面向大规模 Java 代码的资源泄漏检测方法研究

高志伟

2018 年 1 月

中图分类号 TQ028.1

UDC分类号： 540

面向大规模 Java 代码的资源泄漏检测方法研究

作 者 姓 名 高志伟

学 院 名 称 计算机学院

指 导 教 师 计卫星

答辩委员会主席 石峰

申 请 学 位 工学硕士

学 科 专 业 计算机科学与技术

学 位 授 予 单 位 北京理工大学

论 文 答 辩 日 期 2018 年 1 月

Research on Resource Leak Detection for Large Scale Java Code

Candidate Name:	Zhiwei Gao
School or Department:	Computer Science & Technology
Faculty Mentor:	Prof. Weixing Ji
Chair, Thesis Committee:	Prof. Feng Shi
Degree Applied:	Master
Major:	Computer science and technology
Degree by:	Beijing Institute of Technology
The Data of Defence:	1，2018

研究成果声明

本人郑重声明：所提交的学位论文是我本人在指导教师的指导下进行的研究工作获得的研究成果。尽我所知，文中除特别标注和致谢的地方外，学位论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京理工大学或其它教育机构的学位或证书所使用过的材料。与我一同工作的合作者对此研究工作所做的任何贡献均已在学位论文中作了明确的说明并表示了谢意。

特此申明。

作者签名：_____　　签字日期：_____

关于学位论文使用权的说明

本人完全了解北京理工大学有关保管、使用学位论文的规定，其中包括：① 学校有权保管、并向有关部门送交学位论文的原件与复印件；② 学校可以采用影印、缩印或其它复制手段复制并保存学位论文；③ 学校可允许学位论文被查阅或借阅；④ 学校可以学术交流为目的, 复制赠送和交换学位论文；⑤ 学校可以公布学位论文的全部或部分内容（保密学位论文在解密后遵守此规定）。

作者签名：_____　　导师签名：_____

签字日期：_____　　签字日期：_____

摘要

随着云计算、大数据、人工智能等技术的不断发展与应用，软件系统的规模越来越庞大，参与程序设计与开发的人员越来越多，导致软件系统存在各种软件缺陷问题。在众多的软件缺陷和程序安全漏洞中，资源泄漏是影响软件系统稳定性和可靠性最重要的软件故障之一。目前，静态代码分析是最常见的用于检测资源泄漏的技术，该方法在检测软件缺陷时不需要运行程序，只是基于软件源码或者编译后的文件进行数据流分析或者缺陷模式匹配，进而发现软件缺陷。精确的资源泄漏检测算法依赖对整个程序进行精确的数据流分析，但是由于软件系统的复杂特性，在数据流分析中由于程序中过多的执行路径必将导致不可忍受的时间和空间消耗，无法满足在生产环境中对资源泄漏检测的实际需求。

本文基于静态代码数据流分析提出一种面向大规模 Java 代码的资源泄漏检测方案，该方案设计了应用于全局检测和增量检测的分析域提取策略，并基于 IFDS/IDE 数据流分析框架和别名分析算法实现了具有流敏感和路径敏感特性的资源泄漏检测算法。全局检测分析域提取策略应用于开发人员对项目进行全面的缺陷检测过程中，通过构建全局方法调用图、资源类图等策略，提取资源操作层次上的方法作为全局检测分析域，然后对分析域进行全面、高效的资源泄漏检测；增量检测应用于开发人员在开发过程中的即时检测过程，通过资源闭包、指向分析等方法提取当前修改方法在资源操作层次上的影响范围作为增量检测分析域，进而通过资源泄漏检测算法进行缺陷检测。通过实验评估，相比于 FindBugs 和 Fortify 等典型缺陷检测工具，本文中资源泄漏检测算法具有更高的准确率和较低的误报率；在增量检测中，在保证检测准确率的前提下，90% 的增量检测实验可以在 10s 内完成，能够满足在开发人员编辑代码过程中对缺陷进行即时检测的实际应用需求。

关键词： 缺陷检测；资源泄漏；指向分析；数据流分析

Abstract

With the continuous development and application of technologies such as cloud computing, big data and artificial intelligence, the scale of software systems is getting larger and larger, more and more personnel involved in the program design and development, so that various software defects arised in the software system. Among numerous software defects and program security vulnerabilities, resource leak is one of the most important software defects affecting the stability and reliability of a software system. At present, static code analysis is the commonest technique for detecting resource leak, and it can find defects without running a program when detecting software defects, and it only performs data flow analysis or defect pattern matching based on the software source code or the compiled file. Precise resource leak detection algorithm relies on accurate data flow analysis of the entire program. However, too many execution paths in the program will inevitably lead to unpleasant time and space consumption in the data flow analysis because of the complexity of software systems. and it can not meet the actual needs of resource leak detection in a production environment.

Based on the analysis of static code data flow, this paper proposes a resource leak detection scheme for large-scale Java code. The scheme designs the analysis area extraction strategy applying to global detection and incremental detection. Based on IFDS / IDE data flow analysis framework and Alias analysis algorithm, it achieves a resource leak detection algorithm having features of flow-sensitive and path-sensitive. The analysis domain extraction strategy for global detection is applied to conduct a comprehensive defect detection on the project. By constructing a global method call graph and resource class diagram, the method of resource operation level is extracted as a global detection and analysis domain, and then a comprehensive and efficient resource leakage detection is performed on the analysis domain. Incremental detection is applied to the development process of real-time detection process, through the method of resource closure, point analysis, method of resource operation attribute in the influence domain of the current modification method is extracted as an incremental detection and analysis domain. Through experimental evaluation, compared with typical defect detection tools such as FindBugs and Fortify, the resource leak detection algorithm in this paper has higher accuracy and lower false alarm rate. In incremental detection, under the

premise of ensuring the detection accuracy , 90% incremental test can be completed within 10s. And it can meet the real-time detection need of the actual application requirements in the process of editing code.

Key Words: Defect detection; Resource Leak; Point to analysis; Data flow analysis

目录

表格

插图

第 1 章　绪论

1.1　本论文研究的目的和意义

随着现代信息技术互联网科技的高速发展，人类社会已经步入一个高度信息化和智能化的时代，在这个时代，人工智能、数据挖掘、机器视觉等前沿技术不断推动着人类在交通、农业、国防、通信、金融、工业等各个方面步入一个高度智能化的时代。但不可否认的是，这些前沿技术发展的基础是计算设备，诸如手机、笔记本、天河系列计算集群等，将生活中的一个个任务逻辑转化为计算设备上的计算任务的过程就是开发人员开发程序的过程。现在计算机程序已普及到社会的各行业各领域，无论是人工智能、数据挖掘、机器视觉等前沿科技，还是较为成熟的网站开发、计算机系统和网络等，都是运行在计算设备上的程序指令。因此，基于程序开发的软件系统的可靠性和稳定性显得尤为突出。

在计算设备出现之初，软件也就随之出现，只不过早期的软件承载的只是简单的计算逻辑，只有简单的加减乘除运算，但是随着计算设备的不断发展，计算芯片遵循摩尔定律快速的更新迭代，计算设备上能够承载的计算逻辑也越来越复杂，解决的人类生活中的问题也越来越多。软件系统的规模越来越大，软件结构越来越复杂，随之而来的问题就是软件系统的可靠性和稳定性也越来越难保障。软件的质量问题、安全问题和可靠性逐渐成为制约计算机软件发展的瓶颈之一。例如，1996 年 6 月阿丽亚娜 5 型火箭发射由于软件缺陷引发的问题导致在发射 39 秒后偏离轨道，激活了火箭的自毁装置，造成 3.7 亿美元的损失。2009 年 2 月份，Google 邮箱由于数据中心之间的负载均衡软件存在问题导致 Gmail 用户几个小时不能访问邮箱。因此，研究如何保证软件的可靠性和稳定性具有很重要的意义[1]。

软件公司在软件开发的整个生命周期的各个阶段都会采用多种方法和手段保证软件的质量。通常，软件开发公司在软件开发周期中用于软件质量控制方面的资源投入非常大，甚至达到软件总体预算的 80%[2]。没有经过严格测试和质量检测的软件，其软件故障密度一般在 10 个故障/10KLOC，然而软件中存在的这些故障一旦发生，往往都会导致软件系统的崩溃。对于最初发布的软件版本，通常可能包含几千或上万个故障，因此，不可避免地给人们的生活和工作造成很多麻烦和不便。例如，苹果 IOS 系统版本 10 升级到版本 11 时，导致不同机型下存在的卡顿、电量消耗大、应用软件

强退等问题，使得该系统开发进入一个快速的缺陷修复期。游戏公司在上线一款游戏软件时，虽然经过了大量前期测试，但是由于市场设备的多样性，也会导致不同设备上会出现各种问题，影响用户的体验；在软件开发过程中也不乏严格保证软件质量从而赢得用户的案例，在 IE 开发期间，微软为了保证 IE 的可靠性，在数台计算机上持续运行一个星期，并且保证 IE 在几秒内可以访问数千个网站，而且安排测试人员对每个新版本做严格的测试，最终使得 IE 取得了很大的成功。

对于那些安全攸关的软件来说，软件故障的存在可能会造成非常大的经济损失和人身伤害。美国国家标准与技术研究所在过去曾经发布的一项研究表明，每年因为软件缺陷或故障对美国造成的经济损失高达 595 亿美元。由此可想而知，全球由软件故障导致的损失之高。此外，研究还表明，尽管测试不能排除所有的错误，但是通过良好的测试可以在早期开发阶段发现并减少 1/3 以上的故障，减少损失约 222 亿美元。由此可知，软件缺陷或故障的代价是非常高的。因此，如何检测软件中尽可能多的故障，提高软件的质量，已经成为软件开发过程中一个必不可少的环节，特别是安全敏感的软件系统，其稳定性和可靠性更是成为整个系统的关键属性。

在众多的软件缺陷和安全漏洞当中，资源泄漏[3][4]是影响软件系统稳定性和可靠性的重要软件故障之一。资源相关的缺陷是程序对资源对象在调用、分配和回收等操作上的错误导致的[1]。本文所提到的资源主要指的是系统资源，例如文件句柄、网络连接、数据库连接、远程方法调用等。资源泄漏故障作为影响软件稳定性和可靠性最危险的软件故障之一，其检测方案一直是研究人员研究的热点。存在资源泄漏的程序，随着程序长时间运行，泄漏的资源不断增加，将导致程序出现资源不足等异常，甚至导致软件崩溃。一项研究表明，导致软件系统停机故障的原因中高达 86% 是源于资源泄漏，其包含内存泄漏以及其他系统资源泄漏，如文件句柄、网络套接字、远程方法调用、数据库连接丢失等等。其次，资源泄漏也是应用服务型软件系统快速老化的主要原因之一。对于内存存在限制的软件系统（如嵌入式系统）或应用程序执行时间较长的系统（如服务器）而言，资源泄漏问题造成的后果尤为严重。例如 1992 年 9 月，由于一个救护服务系统中存在内存泄漏，在程序连续运行三星期后导致系统崩溃，从而引发了极危险的事故。另外，通过对 Linux 内核开发人员在 2008 到 2009 年统计的静态分析故障报告进行分析总结[5]，如表1.1所示，可以发现资源泄漏错误所占比例高达 1/4 左右，由此可知资源泄漏故障在软件系统中发生的比例非常高。

在软件开发过程中，软件可靠性和稳定性的保证要贯穿软件的整个生命周期，因

表 1.1　Linux 内核人员使用静态分析检测故障的统计 (前 6 项)

Defect Type	2008 Frequency	2009 Frequency	Ranking Change
NULL Pointer Deference	27.95%	27.81%	0
Resource Leak	25.73%	23.34%	0
Unintentional Ignored Expression	9.76%	9.71%	0
Use Before Test	8.09%	8.35%	-1
Use After Free	6.46%	5.91%	-1
Buffer Overflow	6.14%	5.79%	-1

此研究者针对软件的设计、编码、测试、维护等各个阶段进行了很多研究，提出了很多理论和方法来保证软件可靠性和稳定性。由于软件源码几乎包含了软件的所有细节，并且软件的开发过程由于开发人员的水平参差不齐，最容易引入软件缺陷问题，所以软件缺陷问题大约一半都是在源代码级别造成的。因此针对软件源代码的故障检测以及形式化验证是软件质量保障的重要手段，而且静态源代码分析允许我们在软件开发生命周期的早期阶段发现和修复缺陷，能够节省数以百万计的相关成本。

1.2　缺陷检测技术国内外现状

软件缺陷是影响软件可靠性和稳定性的重要因素，资源泄漏作为一种危险系数较高的软件缺陷备受关注。研究如何快速准确的发现软件系统中存在的缺陷十分必要。当前，软件缺陷检测技术按照是否需要程序执行分为静态检测和动态检测。

静态缺陷检测技术是针对源代码中存在的一些潜在语法或语义错误进行检测，如对可能为空的指针进行引用、对已分配的内存没有及时释放、使用未经过初始化的数据等问题，资源泄漏也是其中的一个重要的缺陷，其难点在于需要准确地计算出复杂的执行语义，典型的工具有 Klocwork 的 KlocworkInsight[6]、Coverity 的 Coverity Static Analysis、Fortify 的 Static Code Analyzer，GIMPEL SOFTWARE 的 PC-Lint[7] 和北邮的 DTS[8]。

动态缺陷检测技术主要包括单元检测技术、测试评估以及运行监控等技术。单元技术是基于被检测单元的源码和执行逻辑结构构建测试用例运行，从而检测软件缺陷，其难点在于对复杂的程序结构和路径进行求解的过程，现在比较有效的测试数据构建工具有贝尔实验室研发的 DART[9]、斯坦福大学的研究项目 KLEE[10]、伯克利大学的 CUTE[11] 以及国内北京邮电大学研发的 CTS[12]。测试评估是针对源程序结构的测试覆盖的充分度给与评估，其包含分支覆盖、语句覆盖、路径覆盖、MC/DC 覆盖、变量定

义和使用覆盖等，现在典型的工具有 Parasoft 系列的测试部分、MicroFocus 公司开发的 DevPartner 中的 Code Coverage Analysis、IBM 开发的 PurifyPlus 中的 PureCoverage、北航开发的 QESAT 中的统计覆盖功能以及北邮 CTS 中的覆盖统计功能；运行监控采用插桩或者变异的方法对程序源代码进行修改，在程序运行过程中收集错误执行状态，典型的工具有 Parasoft 公司的 Insure++、Agitar 公司的 AgitarOne、IBM 公司 PurifyPlus[13] 下的 Purify。

1.3 本文研究内容

针对常见的资源泄漏缺陷，本文设计并实现了面向大规模 Java 代码的资源泄漏静态检测方案，并结合实际需求，提供了针对大规模 Java 程序的全局检测方案和增量检测方案。本文研究内容如下：

(1) 基于 IFDS/IDE[14] 过程间数据流分析框架和别名分析算法设计并实现了跨过程的、流敏感的、路径敏感的资源泄漏检测方案；

(2) 全局检测方案中，针对程序中资源操作存在的特点，设计并实现了通过提取资源类方法构建资源泄漏检测分析域的方案；

(3) 增量检测方案中，从被修改方法开始，提取当前修改的方法在资源操作层次上的影响范围，构建资源泄漏检测分析域。

1.4 论文结构

第一章介绍了本文提出的资源泄漏检测的研究背景和意义，并简单阐述了国内外在软件缺陷检测技术上的发展现状，提出本文中面向大规模 Java 项目资源泄漏检测的研究目标。

第二章论述了国内外对缺陷检测技术的相关研究。从静态代码分析、动态软件测试和增量检测技术等方面对国内外研究学者的成果做了论述，并针对性的提出其存在的问题，最后介绍了工业界典型的三种软件缺陷检测的工具。

第三章介绍了面向大规模 Java 程序资源泄漏检测方案的整体架构设计，描述了架构中各部分的设计逻辑与实现基础，最后对全局检测方案中的检测分析域提取、别名分析和资源泄漏检测方法进行详细的方案设计阐述。

第四章介绍了面向大规模 Java 程序资源泄漏增量检测方案的设计，基于第三章的架构，提出在开发过程中从被修改方法开始逐渐构建精确分析域的方案，被详细描述了其中使用的资源闭包和指向分析等技术，最后阐述了增量检测的整个过程。

第五章对本文中提出的面向大规模 Java 程序资源泄漏检测技术进行实验评估，从全局检测和增量检测两方面分别评估了本文中资源泄漏检测的准确率和在增量检测过程中的实用性。

最后总结部分对本文本文中设计的面向大规模 Java 程序资源泄漏检测方案进行总结评估，确定本方案的可行性和创新性，并表明本方案存在的缺陷以及进一步研究目标。

第 2 章　相关工作

软件开发过程中，保证软件可靠性和稳定性要贯穿了软件的整个生命周期，因此研究者针对在软件的整个生命周期中保证软件质量进行了很多研究。资源泄漏故障作为影响软件可靠性和稳定性最危险的漏洞之一，研究如何快速准确的发现软件系统中存在的资源泄漏缺陷有非常重要的意义，众多国内外学者投入大量的资源进行研究，主要研究策略包含静态代码分析、动态软件测试等，并且随着在软件开发过程中缺陷检测越来越重要，一些面向缺陷的增量检测方案也逐渐出现。

