赖关系的语句，并以此进行错误定位。然而，如果程序执行失败是由程序未执行某些语句引发的，由于本文方法缺乏针对程序中未执行方法的分析，构造的依赖关系并不完备，因而存在漏报的可能。除此之外，通过手工分析可疑语句集合中的语句，对照堆栈信息，发现引起当前空指针异常的语句均在可疑语句集合之中，并且从错误赋值语句开始到异常引发点的路径都包含在程序切片之中。因此，本文方法所得结果是正确的。

6　相关工作

目前针对程序故障定位的研究，很多学者提出了不同的解决方法。常用的方法主要有纯静态分析方法和动静结合分析的方法。

使用静态的故障定位方法检查潜在故障的方法有：Hovemeyer 等[4]使用前向过程间数据流分析和注解发现与空指针引用相关的 BUG。Evans[5]生成了 LCLint 来结合注解检查在 C 程序中的错误，如内存泄漏和别名。ESC/Java[6]是一个编译时间的程序检查器，它检查在注解语言中的设计描述与实际代码之间的不一致和潜在的实时错误。与本文技术不同的是，上面这些技术要求用户提供程序的注解，它们所提供的故障定位的质量依赖于注解。

Bush[7]通过检查 C 程序中一个内存错误的外部类来进行准确的过程间分析，包括空引用。这种技术的特点是采用过程自底向上的分析方法计算摘要，在每一个过程内部进行前向路径敏感分析来剪除那些不可达路径。然而，这种方法是一个纯静态的，也同样有静态方法所存在的公共问题。Nanda 和 Sinha[18]提出了一种上下文敏感和路径敏感的过程间分析方法，用于识别潜在的空指针引用。Xie 和 Engler[19]提出了一种用于查找系统错误（如空指针引用未释放锁）的方法。然而，这些方法是纯静态的，也同样有静态方法所存在的公共问题。与这些方法不同的是，将动态生成的信息（实时堆栈信息）与静态分析相结合，可以避免纯静态方法的问题。此外，本文技术在故障定位之前先进行实时堆栈信息指导下的程序切片，减少了搜索空间。

Rountev 等[8]讨论了静态分析不精确，并提出用于评估这种不准确的方法。解决静态分析不准确问题的一个公共方法是结合静态分析和动态分析。

Hangal 和 Lam 开发了一个工具 DIDUCE[9]，尝试通过动态程序常量检测去推测 BUG，收集动态常量是一个非常昂贵的分析，不适用于大型程序。相反，本文方法使用现成的动态信息(实时堆栈信息)查找故障。

Tom 等[10]提出了一种结合静态分析和动态分析的方法来发现 Java 程序中实时错误。该方法首先进行前向过程间符号执行来发现可能揭露 BUG 的约束，然后试图生成测试用例来触发该 BUG。然而，本文方法是后向分析，从发现 BUG 的一条语句（异常引发点）开始，使用有效的实时堆栈信息来指导后向的分析。

Baah 等[20]提出了一种通过建立概率程序依赖图来进行错误定位的方法。该方法首先通过静态分析方法生成程序依赖图，然后通过测试用例的执行信息中的数据来评估节点间的统计依赖关系，从而得到概率程序依赖图，最终将其运用于错误定位中。与本文方法的不同之处在于：1)该方法的静态分析方面主要体现在程序依赖图的建立；2)在动态分析方面，上述方法利用的是测试用例的执行信息，需要收集程序的动态运行信息，而本文方法不需要收集这些信息，可以在实时堆栈中直接获得。

Zhang 等[21]提出了一种静态分析与后向动态切片相结合的错误定位方法。该方法通过计算可执行语句的信赖度值来确定语句产生正确值的概率，其中，信赖度值越大表示语句产生的正确值的概率越高，然后通过剪去信赖度值中较大的值为 1 的语句来缩减动态切片的大小。2 种方法的主要区别在于缩减搜索范围的方法不同，本文是通过实时堆栈信息来缩减搜索范围，两者有各自的适用之处。

Sinha 等[11]提出了一种方法来定位那些在 Java 程序中由于不正确赋值导致实时异常的故障。该方法基于实时堆栈信息，从当前异常引发点开始，进行后向静态数据流分析，依次查找对可疑对象赋空值的语句。而本文方法优势有 2 点。1) 利用堆栈信息，结合对程序结构的分析，首先排除当前执行中肯定不会执行的方法，对剩下的程序进行切片，并在切片基础上进行空指针分析和别名分析。对程序的删减不会引起漏报，因为异常引发“原因”语句肯定包含在本次执行语句当中。而且这种删减使程序切片的速度得以提升；2)在切片基础上采用别名分析，有助于提高故障定位的精度，避免误报和漏报。不足之处在于，Sinha 的方法还能找到可能在下次运行中引发当前异常的那些赋空值语句。而本