2.1　静态代码分析技术

静态分析主要通过简化或者近似等策略对源程序进行分析。这种分析源程序的策略不依赖程序的执行状态，主要根据程序的运行逻辑即控制流图和程序结构进行分析，当前静态分析主要的方案为基于控制流图的数据流分析或者基于程序结构的缺陷模型匹配，这些策略由于不依赖执行状态会导致分析精度有所减低。近年来许多研究人员提出了多种基于形式化的缺陷检测静态分析方案，比如通过定理证明、抽象解释、约束求解等，这些方法通过采用一些数学上的形式化的概念和方法，将程序的逻辑和状态形式化的呈现出来，进而分析这些形式化的表示，通过这种数学形式化的处理，可以得到源程序中一些更精确和广泛的性质。此外还存在一些诸如符号执行和切片分析等的辅助分析策略，它们单独作为分析策略时，目的性不足，但是结合前面提到的分析方案则会获得较高的分析精度。随着对静态分析方案研究的不断深入，一些研究学者发现指向分析的重要性，并逐渐引入别名分析、指针分析、逃逸分析和形态分析，这类分析主要以基本的静态分析框架为基础来提高分析精度。例如 IBM 的 Emina Torlak 等人在 2010 年提出了一种方法间资源泄漏检测的方法[15]。基于 access-path 构建了一个在数据流分析中表示资源类对象的三元组，并随着数据流分析构建其在每条语句下的别名集，该方法在数据流分析过程中逐渐构建别名分析模型，使得静态分析的精度得到很大提高。

学术界和工业界围绕 C++ 和 Java 这些非常受欢迎的程序语言做了大量的研究，其中对这两种语言的内存泄漏和资源泄漏检测研究做了很多工作，并取得了卓越的成果。目前针对资源泄漏的静态分析方案主要分为两类：基于源程序的路径敏感检测方案和路径不敏感检测方案。Radu Rugina 等开发 LC 工具以及 SATC、Monica S.Lam 开

发的 Clouseau 等工具都是基于路径不敏感策略的内存泄漏检测方案，这类方案虽然检测效率非常高，但是通常误报率都在 50% 以上。但是对于一个程序进行完全的路径敏感检测分析，通常会由于路径爆照导致分析代价非常高，诸如大连理工大学开发的基于模型检测的 TMC，路径敏感分析的难点在于如何避免状态空间爆炸。SARURN 基于逃逸分析优化分析策略，该方案虽然保证了一定的分析精度，但是计算量太大，对于越来越大的软件项目来说，分析时间和空间都难以承担。

在实际缺陷检测过程中，对于路劲敏感分析都会采用相关的技术对其进行简化，即在分析过程中采用部分路径敏感。一些研究工作结合有限状态机对程序故障发生模式进行描述，根据程序数据流状态和状态机的状态在程序结构汇合的位置进行不同的合并方案，从而减少数据流分析的迭代次数，例如模型检测器 SLAM 和文件句柄静态验证工具 ESP。这种方法的优势是能够允许用户设定任意属性的状态机，并通过数据流分析策略检测属性的安全性，SLAM 通过谓词化简策略对状态跟踪过程改进，但是所采用的谓词抽象方案可能导致状态数量爆炸。肖庆和宫云战等人提出一种基于多项式复杂度的路径敏感缺陷检测方法[16]，该方法通过提取变量的抽象取值范围表示属性的状态；然后通过属性状态中变量取值为是否为空判断路径的可达性；在控制流图中的汇合节点将相同属性状态的属进行合并来降低计算复杂度。

为了提高静态分析效率，切片技术逐渐被引入到静态分析的控制流模型中。北京邮电大学赵云山在文献 [17] 中对文献 [16] 提出的方案进行改进，在静态分析过程中采用基于缺陷模式的切片方法，减少了控制流图中节点状态迭代计算的消耗。使用切片技术虽然对精度和效率有所改善，但是在控制流模型中产生很多冗余节点。Fastcheck 和 Saber 同样采用了切片技术，但是切片精度和控制流节点的数据流信息精度都有待提高，因此，如何采用切片技术对静态分析过程进行简化，以及采用什么准则保证产生切片的节点数少但是接近源程序的语义属性是应用切片技术的难点。

在静态分析过程中，如果存在资源行为切片，则需要考虑函数调用的问题。在跨过程间的资源泄漏检测分析研究方面，函数内联、提取函数调用栈以及函数摘要等都是比较经典的方案。函数内联通过在每次被调函数被主调函数调用时，将被调函数进行数据流分析，实现完全意义上的上下文敏感，即使该被调函数在之前已经被调用过，但是由于两次调用的上下文不同，所以依然需要进行分析，这种方案能够保证较高的上下文敏感性，但是会导致大量的冗余计算，且分析复杂度随程序规模呈现指数级增加，因此函数内联方案的效率非常低，导致其不能应用于大规模代码的分析中。此外

C 程序中能够通过函数指针对程序进行调用，Java 中通过反射机制加载类后对其方法进行调用等，都会导致在静态分析过程中不能够找到完整的方法调用关系，从而导致这些方法不能够通过内联的方法对其进行分析，此外递归函数的存在也会导致函数内联方案的复杂度急剧提高，函数内联和调用栈分析方案都属于穷尽计算的策略，因此效率都不高。

函数摘要在函数分析过程中将其相关行为和状态进行总结并表示为某种形式化的符号，如果后续分析中再次遇到该函数调用时，根据主调函数的上下文信息对其进行复用，减少被调函数二次分析造成的冗余计算，提高静态分析效率，目前函数间的静态分析策略多采用函数摘要方案。在提取函数摘要策略中，大部分采用提取输入到输出之间映射方案产生函数摘要。为了提高函数摘要对上下文的表现能力，研究人员提出一种基于符号的通过下推自动机构建函数摘要的方案，此外还有基于约束形式的函数摘要方法，但是约束条件在程序的语义表达上不足，并且在函数摘要实例化过程中依赖约束求解器的能力，所以在实际中可用性不高。文献 [18] 在函数分析过程中捕捉其内存操作行为作为函数摘要，没有考虑被调函数参数、返回值及其之间的关系，另外对于循环体，只迭代计算一次，会极大的影响分析的精度。北京邮电大学宫云战等人开发的 DTS 结合有限状态机提出一种内存错误检测的策略，并根据指针映射和区间运算给出内存泄漏检测的算法，在 DTS 中采用的函数摘要是精确的数据流值，但是静态分析中很难得到精确的数据流值的信息，分析过程中函数摘要丢失信息过多导致精度损失。

2.2 动态软件测试技术

缺陷动态检测方法是通过构建测试用例来测试程序是否存在缺陷，主要是实现单元测试的自动化，许多国内外研究人员投入大量的精力研究静态生成测试用例的方案。现在多数公司的软件测试工具套件都包含单元测试的功能，其先对软件静态分析，然后对程序中每个模块设计全面的测试用例进行测试，在动态测试过程中，单元测试效果在提高软件质量方面效果非常显著，但是，如果测试用例较多，测试的时间将会增加，在回归测试将会需要较长的时间。

在自动化测试用例生成的研究中，动态符号执行具有很大的潜力，但是它同样带来不容忽视的问题：路径数随着输入量和代码量的增加呈现指数级增加。Suhabe Bugrara 等人提出了检测动态符号执行过程中冗余状态的方法，将动态执行的效率提

高了很多倍，并且提高了测试用例的覆盖率。

从 2005 年开始，动态测试用例生成技术[17] 逐渐成为一个研究热点，该技术通过分析程序动态执行过程中的信息来得到可达路径与输入数据之间的约束关系，同时在受到输入数据控制的分支语句的跳转时，将约束输出到模型求解器[18][19][20]，从而判定另外一条路径的可达性，如果该路径可达，则求解器生成覆盖目标路径的测试用例，然后利用生成的测试用例对软件系统进行测试。典型的动态测试用例生成系统包括微软亚洲研究院的 DART[9] 和 SAGE[21]、斯坦福大学的 EXE[22]、伯克利大学的 SmartFuzz[23] 等

斯坦福大学研究的 EXE 系统主要用于组件测试，它的优势在于动态测试过程中收集路径可达性约束条件和约束路径求解能力。EXE 的基础是一个源到源的编译器，该编译器在编译过程中通过插桩的方式将源码中的每一个赋值和运算前都插入符号化执行需要的函数调用，并在程序中的分支语句前插入调用求解器的代码，主要用于对当前分支条件求解并构建测试用例。所以在 EXE 系统中，程序执行和符号化执行是交替执行，符号化执行所需信息从程序的执行过程中获取，从而收集程序执行路径的约束条件。在 EXE 中实现了 SMT 求解器 STP[24]，STP 包含了 BV[25][26],、ARRAY[27] 和整数理论[28][29]。在 EXE 中，路径可达性约束的收集和求解比较精确，故系统生成的测试用例比较全面。但是 EXE 系统依然存在以下不足：

(1) EXE 系统分析时需要完整的源码支持，由于 EXE 的检测基于源到源的编译，所以系统需要对能够编译成功的源码进行检测，也就是说如果被测试的软件中应用了第三方的库，但是没有源码，则 EXE 的检测将不能使用，因此 EXE 的使用受到很大局限。

(2) EXE 只能对小规模的应用程序进行检测[21]。在 EXE 的检测过程中，会将所有的输入数据都当做恶意输入，这对符号化执行和路径的约束求解将会造成极大的压力，所以 EXE 的测试规模受到限制。

KLEE[10] 是斯坦福大学既研发 EXE 之后研究的动态测试工具，是 EXE 的升级版本。FLEE 将 EXE 系统中源到源的编译器改为 LLVM 编译器[30]，这个改变使得 KLEE 能够处理各种 LLVM 能够编译的语言，相对于 EXE 具有更广的应用范围。另外 KELL 通过分析约束和变量间的相关性，将约束表达式拆分为相互独立的子集，从而提高约束求解的效率。

SAGE 是微软研发的一个高效的动态测试用例生成系统，其直接对编译后二进制程序进行动态跟踪，生成测试用例。SAGE 基于动态二级制指令追踪系统 Nirvana[31] 监控程序的执行过程并追踪指令执行过程中的执行流日志，然后对日志文件进行离线的符号化分析执行，构建执行路径的可达性约束条件，为输入运行可控的分支路径生成测试用例。与 EXE 不同的是，SAGE 为了能够应用于大规模的程序测试，采取了以下方案：（1）SAGE 忽略了对非线性约束条件的收集，简化了约束收集和求解的复杂性；（2）SAGE 忽略了程序中的指针分析。通过这两项措施，SAGE 系统加快了对大型目标软件的测试用例生成速度，但是在一定程度上损失了测试用例对路径的覆盖率。伯克利大学的 SmartFuzz 与 SAGE 类似，但是 SmartFuzz 采用 Linux 下的二进制执行追踪系统 Valgrind 追踪程序的执行状态并收集约束条件，不足的是 SmartFuzz 系统现在支持整数溢出的发掘。

2.3 增量检测技术

在开发过程中即时进行检测的增量检测技术近年来逐渐被人关注，比较典型的增量检测技术是 Lisa Nguyen Quang Do 等人提出了分层检测的基本思想[32]，将缺陷的检测分为方法、类、文件、包和项目等多个不同的层次。在检测的过程中，从当前编辑的方法内开始，依次从方法、类、文件、包和项目进行检测，并针对每一层的检测及时报告发现的缺陷。分层检测的思想能够对较小范围内的缺陷进行即时检测并报告给用户。但是也存在误报和漏报的情况，主要原因是该增量方案从当前方法进行前向的检测分析，没有考虑当前代码的修改对数据流分析时上游的影响。以如下的代码片段为例，当用户正在编辑 foo 方法时，如果从当前方法开始向下游分析，会发现所有执行路径中 input 都被关闭，不存在资源泄漏发生，但是结合上游的 main 函数一起进行检测分析，则会发现 input 在 foo 方法中被改写了一次，且改写之前 input 指向的资源变量并没有被关闭，所以资源泄漏发生与否不仅取决于当前函数以及当前函数的下游函数，还包括了与这个函数相关的上游函数。文献 [34] 所给出的算法只考虑以当前函数为起点向下调用的函数，忽略其上游的函数，从而会造成误报或者漏报的情况；另外，该方法随着检测层次的逐渐增加，检测的时间也会增加。现在，已有的商业工具更多采用批量式检测方法，即从程序的 main 函数入口进行检测分析，从而不存在上游函数调用的问题，但是检测的时间和空间消耗同样是巨大的。

```
1  public class Main {
2      public static void main(String[] args) throws IOException {
3          FileInputStream input = new FileInputStream(new File("1.txt"));
4          foo(input);
5      }
6      public static void foo(FileInputStream input) {
7          try {
8              input = new FileInputStream(new File("2.txt"));
9              foo1(input);
10         } catch (IOException e) {
11             e.printStackTrace();
12         }
13     }
14     public static void foo1(FileInputStream input) {
15         try {
16             input.read();
17             input.close();
18         } catch (IOException e) {
19             if(input!=null){
20                 try {
21                     input.close();
22                 } catch (IOException e1) {
23                     e1.printStackTrace();
24                 }
25             }
26         }
27     }
28 }
```

2.4 工业界缺陷检测工具简述

资源泄漏故障是影响软件稳定性和可靠性最危险的漏洞之一，在工业界也对缺陷检测做了大量的开发设计工作，比较著名的缺陷检测工具为 FindBugs[33]、Fortify[34]和 Coverity[35]。

FindBugs 是出现相对较早的一个面向 Java 的缺陷检测工具，因其轻量、高效的特点被广泛使用。它通过事先从存在缺陷的代码中发掘各种缺陷的发生模式，并构建

检测这类缺陷的规则，在代码检测过程中，将字节码与这些缺陷规则进行快速匹配来发现源码中可能存在的缺陷。目前 FindBugs 已经支持包括资源泄漏、空指针引用、跨站脚本、SQL 注入在内的多达 60 种问题。FindBugs 是基于 Java 生成的字节码文件进行分析检测的，所以是一种静态检测工具。但是由于 FindBugs 使用的是基于规则的检测策略，对跨方法间的缺陷检测支持不足，所以检测的准确率较低。

Fortify 是全球领先的软件安全产品解决供应商，其公司推出的用于安全检测的工具包括 Fortify SCA[36]，Fortify Manager[37]，Fortify Tracer[37] 和 Fortify Defender[37]，这些产品可以帮助客户在软件开发周期中建立安全机制，保证软件的安全问题。Fortify SCA 是一个静态的、白盒的基于软件源码的安全检测工具。它通过内置的 5 大分析引擎：数据流、语义、结构、控制流和配置流等对应用软件的源码进行静态分析，在分析过程中与软件安全漏洞规则进行匹配、分析，从而找出源码中存在的安全漏洞，并生成缺陷报告，Fortify 是一个相当成熟的软件安全和缺陷检测工具，被多个大型公司引入进行内部的代码安全检测。

Coverity 公司是斯坦福大学的科学家于 2002 年成立的，公司的核心产品就是用于检测软件系统中关键缺陷的系统。Coverity 是唯一一位在 IDC 前十名软件质量工具供应商的静态分析工具厂商，在市场占有率上处于绝对领先地位。Coverity Static Analysis 是检测和解决 C、C++、Java 和 C# 源码中严重缺陷的自动化方案；它将基于布尔可满足性的验证技术用于源码的分析引擎中，利用其特有的的 DNA 图谱技术和 meta-compilation[38] 技术，综合分析软件源代码和编译构建系统，进而发现软件中存在的缺陷。Coverity 解决了很多影响源代码分析有效性的关键问题，比如不同构建集成环境导致的软件安全性问题、不同平台下的编译兼容性和分析有效的错误根源问题等。

2.5 本章小结

本章从静态代码分析、动态软件测试、代码增量检测技术三方面讨论了国内外缺陷检测技术的相关研究，并针对典型的缺陷检测技术，分析其优点与不足；最后介绍了工业界典型的缺陷检测工具 FindBugs、Fortify 和 Coverity。

第 3 章　大规模代码资源泄漏检测

在程序运行中，存在某些系统资源如文件句柄、数据库连接、套接字连接、远程方法调用、图形光标等在系统中是受限的。如果这类资源被分配后，并在后续执行过程中，该资源没有被释放，则该程序存在资源泄漏故障（Resource Leak）。

Java 虚拟机（Java Virtual Machine，JVM）采用垃圾回收器（Garbage Collector，GC）负责内存回收，从而使开发者在开发过程中把精力集中在业务逻辑上的实现上，不需要过多考虑内存的合理利用，因此往往忽视释放程序占用的系统资源。实际上 GC 的触发条件通常与内存使用有关，并不依赖于系统资源情况。GC 启动时具有不确定性，如果 JVM 有充足的可用内存，GC 很可能不会自动触发，然而系统资源比内存更稀缺，例如所有的操作系统都限制可同时开启的套接字、文件句柄的数量。此外，通过文件句柄、网络套接字等占用的系统资源在 GC 过程中是不会被释放的，所以，存在资源泄漏的程序在长期运行过程中，由于申请的系统资源没有释放，会逐渐占用更多的系统资源，直至系统资源耗尽，导致运行程序奔溃，因此 Java 程序中的资源泄漏是一类风险较高的代码缺陷。

为了发现资源泄漏在代码中存在的特点，制定检测资源泄漏的方案，本文通过对存在资源泄漏的 Java 代码进行分析，总结出常见的资源泄漏主要分为以下三种情况:

(1) 方法内部资源泄漏：由方法内代码形成的程序控制流图中存在某个资源对象申请资源后没有被释放。如下代码示例，fis 在申请资源后，还没有进行资源的释放，就被赋值为一个新的资源对象，之前的系统资源没有被释放，故存在资源泄漏。

```
1   public static void foo() throws IOException {
2       FileInputStream fis = null;
3       try{
4           fis = new FileInputStream(new File("in.txt"));
5           fis = new FileInputStream(new File("in1.txt"));
6       } finally {
7           if( fis !=null) fos.close();
8       }
9   }
10
```

(2) 私有域资源泄漏: 系统资源被分配在类的私有域中，并且该类中没有方法释放这个资源，或者释放资源的方法是私有的且没有直接或间接被公有方法调用。如下代码中，fis 作为 $IOStreamtest3$ 的一个私有成员存在，在所有方法中，并没有被显式的调用关闭操作，故存在资源泄漏。

```
1   public class RLField {
2       private FileInputStream fis;
3       public FileOutputStream fos;
4       public FileInputStream getFis() {
5           if(fis==null)
6               try {
7                   fis = new FileInputStream(new File("in.txt"));
8               } catch (FileNotFoundException e) {
9                   e.printStackTrace();
10              }
11          return fis;
12      }
13      public static void main(String[] args){
14          RLField rlField = new RLField();
15          FileInputStream in = rlField.getFis();
16          try{
17              rlField.fos = new FileOutputStream(new File("outn.txt"));
18          }catch (IOException e){
19          } finally {
20              if(rlField.fos!=null)
21                  try{
22                      rlField.fos.close();
23                  }catch(Exception e1){
24                  }
25          }
26      }
27  }
28
```

(3) 初始化资源泄漏：新分配的资源对象作为参数创建一个新的对象，未保存资源对象的引用，如果创建新对象失败，将无法释放该资源。如下代码，假如在 $FileOutputStream$ 的对象被创建后，在创建 $PrintWriter$ 的对象时，程序抛出异常，由于 $FileOutputStream$ 的对象没有保存其引用，导致不能够在异常捕捉

后对其进行释放，从而引发资源泄漏。

```
1   public void foo(){
2       PrintWriter out = null;
3       try {
4           String file = null;
5           try{
6               String enc = null;
7               out = new PrintWriter(new OutputStreamWriter(new
        FileOutputStream(file),enc));
8           } catch (UnsupportedEncodingException e) {
9               out = new PrintWriter(new FileWriter(file));
10          }
11      }catch (IOException e){
12          if(out!=null)
13              out.close();
14      }
15  }
16
```

针对常见的资源泄漏类型，本文设计并实现了面向大规模 Java 代码的资源泄漏检测方案，并结合实际需求，提供了针对大规模 Java 程序的全局检测方案和增量检测方案。本部分主要阐述了本文设计的面向大规模 Java 程序的检测方案的系统架构设计与全局检测方案的设计与实现。对于面向大规模 Java 程序的增量检测方案设计与实现将在第四章进行阐述。

3.1 系统整体设计

本文中设计的面向大规模 Java 程序的资源泄漏检测系统设计如图3.1，其处理过程主要包含五部分：输入、基础构建、分析域提取、资源泄漏分析和结果显示等过程，各部分功能如下：

(1) 输入部分：Java 字节码指的是被检测项目编译后生成的 class 文件；lib 指被检测项目运行时需要的第三方 jar 包，源码即被检测项目的 Java 源码。

(2) 基础构建部分：本系统中，资源泄漏检测需要的基础模块包含程序解析模块、方法调用图生成模块、方法内控制流图构建模块和指向分析模块等，本部分基于

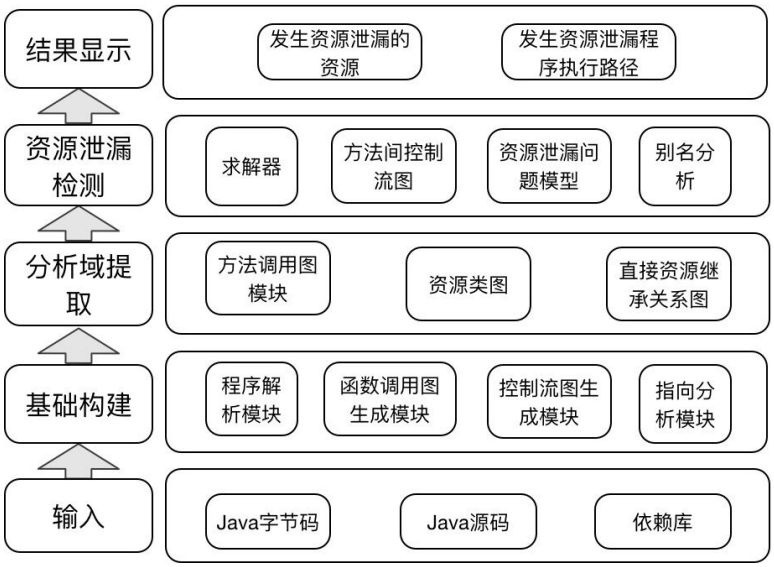


图 3.1　资源泄漏检测系统架构

开源项目 Soot 分别构建程序解析模块、方法内控制流生成模块和指向分析模块，方法调用图生成模块基于程序解析模块构建。

(3) 分析域提取：分析域为进行资源泄漏检测的方法集，无论是全局检测还是增量检测，都需要经过分析域的提取。本文基于全局方法调用图，全局资源类图，全局基础资源继承关系图，通过过滤、资源闭包、指向分析等手段提取资源泄漏检测的分析域。

(4) 资源泄漏分析：本部分基于 IFDS/IDE 数据流分析架构和别名分析算法设计了流敏感、路径敏感的资源泄漏检测方案，主要模块包括资源泄漏问题模型、方法间控制流图生成、求解器和别名分析等。

(5) 结果显示：对于结果显示，本系统为了便于开发人员理解资源泄漏发生的整个过程，除了显示资源泄漏发生时资源的类型和申请资源的位置，还基于分析过程给出了资源从申请到发生泄漏的整个执行路径。

面向大规模 Java 程序的资源泄漏检测系统包含两部分功能，即针对 Java 程序的全局检测过程和增量检测过程，分别对应于软件开发过程中两个不同时间段的缺陷检测。

在一个软件开发完成后或者开发人员在完成阶段性的任务时，会对项目进行一个全面的质量评估，这一过程对应的一般都是针对项目的全局检测，因此本文设计的面

向大规模 Java 项目的资源泄漏全局检测也应用于这一过程。在这一过程中，检测消耗的时间和空间相对较大，检测的准确性要求较高。

本系统中，在开发人员提交全局检测申请后，会读取项目的 class 文件，依赖的 jar 文件以及源码文件作为输入，经过程序解析模块，会将代码转化为一种便于分析的中间表示格式 Jimple，同时构建能基于 Jimple 代码的方法内控制流生成模块，方法调用图生成模块和指向分析生成模块，本系统中指向分析模块基于 Soot 提供的 spark 实现；在基础部分构建完成之后，基于上一部分的程序解析模块、方法调用解析模块和方法内控制流图生成模块构建全局方法调用图、全局资源类图，然后根据方法调用图和资源类图进行分析域的提取。最终提取到的分析域是一系列方法簇，每个方法簇中的所有方法具有可达性。资源泄漏检测都是将这些方法簇作为输入的，每个方法簇进行一次基于跨过程间的资源泄漏分析检测，最后汇总得到资源泄漏检测的结果。

资源泄漏的增量检测方案应用于开发人员开发过程中的即时缺陷检测。在开发人员进行开发时，如果能够即时的检测并返回缺陷检测结果，这种情况下，开发人员能够即时修复缺陷，可想而知，这种修复代价是非常小的，但是应用于开发人员开发过程中的即时检测需要的时间和空间消耗容忍限度是比较低的，也就是说，在个人开发电脑上，每次增量检测在不影响开发体验的情况下快速完成，因此，应用于开发过程中的增量检测方案设计是非常难的。

本文中设计的面向大规模 Java 代码的资源泄漏增量检测方案主要思想是通过各种手段分析当前修改对整个项目的影响范围，提取一个资源泄漏检测可容忍的影响范围作为分析域，进而进行资源泄漏的检测分析，因此增量检测方案设计的难点是精确的分析域提取。在系统设计中，增量检测的输入是开发人员修改的类的字节码文件，在经过 Soot 的增量加载和解析后，动态维护全局方法调用图，资源类图，这一部分需要引入直接资源继承关系图和指向分析模块，通过对被修改的资源类进行资源闭包分析和对其对象实例的指向分析，获取一个较为准确的影响范围作为分析域，进而通过资源泄漏检测算法进行分析检测，快速获取检测结果。

本系统中基础构建模块主要基于 Soot 进行实现，Soot 提供了面向 Java 的程序解析模块、控制流生成模块和指向分析模块，基于此，本文实现了方法调用图构建模块。资源泄漏检测部分则主要基于 IFDS/IDE 数据流分析框架实现，本文在研究了资源泄漏发生的特点之后构建了检测资源泄漏需要的问题模型、方法间控制流构建模块、求解器以及别名分析方案。

3.2 Soot 与 IFDS

3.2.1 Soot

Soot[39] 是一个基于 Java 语言的开源编译器框架，用于优化 Java 字节码。起初，研究人员只是为了开发一种用于指针分析比较的工具，现在随着 Soot 的不断优化改进，Soot 已经成为一个广泛应用于 Java 程序分析和优化的通用框架。Soot 提供 4 种类型的代码中间表示形式：Baf[40]，Jimple[41]，Shimple[42] 和 Grimp[43]。这些中间表示提供了不同级别代码的抽象，并面向不同的应用需求，Soot 的工作流图如图 3.2。

Jimple 是 Soot 的核心，是四种代码中间表示中最重要的。在 Soot 中，Jimple 代码由字节码转化翻译而来，字节码会被首先翻译成无类型的 Jimple 表示，再通过类型引用为其添加类型说明。翻译中非常重要的一步是对表达式作线性化，使得每个表达式只能最多使用 3 个局部变量或者常量。相对于字节码的 200 多种指令，Jimple 只有 15 条，分别是核心指令 $NopStmt$、$IdentityStmt$ 和 $AssignStmt$；方法内控制流指令 $IfStmt$、$GotoStt$、$TableSwitchStmt$ 和 $LookUpSwitchStmt$；方法间控制流指令 $InvoeStmt$、$ReturnStmt$、$ReturnVoidStmt$、监视器指令 $EnterMonitorStmt$ 和 $ExitMonitorStmt$ 和处理异常指令 $ThrowStmt$ 和退出的 $RetStmt$。

Jimple 是一种 Java 源码和字节码的混合体。对于局部变量的声明和赋值表达式用的是 Java，而控制流和方法调用采用的是字节码。在每个方法的 Jimple 表示中，会把所有用到的局部变量和栈位置做出声明，即变量到内存位置的映射，把变量映射到栈中的地址并在方法体中用地址替代变量名。带‘$’的局部变量表示是栈中的位置而不是原 Java 源码中真正声明过的局部变量，其表示在 Java 源码中不出现而在字节码中不可缺少的隐含变量以及储存中间结果的变量等。反之不带‘$’的即与原 Java 源码中局部变量相对应。

本文中设计的面向大规模 Java 代码的资源泄漏检测方案，即以 Soot 将字节码转换为的 Jimple 表示为分析的基础，并根据其构建控制流图，进而沿着控制流图进行数据流分析，检测资源泄漏。

3.2.2 IFDS

IFDS[14] 是 Interprocedure Finite Distributive Subset 的缩写，其描述了一种跨方法的、有限的、分布式子集问题，在这个问题中，f 被描述为分析域上的流处理方法，若

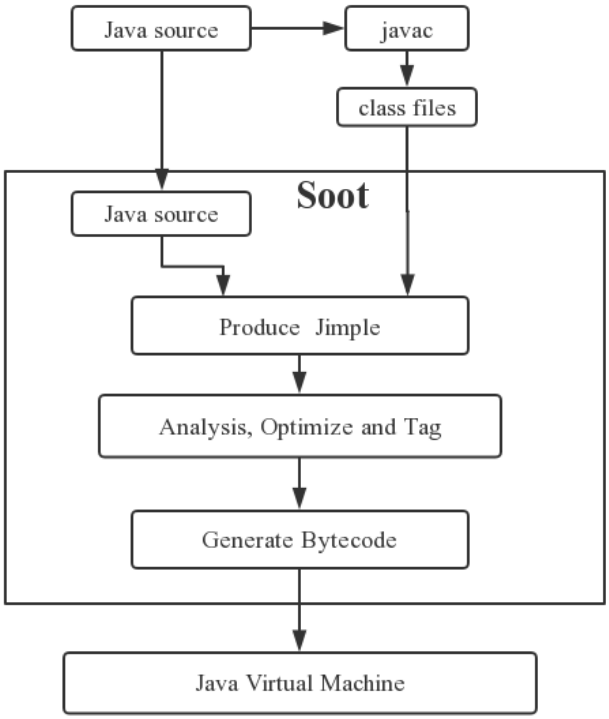


图 3.2　Soot 工作流图

“⊓”被定义为流处理过程中的合并操作，则对于在 D 中存在的任何两个数据 a 和 b，存在 f(a)⊓f(b)=f(a⊓b)。许多具有分布流数据处理特性的数据流分析问题都可描述为 IFDS/IDE 问题。

IFDS 数据流分析框架的主要思想是把所有的满足 IFDS 特性的程序数据流分析问题转化为一个图内节点的可达性问题。基于程序的方法间控制流图，IFDS 创建了一个超图，在这个图中，从一个开始节点到任意一个节点 (s，d) 可达当且仅当数据事实 d 在 s 中。为了实现这个目标，IFDS 将数据流分析问题定义为超图中的边集和点集构建问题。为了把数据流分析问题转化为图的问题，首先需要用传递函数将问题表达成一种二元关系，如图 3.3。

图 3.3 中 0，a，b 表示三个数据流事实。0 表示一个总是有效的空事实, 其用来产生新的有意义的数据流事实。数据流事实代表了对应变量的在所关注问题下的状态，如表示某个变量是否被声明过和初始化。边代表数据流事实的流向，如 $gen, kill$ 和传递。λ 运算中‘.’左侧的 S 表示输入集合，右边表示对 S 的操作。图 3.3 中最左边一列表示 $identity$ 函数的情况，即对 S 无影响，数据流事实在语句执行前后没有发生变化；图 3.3 中间一列表示 $gen(a)$ 的情况, 即设定原来是空集，经过此函数后并入数据

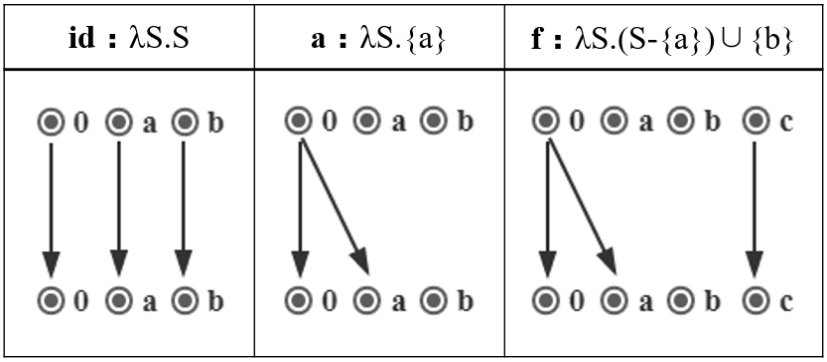


图 3.3　传递函数的二元关系表示

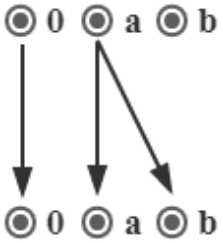


图 3.4　由于别名构建的传递函数的二元关系表示

流事实 a，在图中用 $0 \rightarrow a$ 的边代表新生成有关 a 的数据流事实并把它的值设定为 S 所代表的值；图3.3中最右边一列表示 $kill(a)$ 和 $gen(b)$，原先的 $a \rightarrow a, b \rightarrow b$ 的线被切断，表示经过函数后，数据事实 a 和 b 都发生变化，数据事实 a 不再被包括在内，数据事实 b 被包括在内（与 S 所代表的值相同），而数据事实 c 没有受到任何影响，向下继续传递。

除了以上流处理方法，在 IFDS 中还会遇到一种方法，如图 3.4 ，表示 a 的值不变，而把 a 的值赋给 b。相当于给 a 创建了一个别名 b，这里的值不是表示变量在程序中真正的值是多少，而是指有关此变量的数据流事实的值是什么，如 a 的值可以表示为被初始化和未被初始化两种。