文方法只关注与引发空指针异常的本次执行的那些赋空值语句。然而，本文方法缩减和别名分析工作可以较显著地提高故障定位的效率。

程序切片是在软件调试中广泛应用的技术。Gupta等[22]通过在静态切片中引入动态信息，提出了一种混合切片方法。该方法利用断点、函数调用图等程序调试过程中的动态信息，提高了程序切片的质量和精度。该方法与本文方法相同之处是两者都利用了现成的动态信息，文献[22]中动态信息是程序调试过程中产生的，而本文是空指针异常引发时产生的。2 种方法的不同之处是在程序切片之后，又进行了空指针分析和别名分析来完成故障定位。

目前，许多学者提出了一些基于测试的错误定位方法，典型的包括下列几种方法。

Renieris 等[23]提出了一种基于相似程序光谱的最近邻执行轨迹(NNQ）方法。在NNQ方法中，对于一个失败的执行和许多成功的执行，根据距离准则从成功执行中选择一条程序光谱和失败执行最近的成功执行，通过比较两者光谱的不同从而分离软件错误。Jones 等[24,25]提出了一种 Tarantula 方法，该方法通过计算程序实体的怀疑度值，并将其进行排序从而对源代码进行审查，直至找到软件错误的所在。Santelices 等[26]提出了一种基于多重覆盖的错误定位的策略，该方法指出覆盖信息类型的选取会对错误定位方法的有效性产生很大的影响，综合使用多种覆盖类型的信息，能够提高错误定位方法的有效性，该方法仅组合计算了语句的怀疑度值，在本质上与 Tarantula 方法一致。徐宝文等[27]提出了一种基于组合测试的故障定位方法。该方法根据组合测试的结果，补充一些附加测试用例重新进行测试，然后对结果进行分析，从而可以迅速地锁定很小范围导致故障的原因。

以上方法皆是根据测试时程序执行特征的统计信息来进行错误定位的，需要收集程序的动态执行信息，而本文方法不需要收集这些执行信息，而是直接使用现成的实时堆栈信息，因此，与基于测试的错误定位方法不同的是，本文方法在收集动态信息方面不会花费较多的资源和精力，具有一定的优越性。

7 结束语

针对 Java 程序中空指针异常经常发生并且难以定位其引发的根源的问题，提出了一种动态信息与静态分析相结合的故障自动定位的新方法。该方法首先在实时堆栈信息指导下从异常引发点开始进行后向程序切片，减少了搜索空间；然后，对切片后的程序进行空指针分析和别名分析；最后，用可视化工具显示分析结果和相关源代码。该方法既能在一定程序上克服静态分析结果不精确的不足，又能弥补实时堆栈信息过于粗糙无法单独应用的不足，而且也不需要花费额外的代价来收集动态信息。

将该方法应用到开源项目中的空指针异常的定位，实验结果表明对空指针异常故障定位方面有较好的效果。需要指出的是本文方法是针对程序的一次执行过程中的实时堆栈信息分析，难以获取全面的依赖关系，所以存在漏报的可能。因此，下一步工作是进一步研究如何利用静态分析技术获取程序的其他依赖信息，并开展形式化分析和理论证明，以拓展本文方法的普适性。同时，将本文方法拓展到其他类型的运行时异常的故障定位中，比如数组越界异常等；提出有效的方法克服实时堆栈信息在故障定位方面的不足，提高故障定位的准确率；本文方法还需要进行大量的实验进行验证；进一步完善本文的原型工具，为后续的故障定位提供有价值的信息。

参考文献：

[1] SINHA S, HARROLD M J. Analysis and testing of programs with exception-handling constructs[J]. IEEE Transactions on Software Engineering, 2000, 26(9):849-871.

[2] JIANG S J, XU B W, SHI L. An approach to analysis exception propagation[A]. Proceedings of the 8th IASTED International Conference on Software Engineering and Applications[C]. Anaheim, USA, 2004.300-305.

[3] ROBILLARD M P, MURPHY G C. Static analysis to support the evolution of exception structure in object-oriented systems[J]. ACM Transactions on Software Engineering and Methodology, 2003, 12(2):191-221.

[4] HOVEMEYER D, SPACCO J, PUGH W. Evaluating and tuning a static analysis to find null pointer bugs[A]. Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering[C]. Lisbon, Portugal, 2005.13-19.

[5] EVANS D. Static detection of dynamic memory errors[A]. Proceedings of the ACM SIGPLAN Conference on Programming Languages, Design, and Implementation[C]. Pennsylvania, USA, 1996.44-53.

[6] FLANAGAN C, LEINO K R M, LILLIBRIDGE M, *et al*. Extended static checking for Java[A]. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation[C]. Berlin, Germany, 2002.234-245.

[7] BUSH W R, PINCUS J D, SIELAFF D J. A static analyzer for finding

dynamic programming errors[J]. Software: Practice and Experience, 2000, 30(7):775-802.

[8] ROUNTEV A, KAGAN S, GIBAS M. Evaluating the imprecision of static analysis[A]. Proceedings of the 5th ACM SIGPLAN- SIGSOFT Workshop on Program Analysis for Software Tools and Engineering[C]. Washington, DC, USA, 2004.14-16.