Eric Bodden 等人在 Soot 上实现了一个基于 IFDS 问题，用于过程间数据流分析的框架。这个框架将程序数据流分析的问题，转化为 IFDS 中图可达问题进行求解。这个 IFDS 框架通过图转化的方式解决了过程间数据流分析的难题。算法的时间最差复杂度可达到 $O(ED^3)$，其中，E 是程序中控制流的边数，D 是分析域变量集合的大小，算法的有效性依赖于域 D 的大小。

在 IFDS 中，数据流分析过程被定为 4 中不同的边：Call Edge、Return Edge、Call-to-Return Edge 和 Normal Edge，每种边对应的处理逻辑如下：

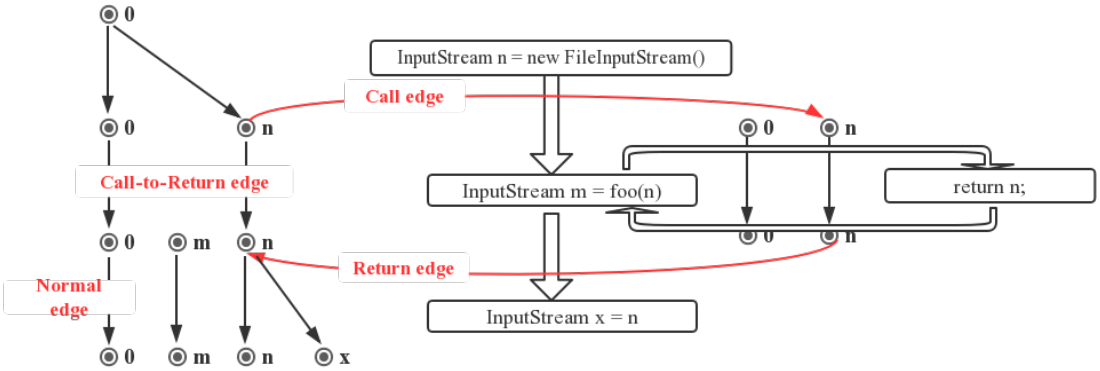


图 3.5　基于 IFDS 的数据流分析过程示例图

- Call Edge: 出现在方法调用过程中，连接方法调用语句和被调方法的边，主要考虑方法调用过程中参数的传递对数据流事实的影响；
- Return Edge: 从被调方法返回主调方法的边，主要考虑被调方法返回值对数据流事实的影响；
- Call-to-Return Edge: 从方法调用语句到被调方法返回的后继语句的边，主要考虑不经过方法调用，而直接向下流动的数据流事实；
- Normal Edge: 连接方法中非方法调用节点和其直接后继的边。

如图3.5，体现了 IFDS 中问题模型构建时实现的四个主要接口对应边的处理逻辑，图中 $0, n, m, x$ 均表示数据流分析过程中的数据事实，在开始分析时，数据事实集中只包含数据事实 0，在遇到资源申请语句时，将申请的资源作为数据事实添加到数据事实集中，在遇到方法调用语句时，将对应的数据事实传递到被调方法中，继续进行数据流分析，被调方法分析完成后，将需要的数据事实传递回主调方法继续数据流分析。本文资源泄漏检测算法的实现即通过将资源泄漏问题抽象成一种 IFDS 问题，实现其问题模型和求解器，并结合别名分析提高其检测的准确率。

3.3　分析域提取

根据 IFDS 进行数据流分析时的复杂度 $O(ED^3)$，随着分析域的增加，IFDS 分析时消耗的时间资源将会线性增加，而且在本文的设计方案中，分析域越大，需要分析的数据流将会越多，因此在针对大规模 Java 项目的检测时，消耗的空间和时间资源将

表 3.1　方法的调用指令

方法调用指令	被调方法说明
invokestatic	调用静态方法
invokespecial	调用实例的父类方法、私有方法及实例构造器方法
invokevirtual	调用实例中所有的虚方法
invokeinterface	调用接口方法
invokedynamic	调用动态方法

会非常大。然而在 Java 项目中，对资源的操作是相对局限的，如果通过 IFDS/IDE 对整个项目进行数据流分析，会耗费大量的时间进行冗余检测分析，极大地降低了检测效率。通过分析 Java 项目中资源使用的特点，本部分设计了一种面向大规模 Java 项目资源泄漏检测分析域提取的方法，旨在减小数据流分析时的冗余分析，提高检测的效率。

分析域提取方法的根本策略是提取项目中能够操作资源的方法进行检测。为了实现这一目标，需要在项目加载过程中完成一些分析准备工作，即全局方法调用图的构建和全局资源类图的构建，构建完成后，根据全局方法调用图和全局资源类图提取项目中需要被分析的方法。

3.3.1　全局方法调用图构建

定义全局方法调用图 $G(M,E)$，其中 M 表示项目中所有的方法（抽象方法和接口除外），E 表示方法之间的调用关系。G 构建出了项目中所有方法的上下文关系，从一个方法 m_i 出发，可以找出调用 m_i 的方法集 $caller(m_i)$ 和被 m_i 调用的方法集 $callee(m_i)$。

在 Soot 加载类过程中，会将每一个类封装为一个 $SootClass$ 对象，$SootClass$ 中包含了这个类里面的所有成员定义和所有方法的 Jimple 表示。本文构建全局方法调用图的过程就是基于每个方法的 Jimple 表示进行的。在 Soot 加载完类文件后，通过 Soot 的 API 获取到所有的 $SootClass$ 对象，然后分析对象中每个方法的 Jimple 代码，对于代码中每个方法调用语句，获取其调用方法的目标位置，构建这两个方法间的调用边，并完善每个方法的 $caller$ 集合和 $callee$ 集合。

在 jdk1.7 之后，方法的调用指令见表3.1，invokestatic、invokespecial、invokevirtual 和 invokeinterface 的分派逻辑是固化在 Java 虚拟机内部的，而 invokedynamic 指令

的分派逻辑是由用户所设定的引导方法决定的，也就是在运行时动态解析出调用点限定符所引用的方法，然后再执行该方法。由于本文使用的是静态分析，所以通过invokedynamic 调用的方法是不能够在静态分析中找到被调用的目标方法的，所以本文关注的方法调用指令只包含 invokestatic、invokespecial、invokevirtual 和 invokeinterface。

```
1  $r2 = new java.io.FileInputStream;
2  $r3 = new java.io.File;
3  specialinvoke $r3.< java.io.File: void <init>(java.lang.String)>("");
4  specialinvoke $r2.< java.io.FileInputStream: void <init>(java.io.File)>($r3);
5  r1 = $r2;
6  r5 = r1;
7  virtualinvoke r1.< java.io.InputStream: int read()>();
```

如上 Jimple 格式代码，在根据代码构建方法调用图时，会对方法内的每一条语句进行分析，判断如果当前语句是方法调用语句，即包含 invokestatic、invokespecial、invokevirtual 和 invokeinterface 的调用语句，如第 7 行代码，即为方法调用语句，方法调用方式为 virtualinvoke，接着提取执行调用的对象，即 $r1$，这一步并不能确定 $r1$ 对应的类，所以接下来会进行后向分析，直到分析到 $r1$ 初始化的那一句代码，即代码块中第 1 行（这里 $r2 与 $r1$ 存在别名关系），可以发现，$r1$ 被初始化为 $FileInputSream$ 的对象，现在可以确定第 7 行代码中 $read$ 方法对应的是 $FileInputStream$ 的成员方法 $read$，从而构建了从 foo 到 $FileInputStream.read$ 方法的调用边。具体的全局方法调用图的构建算法设计见算法3.1，$backAnalysis(obj)$ 是获取对象 obj 实际数据类型的方法，$getMethod(className, methodName, invokeType)$ 是 obj 对象在 invokeType 方式下调用的方法名称为 methodName 的方法的实体，其中 obj 对象是类名为 className 的实例，可以通过 $Soot$ 提供的接口获取。

在算法3.1中，依次分析项目中每一个方法中的每一条语句，如果当前语句是方法调用语句，获取该语句中执行调用的对象的实际数据类型、该语句中被调用方法的方法名以及方法调用类型，最后根据这三个参数到 Soot 解析后的结果中提取该语句实际调用的方法对象实体，构建方法之间的调用关系。当所有方法分析完成后，方法集合和调用关系构成的边集组成全局方法调用图 $G(M, E)$。

Algorithm 3.1 构建全局方法调用图 G(M,E)

Input: 项目中方法集合 MS，
Output: 全局资源类图 C(N,S)

$M \leftarrow MS$
$E \leftarrow \emptyset$
for each $m \in MS$ **do**
  for each $u \in MS.units$ **do**
    if $u\ instanceof\ invoke$ **then**
      $r = u$ 中执行调用的对象变量
      $methodName = u$ 中被调用的方法名字
      $className = backAnalysis(r)$
      if $u.invoke\ instanceof\ invokestatic$ **then**
        $invokeType = invokestatic$
      else if $u.invoke\ instanceof\ invokespecial$ **then**
        $invokeType = invokespecial$
      else if $u.invoke\ instanceof\ invokevirtua$ **then**
        $invokeType = invokevirtua$
      else if $u.invoke\ instanceof\ invokeinterface$ **then**
        $invokeType = invokeinterface$
      $m' = getMethod(className, methodName, invokeType)$
      $E \leftarrow E \cup \{(m, m')\}$
return $G(M, E)$

3.3.2 全局资源类图构建

在介绍全局资源类图之前，首先定义直接资源类，在本论文中，项目中的直接资源类包含以下两部分：

(1) 项目中涉及到的 Java 中提供的资源操作的类和接口被称为直接资源类；

(2) 项目中继承和实现直接资源类的类是直接资源类。

根据直接资源类的定义，定义全局资源类图 $C(N, S)$，其中 N 表示资源类的集合，S 表示资源类之间的包含关系。BR 表示每个资源类直接或间接操作的直接资源类的集合，给定资源类 A 和资源类 B，若 A 的成员变量中存在一个 B 类的对象，则资源类包含图中有一条边 s 由 A 指向 B。若 B 是直接资源类，而 A 是项目中的自定义类，则 A 称为间接资源类。如图3.6，$java.io.InputStream$ 和 $java.io.OutputStream$ 是直接资源类，A,B 和 C 为间接资源类，D 和 E 为非资源类。BR_A 表示 A 直接或间接操作的直接资源类集合，即 $\{java.io.InputStream□java.io.OutputStream\}$，同理 BR_B

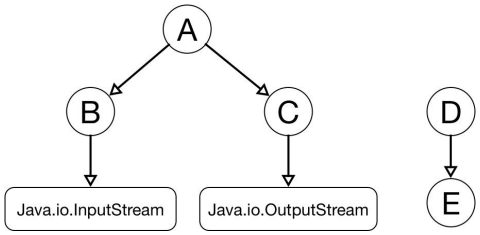


图 3.6　资源类示例图

Algorithm 3.2 构建全局资源类图 C(N,S)

Input: 直接资源类集合 R, 项目中类集合 CP
Output: 全局资源类图 C(N,S)

$N \leftarrow R$
$S \leftarrow \emptyset$
while true **do**
    $temp \leftarrow \emptyset$
    for each $c \in CP$ **do**
        if $(c, c') \notin S \; and \; c' \in N \; and \; c \notin N$ **then**
            $N \leftarrow N \cup \{c\}$
            $S \leftarrow S \cup \{(c, c')\}$
            $temp \leftarrow temp \cup \{c\}$
    if $temp = \emptyset$ **then**
        Break
return $C(N, S)$

为 $\{java.io.InputStream\}$，BR_C 为 $\{java.io.OutputStream\}$，直接资源类的 BR 即为包含其本身的集合。

在构建资源类图时，首先需要定义直接资源类的集合，本文中直接资源类集合为满足直接资源类定义的 Java 源码中和被检测项目中的类的集合。通过这些直接资源类和项目中的类构建全局资源类图的过程见算法3.2。

在算法3.2中，R 为直接资源类的集合，CP 表示项目中的所有类的集合，N 表示直接资源类和间接资源类的集合。循环每次访问 CP 中的类，如果存在某个类 c 中有成员变量是 c' 的对象，并且 c' 是资源类，则 c 也是资源类，(c, c') 即表示 c 与 c' 的包含关系。如果某次循环中，资源类 N 的集合不在增加，则跳出循环，此时的 N 即包含了项目中的所有资源类，S 包含了资源类之间的所有包含关系。

Algorithm 3.3 获取分析域

Input: 全局资源类图 $C(N,S)$，全局方法调用图 $G(M,E)$
Output: 分析域 $A(MA,ME)$,MA 表示分析域中的方法集合，ME 表示入口方法集

```
MA ← ∅
ME ← ∅
for each m ∈ M do
    S_obj ← 方法 m 中操作的所有数据对象的实例集合
    for obj ∈ S_obj do
        if obj instanceof Class X and X ∈ N then
            MA ← MA ∪ m
            break
for each m ∈ MA do
    flag ← 0
    for each m' ∈ M do
        if (m', m) ∈ E and m' ∈ MA then
            flag ← 1
            break;
    if flag = 0 then
        ME ← ME ∪ m
return A(MA, ME)
```

3.3.3 分析域提取

本文分析域被定义为一系列方法的集合，其方法需要满足的条件为：（1）方法中包含的局部变量是资源类的对象；（2）方法是资源类的成员方法。在全局方法调用图 $G(M,E)$ 和全局资源类图 $C(N,S)$ 构建完成后，即可进行分析域的提取。提取分析域的过程见算法3.3。

在算法3.3中，对全局方法调用图中的每一个方法进行分析，如果该方法是某个资源类的成员方法或者方法内部定义的局部变量中包含资源类的对象，则将该方法添加到分析域中。对于分析域中的每一个方法，如果某个方法其 $caller$ 集合中所有方法都不在分析域中，则将该方法添加到入口方法集中，最终筛选出来的方法即为资源泄漏检测的入口方法集合。

考虑如图3.7呈现的某个 Java 项目的类图，该项目中有 5 个类，分别为 XML、$XMLReader$、$XMLWriter$、$XMLParser$ 和 $XMLNode$，$XMLReader$ 负责读取 XML 格式的文件，$XMLWriter$ 负责将以 XML 格式写出数据到文件中，$XMLParser$ 将字符创解析为 $XMLNode$ 格式的数据，$XMLNode$ 为 XML 格式中每个结点的表示形式。XML 中有一个 $deal$ 方法，这个方法相继将成员变量 $xmlReader$、$xmlParser$、

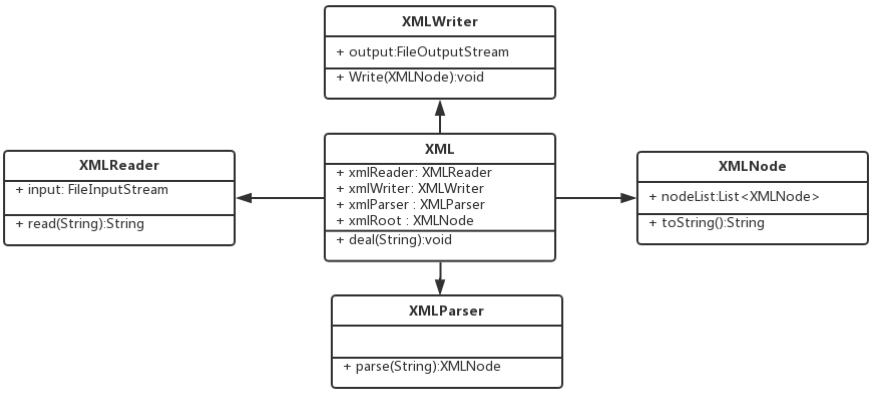


图 3.7　实例类图

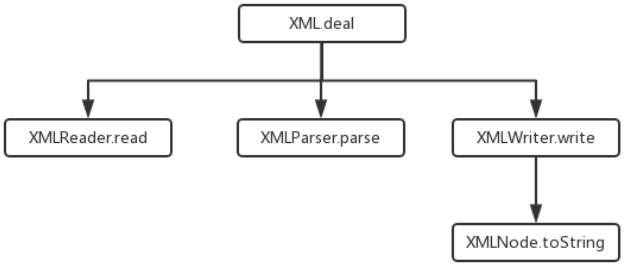


图 3.8　实例方法调用图

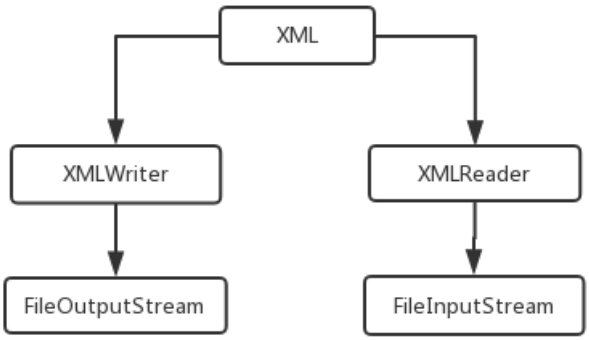


图 3.9　实例资源类图

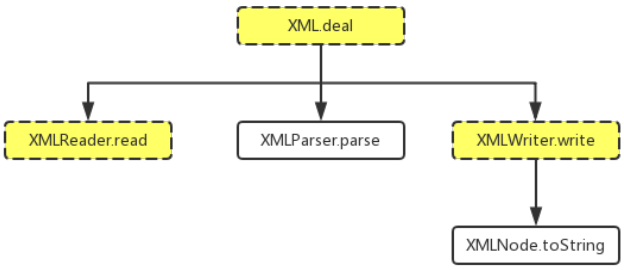


图 3.10　分析域提取之后的方法调用图

$xmlRoot$ 和 $xmlWriter$ 实例化，并依次执行了 $XMLReader.read$、$XMLParser.parse$、$XMLWriter.write$ 方法，所以整个程序的方法调用图如图4.4。

对这样一个程序进行分析域的提取，首先构建资源类图。分析整个程序，可以发现该 Java 程序中涉及的直接资源类包含 $FileInputStream$ 和 $FileOutputStream$，$XMLReader$ 中的成员 $input$ 的类型为 $FileInputStream$，$XMLWriter$ 的成员 $output$ 的类型为 $FileOutputStream$，所以 $XMLReader$ 和 $XMLWriter$ 是间接资源类，XML 中包含类型为 $XMLReader$ 和 $XMLWriter$ 的成员变量，所以 XML 也是资源类。因此，该 Java 程序的资源类图如图3.9。

根据图3.9中的资源类图和图4.4中的全局方法调用图，提取项目的分析域。依次分析方法调用图中的每个方法，如果该方法中操作的类包含资源类，则该方法为资源类方法，添加到分析域中。在该实例中的方法中，$XML.deal$ 操作了 $XMLReader$、$XMLWriter$ 等资源类，$XMLReader.read$ 操作了 $FileInputStream$，$XMLWriter$ 操作了 $FileOutputStream$，其他方法均没有操作资源类，所以最后提取的分析域中的方法见图3.10中虚线边框的方法，即 $XML.read$、$XMLReader.read$ 和 $XMLWriter.write$，并且这三个方法构成一个具有连通性的图，入口方法为 $XML.read$。

3.4　资源泄漏检测过程

本文中设计的跨过程间的资源泄漏检测数据流分析过程是基于 IFDS/IDE 实现的。其主要难点为资源泄漏问题模型的构建和别名分析。对于每一种 IFDS 问题，要基于 IFDS/IDE 架构实现其求解过程，都必须构建在 IFDS/IDE 架构下的问题模型、跨过程间控制流生成器和问题求解器。本文在分析了 Java 程序资源泄漏存在的特点之后，构建了资源泄漏检测问题模型和求解器，跨方法间的控制流生成器基于 Soot 提供的控

制流生成模块和全局方法调用图实现。

3.4.1 资源泄漏问题模型构建

资源泄漏检测问题模型主要定义了在跨过程间的控制流图中四种不同的边（Normal edge、Call-edge、Return-edge 和 Call to Return edge）中的数据流处理逻辑。在沿着控制流图进行数据流分析时，执行对应边中的数据流处理逻辑。资源泄漏检测主要关注的是某个资源在进行资源申请后，沿着控制流图执行其所有可能的路径，如果存在某条可执行路径中资源没有释放，则表明该资源在这条执行路径上发生了资源泄漏。在数据流分析过程中，保存所有被关注的资源变量对应的数据事实在执行每条语句时的状态，最后在求解器中分析，获取发生资源泄漏的资源对象和路径。

3.4.1.1 Normal-Edge 处理过程

$$< x \leftarrow y > (\alpha) = \{x|\alpha = y\} \cup \{\alpha\} \tag{3.1}$$

$$< if(x = null) > (\alpha) = \begin{cases} \emptyset & if\ x = null \wedge x = \alpha \\ \{x\} & if\ x = null \wedge x \neq \alpha \\ \{\alpha\} & if\ x \neq null \end{cases} \tag{3.2}$$

Normal-Edge 表示控制流图中的赋值和判断语句的执行过程，资源泄漏检测问题模型中，针对 Normal-Edge 只处理两种情况，判断表达式和赋值表达式。具体的数据事实传递表达式见公式 3.1 和 3.2，具体分析如下：

(1) 如果当前语句是判断是否为 null 表达式，并且判断语句中一方是被关注的资源对象，如果判断为真，则将该资源对应的数据事实从数据流中去除；如果判断非真，则数据事实集不变；

(2) 如果当前语句是赋值表达式，并且表达式右边为资源对象，则表达式左边的变量添加到数据事实集中；

(3) 其他情况，数据事实集合不变。

3.4.1.2 Call-Edge 处理过程

Call-Edge 考虑的是在数据流分析过程中从主调方法到被调方法的数据流传递，如果一个资源对象从主调方法传递到被调方法，虽然其内存指向是一样的，但是在数据流分析过程中，其对应的数据事实在主调方法和被调方法中的上下文环境是不一样的，所以从主调方法到被调方法的数据流传递主要关心的是数据事实的上下文环境。

$$< x \leftarrow a.fun(p) > (\alpha) = \begin{cases} \{\alpha\} & if\ \alpha = 0 \vee \alpha \in static \\ \{this.\omega\} & if \alpha = a.\omega \\ \{p\} & if\ \alpha = p \\ \emptyset & otherwise \end{cases} \tag{3.3}$$

在执行方法调用时，数据流中的数据事实变化规则见公式 3.3。其具体含义如下：

(1) 如果方法的参数是资源对象，则该资源对应的数据事实传递到被调方法，并转换为被调方法的上下文；

(2) 如果某个静态变量是资源对象，该资源对应的数据事实传递到被调方法；

(3) 如果与被调方法属于同一类的成员变量是资源对象，则对应的数据事实传递到被调方法，并转换为被调方法的上下文；

(4) 其他情况，传递到被调方法的数据事实集为空。

3.4.1.3 Return-Edge 处理过程

Return-Edge 是针对被调方法执行完成后，数据流中的数据事实从被调方法传递到主调方法的处理逻辑，相当于 Call-Edge 的逆向操作，需要考虑包含被调方法返回值、被调方法参数、静态变量以及与被调方法属于同一类对象的成员变量等对应的数据事实。

$$< x \leftarrow a.fun(p) > (\alpha) = \begin{cases} \{\alpha\} & if\ \alpha = 0 \vee \alpha \in static \\ \{\alpha.\omega\} & if\ \alpha = this.\omega \\ \{p\} & if\ \alpha = p \\ \{x\} & if\ \alpha = retVal \\ \emptyset & otherwise \end{cases} \tag{3.4}$$

在执行 Return-Edge 的处理逻辑时，同样需要注意数据事实上下文环境的切换。公式 3.4 描述了从被调方法到主调方法的数据流中数据事实集的传递规则，具体分析如下：

(1) 如果被调方法的返回值是资源对象，则其对应的数据事实传递到主调方法，并将上下文环境转换到主调方法；

(2) 如果被调方法中存在某个静态变量是资源对象，则其对应的数据事实传递到主调方法；

(3) 如果与被调方法属于同一类的某个成员变量是资源对象，则对应数据事实传递到主调方法，并将上下文环境转换到主调方法；

(4) 其他情况，传递到主调方法的数据事实集为空。

3.4.1.4 Call-to-Return-Edge 处理过程

Call-to-Return-Edge 是介于 Call-Edge 和 Return-Edge 之间的一个过程，即如果某个数据流流向了被调方法，则在其返回主调方法之前主调方法中不应该存在这条数据流。遵循这样一种规则，构建的数据事实传递规则如公式 3.5，但是在这一过程中，需要考虑资源的申请操作和释放操作，其中 $Resource.init$ 是一个记录了各种资源初始化操作的集合，$Resource.release$ 是记录了资源释放操作的集合。

$$< x \leftarrow a.fun(p) > (\alpha) = \begin{cases} \emptyset & if\ \alpha \in static \vee \alpha = a.\omega \vee \alpha = p \\ \{x\} & if\ \alpha = 0 \vee fun \in Resource.init \\ \emptyset & if\ \alpha = a \vee fun \in Resource.release \\ \{\alpha\} & otherwise \end{cases} \tag{3.5}$$

根据表达式 3.5，可以发现 Call-to-Return-Edge 的数据事实传递规则如下：

(1) 如果该数据事实被传递到被调方法（静态变量、参数、对应类成员变量等对应的数据事实），则将该数据事实从集合中去除；

(2) 如果被调方法是资源申请方法，则根据表达式左边的变量创建数据事实添加到集合中；

(3) 如果被调方法是资源释放方法，则该资源对应的数据事实从集合中去除；

(4) 其他情况，数据事实集合不变。

3.4.2 别名分析

在 Java 程序中，多个变量可能指向堆中的同一个对象。对指向同一内存对象的多个变量的任何一个进行修改，都可以导致对象的状态的改变，因此，在针对 Java 程序的数据流分析中，别名分析[44] 是一个很重要的部分。如下文代码框中的程序，$in1$ 与 $in2$ 存在别名关系，对任何一个变量执行关闭操作都是直接影响到同一个内存对象中的。在本文的设计方案中，如果在数据流分析中不进行别名分析，也就是在分析过程中将不会构建 $in1$ 与 $in2$ 的别名关系，则当 $in1$ 执行关闭操作后，$in2$ 并没有执行状态的更新，则会导致最后误报 $in2$ 没有关闭，因此在数据流分析过程中，别名分析非常重要。

```
1  InputStream in1 = new FlieInputStream();
2  InputStream in2 = in1;
3  in1.close();
```

别名分析是数据流分析中非常重要的部分，常见的别名分析构建过程有两种：（1）在数据流分析之前通过指向分析构建整个项目的别名分析策略；（2）在数据流分析过程中，需要对某个对象变量进行别名分析时，对分析过的代码进行后向分析，获取该对象的别名集；本文构建别名集的方式是随着数据流分析逐渐构建的，并且考虑了跨过程间的数据流分析时的别名集转换过程，能够保证整个分析过程中别名集的准确性。本文中别名集的构建过程包含两部分：方法内的别名分析和方法间别名传递。

表 3.2　别名分析

Statement	alias
p=q	$gen = \{a \mid q \in in \wedge alias(a,q)\}\ kill = \{a \mid a \in in \wedge alias(a,p)\}$
p = new R	$gen = \{p\}\ kill = \{a \mid a \in in \wedge alias(a,p)\}$
p = null	$gen = \emptyset\ kill = \{a \mid a \in in \wedge alias(a,p)\}$

3.4.2.1　方法内别名分析

别名分析中，定义三元式 $< p, T, A >$，p 表示对象在程序中的表示，T 表示对象的类型，A 表示 p 的别名集合。对于在程序中出现的导致别名集发生变化的语句，如表3.2第一栏，主要包含赋值语句、资源申请和置空语句，针对这三种语句分别提取了其别名集的变化规律，见表3.2第二栏，其中，gen 表示这条语句生成的数据事实集中每个数据事实的别名集需要添加 p 作为别名的数据事实集合，$kill$ 表示这条语句生成的数据事实集中的每个数据事实别名集需要去掉 p 的数据事实集合，in 表示在分析这条语句之前存在的数据事实集。$alias(a,p)$ 表示 a 和 p 是别名关系，$alias(a)$ 表示变量 a 的别名集。

对于 $p = q$ 类型的语句，如何某个数据事实与 q 存在别名关系，则添加 p 作为别名，如果该数据事实之前与 p 存在别名关系，则在其别名集中去掉 p。$p = newR$ 与 $p = null$ 相对简单，在之前与 p 存在别名关系的数据事实的别名集中去掉 p。基于这些变换过程，在方法内进行数据流分析时，逐渐构建被观察数据在每一条语句时的别名集。定义某个资源对象的三元组 $< p, R, A >$，p 为资源对象对应的变量，R 表示资源类型，A 中存放了 p 的别名集，对于每个资源对象，其初始的三元组为 $< p, R_p, \{p\} >$。

表3.3呈现了别名分析随程序的变化过程，第一列表示执行程序的每条语句，第二列表示变量 p 随着程序分析过程中，别名集的变化过程。$p = newR$ 时，p 的别名集中只有 p，随着 $q = p$ 和 $m.r = q$ 的执行，可以发现，q 和 $m.r$ 与 p 指向同一个内存实例，所以具有别名关系，$if(q == null)$ 为 TRUE 时，表示 q、p 和 $m.r$ 均为 null，所以其三元组表达式相当于 $< p, R, \{\} >$，当别名集合为空时，该变量表示资源未申请或已经释放。当 $if(q == null)$ 为 FALSE 时，资源对象表示为 $< p, R, \{q, p, m.r\} >$，继续执行 $release\ p$，释放 p 所指向的资源对象，同时 q 和 $m.r$ 指向的资源对象也被释放，资源对象的表示变为 $< p, R, \{\} >$。

表 3.3　方法内别名分析实例

程序	资源对象表示
p = new R	<p,R,p>
q = p	<p,R,q,p>
m.r = q	<p,R,m.r,q,p>
if(q == null) goto L	T:none F:<p,R,m.r,q,p>
release p	none
L:	none

Algorithm 3.4 方法调用过程中的别名传递算法

Input: 数据事实 fact<v,A,AS>，方法参数集 Arg，被调方法所属类对象 t
Output: 数据事实 fact

1: $newAlias \leftarrow \emptyset$
2: **for** each $v \in fact.A$ **do**
3: **for** each $s \in Arg$ **do**
4: **if** $v = s$ **then**
5: $newAlias \leftarrow newAlias \cup \{v\}$
6: **if** $v = t$ **then**
7: $newAlias \leftarrow newAlias \cup this$
8: $fact.AS.push(fact.alias - newAlias)$
9: $fact.A \leftarrow newAlias$
10: **return** $fact$

3.4.2.2　方法间别名传递

数据流分析过程中，当发生方法调用时，数据事实的上下文需要由主调方法的上下文环境变转换为被调方法的上下文环境；数据事实表示数据流分析过程中被关注数据的状态。同样的，该数据流别名集中的变量也需要由主调方法的变量转换为被调方法的变量。本部分结合方法调用的过程描述本文中在方法间的别名传递过程。

定义 M 表示主调方法的上下文环境，N 表示被调方法的上下文环境。在发生方法调用时，如果某个数据流传入被调方法，则考查该数据流当前的别名集，保存当前的别名集，并提取别名集中能够传入被调方法的变量，将其转换为被调方法中的变量名，然后添加到新的别名集。转换过程见算法3.4。算法中每个数据事实用一个三元组 $< v, A, AS >$ 表示，v 表示数据事实的名称，A 表示该数据事实的别名集合，AS 是一个栈结构，在方法调用过程中，存放上一级方法中该数据事实的别名集。

算法3.4中，如果某个数据事实需要由主调方法上下文切换到被调方法上下文，则访问该数据事实别名集，如果别名集中某个变量是被调方法的参数或被调方法的对象

Algorithm 3.5 方法返回过程中的别名传递算法

Input: 数据事实 fact<v,A,AS>，方法参数集 Arg，被调方法所属类对象 t，方法返回值 r

Output: 数据事实 fact

```
1: newAlias ← fact.AS.pop()
2: for each v ∈ fact.A do
3:     for each s ∈ Arg do
4:         if v = s then
5:             newAlias ← newAlias ∪ {v}
6:     if v = torv = r then
7:         newAlias ← newAlias ∪ v
8: fact.A ← newAlias
9: return fact
```

时，将该变量保存为一个新的别名集，也就是在被调方法中呈现的别名集合，主调方法别名集入栈保存，在被调方法返回时出栈。

当被调方法执行完，执行 return 语句时，所有数据事实的上下文环境需要由被调方法转换为主调方法，根据方法间别名变换规则，数据流的别名集变换如算法3.5。在处理被调方法返回主调方法的过程中，首先提取数据事实中 AS 中保存的主调方法的别名集，并与在被调方法中形成的别名集进行聚合，将方法参数和返回值对应的数据事实返回到主调方法中。

3.5 全局检测设计与实现

全局检测应用于开发人员完成项目开发后进行的全面缺陷评估过程，因此，检测重点是保证全面性、准确性和高效性。本文通过全局方法调用图和全局资源类图提取项目内资源操作层次的所有方法作为分析域，进而进行资源泄漏分析检测。全局方法调用图的构建过程见算法3.1，分析域提取过程见算法3.3，资源泄漏检测算法设计见 3.4 节。分析域是一系列方法簇的集合，每个方法簇中包含若干方法，并且每个方法簇内的方法具有连通性，数据流分析中，需要指定入口方法，即一个方法簇构成的调用关系图中没有被调用边的方法，资源泄漏检测分析过程就是从一个方法簇的多个入口方法开始，逐渐分析方法簇内的所有方法。如图3.11表示了全局检测的整个工作流程，主要分为以下几个步骤：

1. 项目的加载与解析：提取项目中的源码、引用的第三方库和编译后的 class 文件作为系统的输入，应用 Soot 对输入进行加载和解析，转化为 Jimple 格式的中间