[9] HANGAL S, LAM M S. Tracking down software bugs using automatic anomaly detection[A]. Proceedings of the International Conference on Software Engineering[C]. Orlando, Florida, USA, 2002. 291-301.

[10] TOMB A, BRAT G P, VISSER W. Variably interprocedural program analysis for runtime error detection[A]. Proceedings of the International Symposium on Software Testing and Analysis[C]. London, England, United Kingdom, 2007.97-107.

[11] SINHA S, SHAH H, GORG C, *et al*. Fault localization and repair for Java runtime exceptions[A]. Proceedings of the International Symposium on Software Testing and Analysis[C]. Chicago, Illinois, USA, 2009.153-164.

[12] WEISER M. Program slicing[J]. IEEE Transactions on Software Engineering, 1984, 10(4):352-357.

[13] KOREL B, LASKI J. Dynamic program slicing[J]. Information Processing Letters, 1988, 29(3):155-163.

[14] 姜淑娟, 徐宝文, 史亮. 一种基于异常传播分析的数据流分析方法[J]. 软件学报, 2007,18(1):74-84.
JIANG S J, XU B W, SHI L. An approach of data-flow analysis based on exception propagation analysis[J]. Journal of Software, 2007, 18(1):74-84.

[15] 姜淑娟, 徐宝文, 史亮等. 一种基于异常传播分析的依赖性分析方法[J]. 软件学报, 2007, 18(4):832-841.
JIANG S J, XU B W, SHI L, *et al*. An approach to analyzing dependence based on exception propagation analysis[J]. Journal of Software, 2007, 18(4):832-841.

[16] JIANG S J, ZHANG H C, WANG Q T, *et al*. A debugging approach for Java runtime exception based on program slicing and stack traces[A]. Proceedings of the 10th International Conference on Quality Software[C]. Zhangjiajie, China, 2010.393-398.

[17] VALLÉE-RAI R, LAM P, VERBRUGGE C, *et al*. Soot(postersession): a Java byte code optimization and annotation framework[A]. Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications[C]. Minneapolis, Minnesota, USA, 2000.113-114.

[18] NANDA M G, SINHA S. Accurate inter-procedural null-dereference analysis for Java[A]. Proceedings of the 31st International Conference on Software Engineering[C]. Vancouver, Canada, 2009.133-143.

[19] XIE Y, ENGLER D R. Using redundancies to find errors[A]. Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering[C]. Charleston, SC, USA, 2002.51-60.

[20] BAAH G K, PODGURSKI A, HARROLD M J. The probabilistic program dependence graph and its application to fault diagnosis[J]. IEEE Transactions on Software Engineering, 2010, 36(4): 528-545.

[21] ZHANG X Y, GUPTA N, GUPTA R. Pruning dynamic slices with confidence[A]. Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Impl- ementation[C]. Ottawa, Ontario, Canada, 2006.169-180.

[22] GUPTA R, SOFFA M L. Hybrid slicing: an approach for refining static slices using dynamic information[A]. Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering[C]. Washington, DC, USA, 1995.29-40.

[23] RENIERIS M, REISS S P. Fault localization with nearest neighbor queries[A]. Proceedings of the 18th IEEE/ACM International Conference on Automated Software Engineering[C]. Montreal, Canada, 2003. 30-39.

[24] JONES J A, HARROLD M J. Empirical evaluation of the Tarantula automatic fault localization technique[A]. Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering[C]. Long Beach, CA, USA, 2005.273-282.

[25] JONES J A, HARROLD M J, STASKO J. Visualization of test information to assist fault localization[A]. Proceedings of the 24th International Conference on Software Engineering[C]. Orlando, Florida, 2002. 467-477.

[26] SANTELICES R, JONES J A, YU Y B, *et al*. Lightweight fault-localization using multiple coverage types[A]. Proceedings of the 31st International Conference on Software Engineering[C]. Vancouver, Canada, 2009.56-66.

[27] 徐宝文, 聂长海, 史亮等. 一种基于组合测试的软件故障调试方法[J]. 计算机学报, 2006, 29(1): 132-138.
XU B W, NIE C H, SHI L, *et al*. A software failure debugging method based on combinatorial design approach for testing[J]. Chinese Journal of Computers, 2006, 29(1): 132-138.

作者简介：

姜淑娟（1966-），女，山东莱阳人，中国矿业大学教授、博士生导师，主要研究方向为编译技术、软件工程等。

王兴亚（1990-），男，山东淄博人，中国矿业大学博士生，主要研究方向为软件分析与测试。

张艳梅（1982-），女，河北唐山人，博士，中国矿业大学讲师，主要研究方向为异常处理、软件分析与测试等。

李威（1985-），男，江苏徐州人，硕士，主要研究方向为软件分析与测试、错误定位等。

鞠小林（1976-），男，江苏南通人，中国矿业大学博士生，南通大学讲师，主要研究方向为软件分析与测试。

刘颖祺（1992-），男，江苏徐州人，主要研究方向为软件分析与测试。