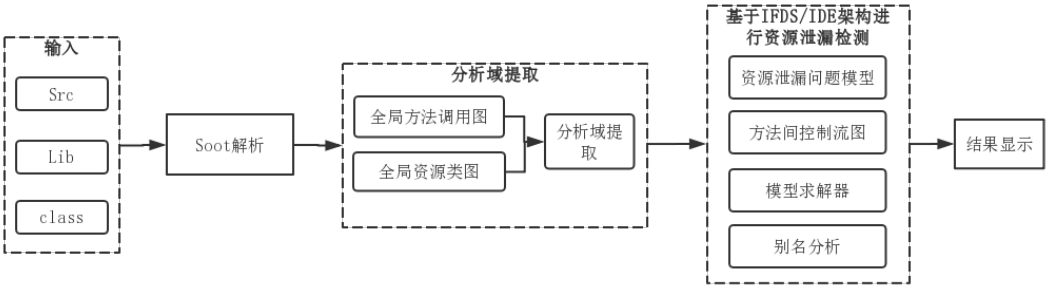


图 3.11　全局检测流程图

表示形式;

2. 分析域提取：结合 Soot 加载项目后形成的控制流图和 Jimple 代码，构建全局方法调用图和全局资源类图，根据资源类遍历方法调用图的所有方法，提取与资源操作有关的方法作为分析域；

3. 资源泄漏检测：基于 IFDS/IDE 数据流分析框架和别名分析算法实现资源泄漏检测算法，将提取分分析域作为资源泄漏检测算法的输入，进行数据流分析；

4. 结果显示：分析数据流分析结果，获取资源泄漏检测的结果并显示。

3.6　本章小结

本章对面向大规模 Java 程序的资源泄漏检测方法架构设计进行详细阐述，并介绍了架构中使用的基础技术 Soot 和 IFDS，然后通过描述全局检测整个过程介绍了全局检测过程中的分析域提取算法和基于 IFDS/IDE 的资源泄漏检测算法的设计与实现；最后介绍了全局检测整个过程的设计与实现。

第 4 章　大规模代码资源泄漏增量检测

Boehm 在论文[45] 中提到过，在软件开发过程中，修正错误的代价随时间几乎是呈指数增长，因此，软件故障发现的越早代价越小。如何针对大规模代码进行精确的即时缺陷检测成为新的研究热点，本部分基于第三部分的全局方法调用图和全局资源类图，设计了一种从当前被修改方法开始，提取当前方法的修改对整个项目数据流影响范围的一种策略，将找到的影响范围作为增量分析过程中针对当前修改需要的分析域，进而进行资源泄漏的检测。本部分的核心思想为：从用户编辑的方法入手，通过缩小检测范围来缩短资源泄漏检测时间。为了实现这一目标，通过资源闭包分析和资源对象指向分析两种方法逐渐缩小检测范围。

4.1　分析准备工作

第三部分设定了全局方法调用图和全局资源类图的概念，本部分引入一个新的概念：直接资源类继承关系图。在 Java 中，继承和多态是非常典型的特点。

```
1  public void foo() throws IOException {
2      FileInputStream in = new FileInputStream(new File("in.txt"));
3      in.read();
4      close(in);
5  }
6  public void close(Closeable io){
7      if(io!=null){
8          try {
9              io.close();
10         } catch (IOException e) {
11             e.printStackTrace();
12         }
13     }
14 }
```

考虑如上代码场景，在 foo 方法中申请了一个 $FileInputStream$ 的 IO 资源 in，调用 $close$ 方法，将 in 作为参数传递，在 $close$ 方法中，参数是 $Closeable$ 类型的，由于 $Closeable$ 是 Java IO 资源较上层的接口，Java IO 中的资源类都实现了这个接口，所以在实际运行中，$close$ 方法中 IO 资源会对应 foo 方法中的 in 实例对象，所以在代

码编辑过程中，对 $close$ 方法中 IO 资源对象的修改是可能影响 foo 方法中 in 实例对象的最终状态的，因此 foo 方法就是修改 $close$ 方法的被影响范围，同理，修改 foo 方法中的 in，也会影响 $close$ 方法中 IO 资源对象的状态。

以 JavaIO 资源为例，在静态分析过程中，要想完整的提取到被修改方法的数据流影响范围，需要构建用于分析直接资源类继承关系的全局直接资源类继承关系图，通过该图能够快速的获取到当前直接资源类的继承关系和被继承关系。定义全局直接资源继承关系图 $I(R,T)$，R 表示项目中直接资源类和涉及到的 JDK 源码中直接资源类的集合，T 表示直接资源类中 extend 或 implement 关系的集合。例如，$FileInputStream$ 继承 $InputStream$ 实现，所以在全局直接资源类继承关系图中存在一条边由 $InputStream$ 指向 $FileInputStream$，$InputStream$ 实现了 $Closeable$ 的接口，故存在由 $Closeable$ 指向 $InputStream$ 的边。

本部分的增量检测分析域的构建即基于全局方法调用图、全局资源类图和全局资源继承关系图来进行分析。为了便于后面分析，构建一个用于分析的实例，其 UML 类图如图4.1，全局直接资源类继承关系如图4.2，全局资源类包含关系如图4.3，全局方法调用如图4.4。该实例中，入口方法 $main$ 分别创建了类 A、B 和 C 的对象，并调用了他们的 $open$ 和 $close$ 方法，类 A 创建了类型为 $XMLReader$ 的成员对象，$XMLReader$ 基于 $InputStream$ 实现；类 B 和 C 创建了类型为 $FileInputStream$ 的成员对象，并且类 A、B 和 C 的 $close$ 方法都是调用 $Util$ 类的 $close$ 方法实现资源关闭。

4.2 资源闭包分析

资源闭包分析是在生成的方法调用图和资源类图基础之上，通过缩小资源泄漏的检测范围，消除传统检测算法中大量冗余计算而造成的性能下降，从而提高检测效率，在本文中，某个类的资源闭包包含两部分：

1. 全局直接资源类继承关系图中，某个直接资源类的闭包包含其继承的所有祖先类和被继承的所有子孙类，其定义了类继承关系上的闭包；

2. 全局资源类图中，能够到达某个直接资源类的的所有资源类为该直接资源类的可操作性闭包。

根据上面的类的资源闭包的定义，在修改某个方法中的对象 obj 时，首先要考虑该对象 obj 能够对哪些直接资源类具有可操作性，然后提取这些直接资源类，从全局

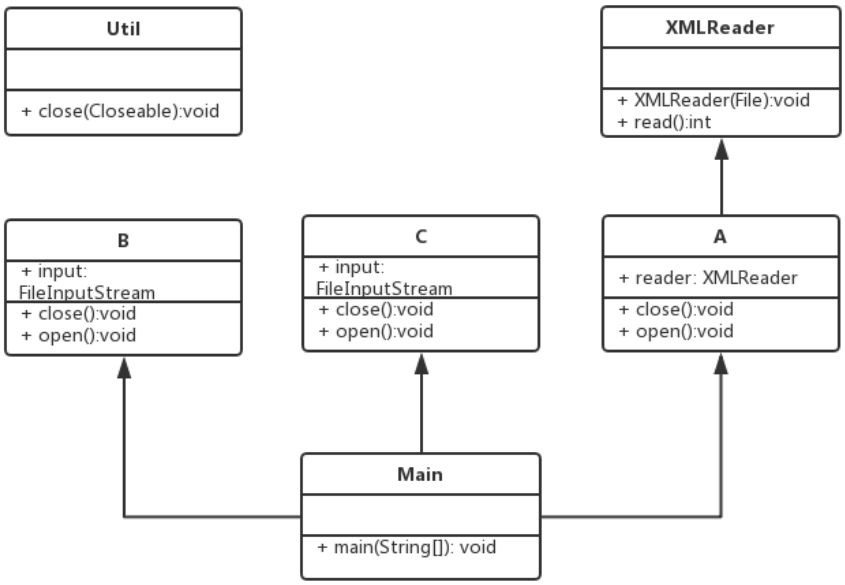


图 4.1　增量分析实例 UML 图

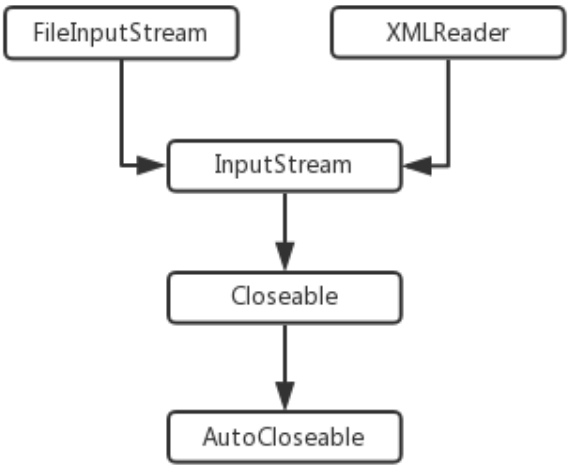


图 4.2　增量分析实例直接资源类继承关系图

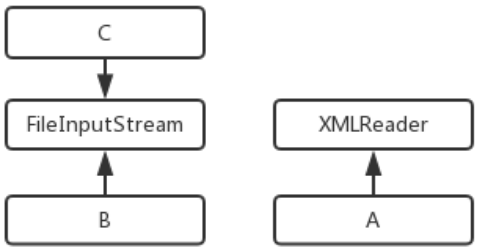


图 4.3　增量分析实例资源类图

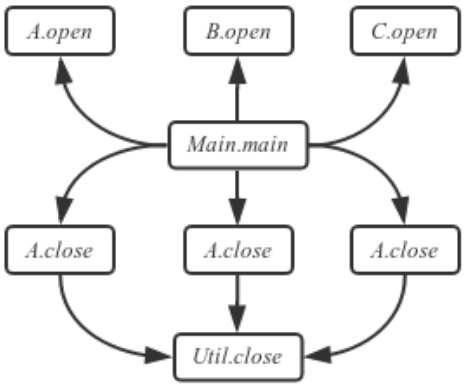


图 4.4　增量分析实例全局方法调用图

直接资源继承关系图中获取这些直接资源类在继承关系上的闭包，再从全局资源类图中，获取这些直接资源类的可操作性闭包。这两个闭包中的资源类的并集即为方法中对象 obj 的资源闭包。某个方法可操作的资源类的闭包即为方法中所有资源类对象的资源闭包的并集，通过方法可操作的资源类的闭包获取修改该方法在全局中的影响范围。

如算法4.1所示，对于类 A，BR_A 表示 A 的实例能够直接或者间接操作的直接资源类的集合，在构建资源类图的时候会计算得到所有资源类直接或间接操作的资源类集合。对于一个给定的方法 m，为了求解修改该方法的影响域 $coverage(m)$，首先需要从该方法操作的所有数据对象集合 S_{obj} 出发，通过 S_{obj} 找出方法 m 可操作的直接资源类集合，根据全局直接资源类继承关系图，获取这些直接资源类在继承结构上的资源闭包 R_m，最后根据方法调用图，从方法 m 开始，进行 m 的资源闭包求解。对于任意的方法 m'，如果 m' 与 m 存在直接的或者间接的调用或被调用关系，且 Rm' 和

Algorithm 4.1 计算方法 m 影响区域 coverage(m)

Input: 方法调用图 G=(M,E), 资源类包含图 C=(N,S), 直接资源类继承关系图 I(R,T), 待分析方法 m

Output: 方法 m 的影响区域 coverage(m)

```
1: coverage(m) ← {m}
2: S_obj ← m 中所有数据对象实例集合
3: R_m ← ∅
4: for each obj ∈ S_obj do
5:     if obj instanceof ClassX then
6:         if X ∈ N then
7:             BR_x ← 图 C 中类 X 可达的直接资源类的集合
8:             R_m ← R_m ∪ BR_x
9:         else if X ∈ R then
10:            R_m ← R_m ∪ {X}
11: M ← coverage(m)
12: while M ≠ ∅ do
13:     Temp ← ∅
14:     for each n ∈ M do
15:         if ((n, m') ∈ E or (m', n) ∈ E) and m' ∉ coverage(m) then
16:             S'_obj ← m' 操作所有数据对象的实例集合
17:             for each obj ∈ S'_obj do
18:                 if obj instanceof ClassX then
19:                     BR_X ← 图 C 中类 X 可达的直接资源类的集合
20:                     if BR_X ∩ R_m ≠ ∅ then
21:                         coverage(m) ← coverage(m) ∪ {m'}
22:                         Temp ← Temp ∪ {m'}
23:     M ← Temp
24: return coverage(m)
```

Rm 有重合，则 m' 就会被加入到 m 的资源闭包 $coverage(m)$ 中。

由于全局方法调用图、资源类图和直接资源继承关系图是一次性事先构建好的，在代码增量修改过程中只需要进行少量同步更新与维护，因此算法4.1的运行时开销非常小。

考虑图4.1中例子，假设当前正在被修改的方法为 $B.open$，根据算法4.1，找到 $B.open$ 中包含的资源类能够直接和间接操作的直接资源类的集合，即 $FileInputStream$，然后在全局直接资源基础关系图中找到其继承关系上的闭包，即 $InputStream$，$Closeable$ 和 $AutoCloseable$，故 $R_{B.open} = \{InputStream, InputStream, Closeable, AutoCloseable\}$；搜索与 $B.open$ 存在调用关系的方法，即 $Main.main$，由于 $Main.main$ 方法中包含类 A、B 和 C 的对象，故 $R_{Main.main} = BR_A \cup BR_B \cup BR_C$，即 $R_{Main.main} =$

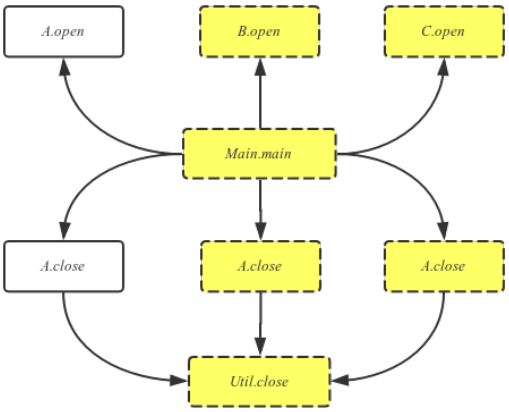


图 4.5　资源闭包分析结果

$\{InputStream, InputStream, Closeable, AutoCloseable, XMLReader\}$，$R_{B.open}$ 和 $R_{Main.main}$ 交集不为空集，所以将 $Main.main$ 加入到 $coverage(B.open)$ 中，然后扫描与 $Main.main$ 存在调用关系的方法，按照算法进行下去，直到 $coverage(B.open)$ 集合不再增加时，修改 $B.open$ 的影响范围即为 $coverage(B.open)$，如图4.5所示即为算法4.1求得的方法 $B.open$ 的影响范围。

4.3　资源对象指向分析

在4.2中通过资源闭包分析得到修改某个方法的影响范围，但是该范围是建立在不同方法操作同一个资源类的基础上的，没有考虑不同方法操作的这个类的对象是否是同一个，例如图4.4中的方法 $B.open$ 和方法 $C.open$，虽然都操作了 $FileInputStream$ 对象，但是不是同一个实例，因此在分析方法 $B.open$ 时，并不需要分析方法 $C.open$。在计算方法 $B.open$ 的影响范围时，可以进一步利用资源变量的指向分析得到操作同一个资源对象的方法集。而在指向分析的过程中，首先以方法为单位获得一个方法中所有资源类对象的指向集合的并集，接着判断方法之间的指向集合的交集是否为空即可排除无关方法，进一步缩小检测范围。

若要得到与方法 $B.open$ 中具有相同资源变量引用对象的所有相关方法，则需要首先提取方法 $B.open$ 中所有的资源变量，并构建其指向集，然后搜索所有与方法 $B.open$ 中资源变量指向集有交集的方法。最后得到如图4.6所示的结果，从而进一步缩小资源泄漏的检测范围，提高检测效率。算法的具体描述如算法4.2所示，其中 $reachingObjects(obj)$ 表示 obj 在内存中的一个指向集。

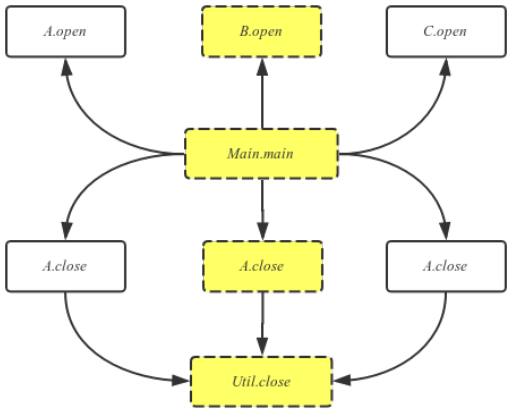


图 4.6　指向分析结果

Algorithm 4.2 使用指向分析进一步过滤 m 的影响域 coverage(m)

Input: 待分析方法 m，通过资源闭包分析得到的影响域 coverage(m)
Output: 过滤后的方法 m 的影响域 coverage(m)

$patA \leftarrow \emptyset$
$S_{obj} \leftarrow m$ 操作的所有的数据对象的变量集合
for each $obj \in S_{obj}$ **do**
    if $obj\ instanceof\ ClassXandX \in N$ **then**
        $pstA \leftarrow pstA \cup reachingObjects(obj)$
for $m' \in coverage(m)$ **do**
    $pstB \leftarrow \emptyset$
    $S_{obj} \leftarrow m'$ 操作的所有数据对象的变量集合
    for each $obj \in S_{obj}$ **do**
        if $obj\ instanceof\ ClassXandX \in N$ **then**
            $pstB \leftarrow pstB \cup reachingObjects(obj)$
    if $pstA \cap pstB = \emptyset$ **then**
        $coverage(m) \leftarrow coverage(m) - \{m'\}$
return $coverage(m)$

通过指向分析进一步过滤被修改方法影响域见算法4.2,其中方法 $reachingObjects(obj)$ 可以返回 obj 对象的指向集。假设 m 为用户修改的方法，算法中首先计算方法 m 可能操作的资源类对象的指向集合，即 $ptsA$；其次一次判断 $coverage(m)$ 中的每个方法 m'，计算 m' 可能操作的所有资源类对象指向集合即 $ptsB$；通过这两个指向集合判断 m 是否有可能与 m' 操作同一个资源类对象的实例，也就是 $ptsA$ 与 $ptsB$ 是否有交集，如果没有交集，则证明方法 m 中操作的任何资源类对象的实例不可能被方法 m' 中的对象访问，因此 m' 不可能在方法 m 的影响域中，故从 $coverage(m)$ 除掉 m' 方法。该算法对方法操作的资源对象指向集合两两求交集，而非所有方法资源对象集合一起求

交集，因而可能会将一些非实际相关的方法加入到检测集合中来，但是由于在检测阶段会进行精确的数据流分析，因此并不影响最终的检测结果。

根据算法4.2, 对于增量分析实例，如图4.1, 在经过资源闭包分析后得到的分析域如图4.5, 算法4.2中，首先提取方法 $B.open$ 中所有资源类对象的指向集，将其并集作为方法 $B.open$ 操作的资源类对象实例的集合。搜索资源闭包分析得到的影响域 $coverage(B.open)$，即集合 $\{Main.main, B.open, B, close, C.open, C.close, Util.close\}$，依次获取该方法集中每个方法的指向集，如果存在某个方法的指向集与 $B.open$ 的指向集没有交集，则将该方法从 $coverage(B.open)$ 中移除，根据实际情况，$C.open$ 和 $C.close$ 中的资源对象实例与方法 $B.open$ 中的资源对象实例并不是同一个，所以其指向集的交集为空，故需要从 $coverage(B.open)$ 移除方法 $C.open$ 和 $C.close$。过滤后的方法 $B.open$ 的影响域 $coverage(B.open) = \{Main.main, B.open, B, close, Util.close\}$，即图4.6中虚线表示的方法结点组成的区域。

4.4 增量检测设计与实现

增量检测设计的目的就是能够在开发人员编辑代码过程中快速的返回当前修改是否造成新的资源泄漏发生，因此本文最终的增量检测方案结合算法4.1和算法4.2。对于一个大型的 Java 程序，要对其进行全面的指向分析的构建是一个非常消耗时间和内存的操作，因此本文首先通过算法4.1较为粗略的获取被修改方法的影响域，但是可以过滤掉项目中大部分无关方法，如果当前获取到的影响域小于设定的阈值 K，则不会进行后续的指向分析进一步过滤结果，直接进行资源泄漏检测分析，反之需要将前一步得到的影响域进行指向集的构建，通过算法4.2进一步过滤被修改方法的影响域。

对于通过算法4.1和算法4.2得到的被修改方法的影响域，不难分析其肯定是全局方法调用图中一个具有连通性的子图，要通过资源检测算法进行分析，必须找出影响域的入口方法，从入口方法开始进行分析。一个方法域的入口方法定义为一个在影响域中只有调用边而没有被调用边的方法。一个影响域可能包含一个或多个入口方法，资源泄漏的检测分析过程必须从多个入口方法开始，利用 IFDS/IDE 构建的资源泄漏检测算法逐一进行检测。

如图4.6中的得到的方法 $B.open$ 的影响域，只存在一个入口，即方法 $Main.main$，因此在资源泄漏检测时，只需要从 $Main.main$ 开始进行检测分析，沿着方法调用图对影响域中的方法进行数据流分析，对于不在影响域中的方法直接跳过即可。

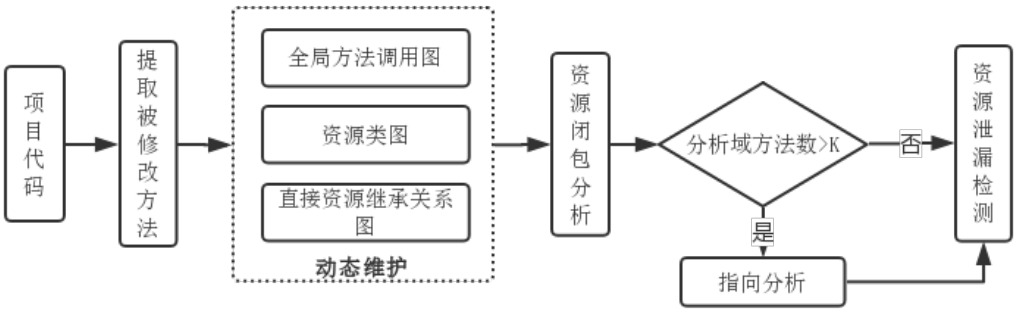


图 4.7　增量检测过程

本部分中提出的资源泄漏增量检测算法，通过资源闭包分析和指向分析等手段将资源泄漏的全局检测范围逐步缩小到时间上可承受的检测范围；在通过资源闭包进行分析的过程中，对未引用资源类的方法进行“剪枝”，从而避免非资源相关方法的冗余分析；另外，在进行资源变量的指向分析，进一步构建出范围缩小的方法域，从而提高资源泄漏检测的效率和准确性。增量检测算法的具体过程如图4.7，主要分为以下几个步骤：

1. 动态维护全局方法调用图、资源类图、直接直接继承关系图：获取当前项目中被修改的所有方法集合和相关类集合，根据相关方法和类的增加、删除、修改操作，动态修改方法调用图和资源类图;

2. 资源闭包分析获取被修改方法影响域：对于所有被增量修改的方法，判断是否包含资源类变量，过滤出所有包含资源变量的方法集合；从所有操作资源变量的方法向外扩展，找出能够操作相同直接资源类的方法，形成与被修改方法中包含相同直接资源类的方法域作为被修改方法的影响域;

3. 指向分析：若被修改方法的影响域中包含的方法数目大于阈值 k，通过指向分析策略在影响域内找出操作相同资源对象实例的所有方法，生成更小的影响域；

4. 资源泄漏检测：从影响域构成的方法调用图的入口方法开始，利用过程间的资源泄漏检测算法进行检测与分析。

4.5 本章小结

本章描述了面向大规模 Java 程序的资源泄漏检测方法增量检测的整个过程。基于全局方法调用图、资源类图和直接资源继承关系图，通过资源闭包分析、指向分析等手段提取出被修改方法在资源操作层次上影响域，从而构建了从单一方法快速获取分析域的策略，加速了资源泄漏检测过程。

第 5 章　实验评估

5.1　实验设置

在本次评估中，使用的测试环境为 Windows 7 64 位系统，4 核 CPU 以及 8GB 内存。Eclipse 版本为 4.5.2，使用的 JDK 版本为 1.8。

本文中测试用例一共分为两类，大规模 Java 测试程序集和小规模 Java 测试集。其中大规模 Java 测试集见表5.1，测试集包含 Tomcat、Weka、Freeplane 和 JexcelAPI 等。Tomcat 是 Apache 软件基金会的一个程序，其开源性、性能稳定并且免费的特点广受各位 Java 爱好者的喜爱，是目前较为流行的 Web 应用服务器。Weka 是一款免费的、开源的、非商业化，基于 Java 环境下开源的机器学习以及数据挖掘软件。Freeplane 是一款 Java 开发的类似于思维导图的软件。JexcelAPI 是基于 Java 开发的操作 Excel 的接口，具有良好的扩展性。小型测试集由 DroidLeaks[46] 和 Defects-bench 两部分组成，DroidLeaks 是相关研究人员从 34 个现实生活中开源 Android App 中抽取出来的真正存在的资源泄漏测试集，由于 DroidLeaks 着重于 Android 相关的测试用例，本文只抽取了测试集中与 Java 相关的测试用例，但是从 DroidLeaks 抽取出来的测试用例对 Java 各种语言特性的测试覆盖不全面，所以测试不够全面，因此本文设计了更全面的的资源泄漏测试集 Defects-bench，从 Java 语言的各个方面考虑，包含多态、反射、内部类、静态成员、集合、别名、try-with-resource 和方法间的数据传递等各种特性，用以保证检测测试全面性。在小型测测试集中，DroidLeaks 中共有 16 个测试用例，其中包含资源泄漏缺陷的测试用例 15 个，没有资源泄漏的测试用例 1 个；Defects-bench 中共有 73 个测试用例，其中包含资源泄漏缺陷的测试用 53 个，不存在资源泄漏的测试用例 18 个。

本文中为了验证本方案资源泄漏检测的有效性，与 FindBugs 和 FortiFy 等知名缺

表 5.1　大规模 Java 测试程序详细说明

测试用例名	Java 文件个数	代码行数（万行）
Tomcat-8	1591	31
Weka	1568	50
Freeplane	876	12
Jexcelapi	458	9

表 5.2　小型 Java 测试集详细说明

测试用例名	Java 文件个数	代码行数（万行）
DroidLeaks	16	958
Defects-bench	71	2132

表 5.3　DroidLeaks 检测结果

方法	正确率 (%)	漏报率 (%)
FindBugs	6.25	93.75
Fortify	87.50	12.50
本文方法	93.75	6.25

陷检测工具进行对比。FindBugs 是一个静态分析工具，它通过一组缺陷模式对比分析来发现可能的问题，其提供对 Eclipse 的无缝插件集成，可以即时查找代码存在的缺陷。Fortify SCA 是一个静态的、白盒的软件源代码安全测试工具。它通过内置的五大主要分析引擎：数据流、语义、结构、控制流、配置流等对应用软件的源代码进行静态的分析。Fortify 支持多达 21 种编程语言代码的检测。本文分别使用这两种工具对测试用例进行检测，并与本文提出的方法得到的检测结果进行对比。其中 FindBugs 的版本为 3.0.1，Fortify 的版本为 5.1。其中 Fortify 由于版本原因，检测时最高的 JDK 版本为 1.6。

5.2　全局检测对比分析

本文针对小型测试集分别使用 FindBugs、Fortify 及本文方法进行分析检测，对每一个测试结果进行人工分析，最后统计结果的正确率、漏报率和误报率，正确率是指测试用例的确存在着资源泄漏问题而工具也检测出来该问题（True Positive）和测试用例没有资源泄漏的问题工具也没有检测出来问题（True Negative）个数的和，漏报是指测试用例存在资源问题但是检测工具没有并没有检测出来该问题（False Negative），误报是指测试用例不存在资源泄漏问题但是方法检测出来存在问题（False Positive）。DroidLeaks 的检测结果见表5.3，Defects-bench 的检测结果见表5.4。从表中可以看出在两个测试用例集中本文方法表现出比 FindBugs 和 Fortify 更高的正确率，较低的误报及漏报率，这是由于本文应用了过程间路径敏感的数据流分析和别名分析。FindBugs 漏报率较高的原因是其不支持方法间的资源泄漏检测。

表 5.4　Defects-bench 检测结果

方法	正确率 (%)	漏报率 (%)	误报率 (%)
FindBugs	38.03	84.65	1.41
Fortify	59.15	39.44	0.00
本文方法	61.97	35.21	1.41

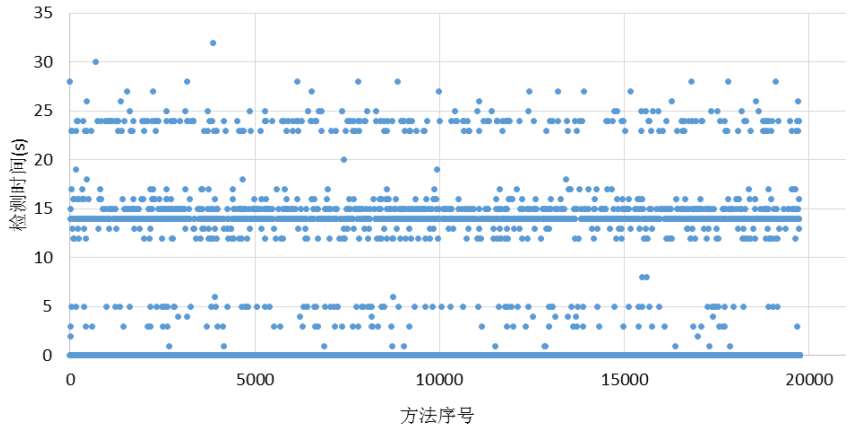


图 5.1　Tomcat 中方法增量检测时间分布

5.3　增量检测实验分析

在进行增量检测时，本文评估的方式是统计增量检测中对缺陷的召回率以及增量检测的时间。本文对召回率的定义是：针对每个全局检测中检测出来的缺陷，假设每个方法为当前编辑方法，从每个方法开始进行增量检测，看是否能够再次检测到这些缺陷。在评估过程中，对大规模的 Java 测试集进行增量检测时间和缺陷召回率评估，对小规模的测试集只进行召回率的评估。在本实验中，设置使用指向分析的阈值 K 为 200，即在增量检测中，通过资源闭包分析得到的分析域中方法数超过 200 时，需要进行资源对象指向分析来进一步减小分析域。

图5.1、图5.2、图5.3和图5.4分别表示 Tomcat、Weka、Freeplane 和 JexcelAPI 在增量检测中每个方法完成检测的时间散点图分布。横坐标表示入口方法序号，纵坐标则表示从该方法开始进行增量检测完成检测的时间。图5.5分别表示 Tomcat、Weka、Freeplane 和 JexcelAPI 在增量检测中每个方法完成检测的时间分布。

根据图5.1和图5.2，在 Tomcat 近 2 万个方法中，绝大多数的方法增量检测都是在 30s 内完成，只有极个别的方法执行时间超出 30s，但也都控制在 35s 以内。而由于 Weka 比 Tomcat 超出了接近 20 万行代码，导致 Weka 的部分方法检测时间超过了

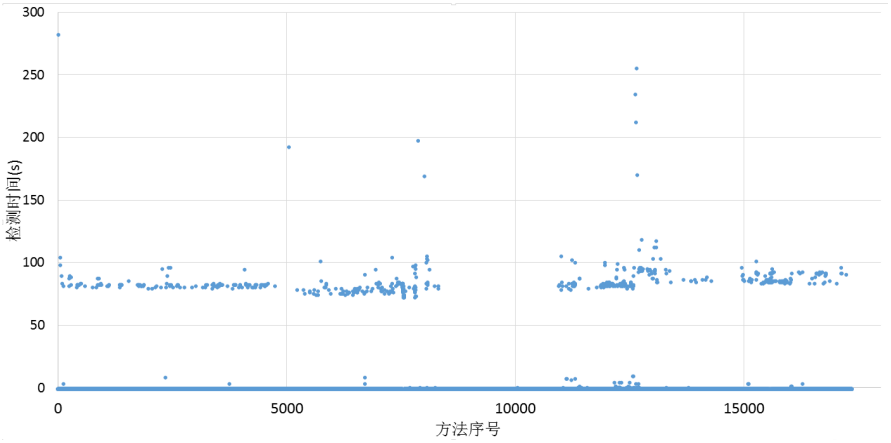


图 5.2　Weka 中方法增量检测时间分布

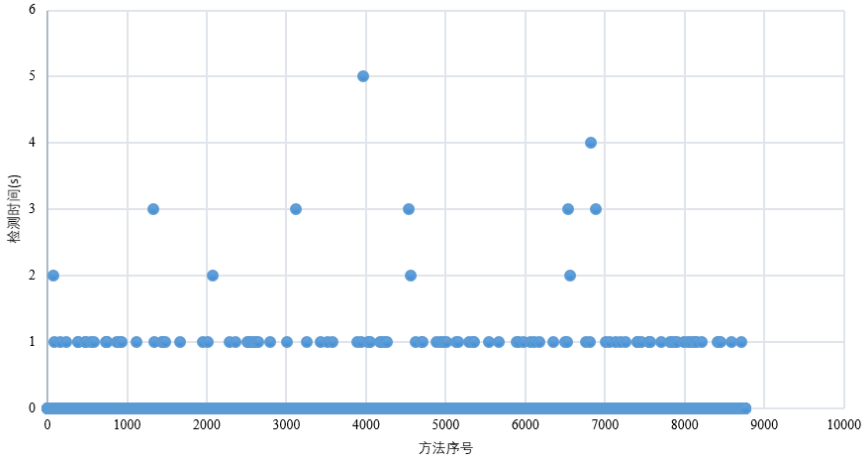


图 5.3　Freeplane 中方法增量检测时间分布

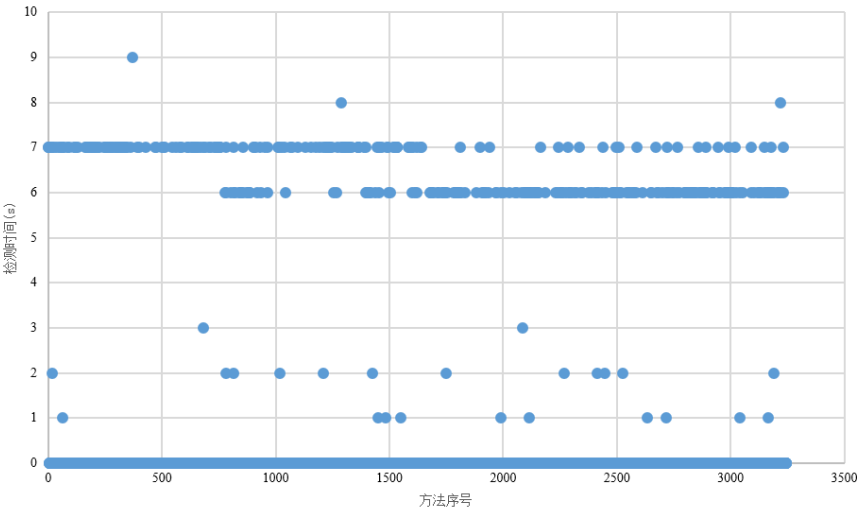
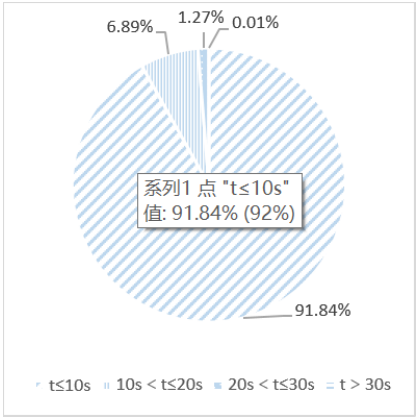
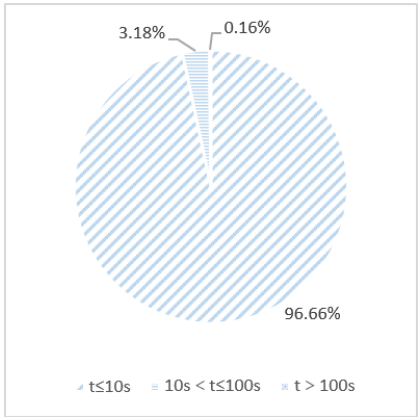


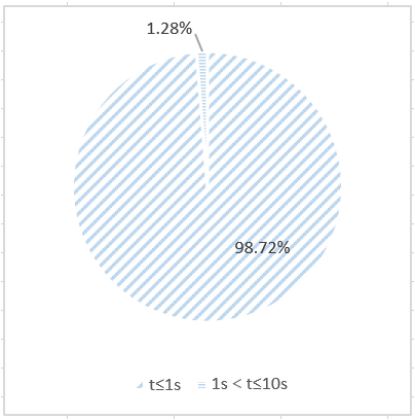
图 5.4　JexcelAPI 中方法增量检测时间分布



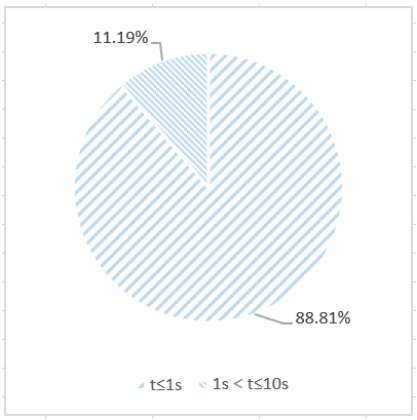
(a) Tomcat 增量检测时间分布



(b) weka 增量检测时间分布



(c) Freeplane 增量检测时间分布



(d) JexcelAPI 增量检测时间分布

图 5.5　方法增量检测时间占比分布

表 5.5　不同测试用例集在增量检测中的召回率

测试用例名	召回率 (%)
DroidLeaks	100.00
Defects-bench	100.00
Tomcat	98.26
Weka	87.87
Freeplane	100.00
JexcelAPI	100.00

100s，但是从图5.5的时间分布图来看，Tomcat 和 Weka 检测时间小于 10s 的方法占比仍达到 90% 以上，因此，大多数情况下仍可以达到即时检测和报告的效果。

表5.5给出了增量检测相对于全局检测的召回率情况。不难发现，对于小规模测试程序，本文提出增量检测方法，能够达到 100% 的召回率；即使对于 Weka，仍然能够获得 87% 以上的召回率，也就是说，对于 Weka，全局检测中超过 87% 的资源泄漏

可以在增量检测中被即时检测到。召回率降低主要是由于方法调用图在增量代码分析的时构建不完整造成的。在 Java 项目中，由于多态的特性，从某一个方法出发，并不能确定某些对象实际对应的类，所以也不能确定其对应的成员方法，在进行增量分析时，构建的方法调用图不完整，故召回率达不到 100%。因此，实际应用中可以采用增量和批量相结合的方式，既解决了检测时间的问题，也保证了较高的准确率。

5.4 本章小结

本章对面向大规模 Java 程序的资源泄漏检测方案进行实验评估。通过 DroidLeaks 和 Defects-bench 对资源泄漏检测算法的正确率进行评估;通过 Tomcat、Weka、Freeplane 和 JexcelAPI 等大型项目对增量检测的错误召回率和检测时间进行评估；根据评估结果可知，相比于 FindBugs 和 Fortify 等典型缺陷检测工具，本文中资源泄漏检测方案具有更高的准确率和较低的误报率，在增量检测中，在保证检测准确率的前提上，90%的增量检测实验可以在 10s 内完成。

结论

随着大数据、云计算、人工智能等高新技术的不断发展，软件系统的处理逻辑和架构越来越复杂，导致软件缺陷越来越不容易被发现，资源泄漏作为一种危险系数高、出现次数多的软件缺陷，极大的影响了软件的安全性和可靠性。因此本课题面向大规模 Java 程序提出了一种高效的全局和增量式的静态资源泄漏检测方案，该方法支持过程间流敏感、路径敏感的资源泄漏检测；在进行全局资源泄漏检测时，构建全局方法调用图和全局资源类图，基于资源类进行资源方法的过滤和提取，进而缩小资源泄漏检测范围；在增量检测中，以开发人员修改的方法为入口点，通过资源闭包分析、指向分析等策略逐步缩小待分析资源相关方法的范围，从而避免非资源相关方法的冗余分析，提高资源泄漏检测的效率和准确性。通过实验评估，相比于 FindBugs 和 Fortify 等典型缺陷检测工具，本课题中资源泄漏检测方案具有更高的准确率和较低的误报率，在增量检测中，在保证检测准确率的前提上，90% 的增量检测实验可以在 10s 内完成，能够满足在开发人员编辑代码过程中对缺陷进行即时检测的实际应用需求。

围绕大规模 Java 程序的资源泄漏检测方案，本课题主要做了以下工作：

(1) 分析 Java 程序中资源泄漏的问题模型，构建基于 IFDS/IDE 过程间数据流分析框架的跨过程的、流敏感的、路径敏感的资源泄漏检测方案，并应用别名分析算法，提升资源泄漏检测的准确率；

(2) 全局检测方案中，针对程序中资源操作存在的特点，构建全局方法调用图和资源类图，通过提取资源类方法作为资源泄漏检测分析域减少资源泄漏检测冗余分析；

(3) 增量检测方案中，从被修改方法开始，依据资源闭包分析、指向分析的方法提取当前修改的方法在资源操作层次上的影响范围，构建资源泄漏检测分析域，减低资源泄漏检测时间消耗。

本课题研究的创新性如下：

(1) 基于 IFDS/IDE 数据流分析架构和别名分析算法构建资源泄漏检测方案，具有跨过程、流敏感、路径敏感等特性；

(2) 基于指向分析、资源闭包分析等方法，提取开发人员当前修改的方法在资源操作层次上的影响范围作为资源泄漏检测分析域，提高检测效率。

本课题中提出的面向大规模 Java 程序的资源泄漏检测方案依然存在一些不足，例如在大规模项目的增量检测过程中，内存消耗较大、资源泄漏检测算法对静态变量的检测支持不足等，这也是本课题继续研究的方向。

参考文献

[1] 王克朝, 王甜甜, 苏小红, et al. 软件错误自动定位关键科学问题及研究进展 [J]. 计算机学报, 2015, 38 (11): 2262–2278.

[2] Maraia V. The Build Master: Microsoft's Software Configuration Management Best Practices [M]. Addison-Wesley Professional, 2005.

[3] Bao Y, Sun X, Trivedi K S. Adaptive software rejuvenation: Degradation model and rejuvenation scheme [C]. In Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on, 2003: 241–248.

[4] Bao Y, Sun X, Trivedi K S. A workload-based analysis of software aging, and rejuvenation [J]. IEEE Transactions on Reliability, 2005, 54 (3): 541–548.

[5] Guo P J, Engler D R. Linux Kernel Developer Responses to Static Analysis Bug Reports. [C]. In USENIX Annual Technical Conference, 2009: 285–292.

[6] Lifan Z X L. Application and Research on Static Test Technology Based on Klocwork Insight [J] [J]. Computer & Telecommunication, 2011, 11: 033.

[7] Evans D. Static detection of dynamic memory errors [C]. In ACM SIGPLAN Notices, 1996: 44–53.

[8] 金大海, 宫云战, 王雅文, et al. 软件代码测试技术 [J]. 信息通信技术, 2015, 3: 007.

[9] Godefroid P, Klarlund N, Sen K. DART: directed automated random testing [C]. In ACM Sigplan Notices, 2005: 213–223.

[10] Cadar C, Dunbar D, Engler D R, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. [C]. In OSDI, 2008: 209–224.

[11] Sen K, Marinov D, Agha G. CUTE: a concolic unit testing engine for C [C]. In ACM SIGSOFT Software Engineering Notes, 2005: 263–272.

[12] 唐容. 支持非数值型测试用例自动生成的抽象内存建模技术研究 [D]. [S. l.]: 北京邮电大学, 2013.

[13] Gbajabiamila J, Kejriwal A, Patodia Y. An Evaluation of Rational PurifyPlus [J], 2008.

[14] Bodden E. Inter-procedural data-flow analysis with ifds/ide and soot [C]. In Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, 2012: 3–8.

[15] Torlak E, Chandra S. Effective interprocedural resource leak detection [C]. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, 2010: 535–544.

[16] 肖庆, 宫云战, 杨朝红, et al. 一种路径敏感的静态缺陷检测方法 [J]. 软件学报, 2010, 21 (2):

209–217.

[17] Engler D, Dunbar D. Under-constrained execution: making automatic code destruction easy and scalable [C]. In Proceedings of the 2007 international symposium on Software testing and analysis, 2007: 1–4.

[18] Bruttomesso R, Cimatti A, Franzén A, et al. The mathsat 4 smt solver [C]. In CAV, 2008: 299–303.

[19] Brummayer R, Biere A. Boolector: An efficient SMT solver for bit-vectors and arrays [J]. Tools and Algorithms for the Construction and Analysis of Systems, 2009: 174–177.

[20] Moy Y, Bjørner N, Sielaff D. Modular bug-finding for integer overflows in the large: Sound, efficient, bit-precise static analysis [J]. Microsoft Research, 2009, 11.

[21] Godefroid P, Levin M Y, Molnar D A, et al. Automated whitebox fuzz testing. [C]. In NDSS, 2008: 151–166.

[22] Cadar C, Ganesh V, Pawlowski P M, et al. EXE: automatically generating inputs of death [J]. ACM Transactions on Information and System Security (TISSEC), 2008, 12 (2): 10.

[23] Molnar D, Li X C, Wagner D. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. [C]. In USENIX Security Symposium, 2009: 67–82.

[24] Ganesh V, Dill D L. Decision procedures for bit-vectors, arrays and integers [M]. 2007.

[25] Brummayer R, Biere A, Lonsing F. BTOR: bit-precise modelling of word-level problems for model checking [C]. In Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning, 2008: 33–38.

[26] Bruttomesso R, Cimatti A, Franzén A, et al. A lazy and layered SMT (BV) solver for hard industrial verification problems [C]. In CAV, 2007: 547–560.

[27] De Moura L, Bjørner N. Z3: An efficient SMT solver [J]. Tools and Algorithms for the Construction and Analysis of Systems, 2008: 337–340.

[28] Dutertre B, De Moura L. The yices smt solver [J]. Tool paper at http://yices. csl. sri. com/tool-paper. pdf, 2006, 2 (2): 1–2.

[29] Dutertre B, De Moura L. A fast linear-arithmetic solver for DPLL (T) [C]. In Computer Aided Verification, 2006: 81–94.

[30] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation [C]. In Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, 2004: 75.

[31] Bhansali S, Chen W-K, De Jong S, et al. Framework for instruction-level tracing and analysis of program executions [C]. In Proceedings of the 2nd international conference on Virtual execution

environments, 2006: 154–163.

[32] Do L N Q, Ali K, Livshits B, et al. Just-in-time static analysis [C]. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2017: 307–317.

[33] Ayewah N, Hovemeyer D, Morgenthaler J D, et al. Using static analysis to find bugs [J]. IEEE software, 2008, 25 (5).

[34] Chess B, McGraw G. Static analysis for security [J]. IEEE Security & Privacy, 2004, 2 (6): 76–79.

[35] Bessey A, Block K, Chelf B, et al. A few billion lines of code later: using static analysis to find bugs in the real world [J]. Communications of the ACM, 2010, 53 (2): 66–75.

[36] Ayewah N, Pugh W. A report on a survey and study of static analysis users [C]. In Proceedings of the 2008 workshop on Defects in large software systems, 2008: 1–5.

[37] D’ Souza D, Kim Y P, Kral T, et al. Tool evaluation report: Fortify. 2014.

[38] Surana P. Meta-compilation of language abstractions [J], 2006.

[39] Lam P, Bodden E, Lhoták O, et al. Soot-a Java bytecode optimization framework [C]. In Cetus Users and Compiler Infrastructure Workshop, 2011: 1–11.

[40] 梅宏, 王千祥, 张路, et al. 软件分析技术进展 [J]. 计算机学报, 2009, 32 (9): 1697–1710.

[41] Vallee-Rai R, Hendren L J. Jimple: Simplifying Java bytecode for analyses and transformations [J], 1998.

[42] Umanee N. Shimple: And Investigation of Static Single Assignment Form [D]. [S. l.]: McGill University, 2005.

[43] Østvold B M, Kristoffersen T. Analysis of object-oriented programs: a survey [J], 2005.

[44] 刘英, 刘磊. 面向 Java 的实用别名分析技术 [J]. 计算机研究与发展, 2000, 37 (5): 595–600.

[45] Broy M, Denert E. Pioneers and Their Contributions to Software Engineering: Sd & M Conference on Software Pioneers, Bonn, June 28/29, 2001, Original Historic Contributions [M]. Springer Berlin Heidelberg, 2001.

[46] Liu Y, Wei L, Xu C, et al. DroidLeaks: Benchmarking Resource Leak Bugs for Android Applications [J]. arXiv preprint arXiv:1611.08079, 2016.

致谢

研究生学习生涯即将结束，在这两年时光里，作为一名学生，无论是学术上还是生活上，都学到了很多。在此即将毕业之际，我衷心的感谢这两年给我带来学习和生活指导的导致计卫星老师，感谢计老师两年来对我的帮助。计老师为人谦和，治学态度严谨，在科研方面从严要求，以身作则，对学生负责，不论是在科研还是做人方面，计老师都为我们树立了榜样，从计老师身上学到的东西都将我受益终身。

同时也要感谢实验室课题组的石峰老师、高玉金老师、王一拙老师和魏晋老师，感谢他们在我学习和科研中提供的帮助；感谢实验室的石剑君大师姐、付文飞师姐在我遇到困难时给与我的指导；感谢廖心怡同学、张露露同学、张晶晶同学和罗辉同学，在我毕设完成过程中提供的帮助；感谢田泽明是滴、高建花师妹、李安民师弟、谈兆年师弟、杨恬师弟，感谢你们给实验室带来的欢乐，让实验室更温暖，感谢你们两年的陪伴。

特别感谢我的父母和女朋友，在我生活中遇到困难时给与极大的鼓励和帮助，无论怎样，你们一直坚定不移的鼓励支持我，在此，非常感谢你们对我的付出。

最后，谨向百忙之中参加我论文评审和答辩的老师致以崇高的敬意个衷心的感谢。

攻读学位期间发表论文与研究成果清单

[1] 第一作者. 大规模代码增量式资源泄漏检测方法研究 [J]. 软件学报,（核心期刊）

[2] 第一作者. Rapidly Generate and Visualize the Digest of Massive Time Series Data[C]. IEEE Third International Conference on Big Data Computing Service and Applications. IEEE, 2017:157-164,(EI)