基于深度学习的空指针引用缺陷检测系统的设计与实现

罗辉

2018 年 5 月

中图分类号：TQ028.1

UDC分类号：540

基于深度学习的空指针引用缺陷检测系统的设计与实现

作 者 姓 名　罗辉

学 院 名 称　软件学院

指 导 教 师　田东海教授

答辩委员会主席　计卫星教授

申 请 学 位　工学硕士

学 科 专 业　软件工程

学位授予单位　北京理工大学

论文答辩日期　2018 年 5 月

Design and implementation of null pointer reference defect detection system based on deep learning

Candidate Name:	Hui Luo
School or Department:	Software Institute
Faculty Mentor:	Prof. Donghai Tian
Chair, Thesis Committee:	Prof. **
Degree Applied:	Master of Science
Major:	Software engineering
Degree by:	Beijing Insititute of Technology
The Date of Defence:	June，2018

基于深度学习的空指针引用缺陷检测系统的设计与实现

北京理工大学

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

空指针引用是程序中比较常见的缺陷之一，研究表明该缺陷在编译后的程序中大量存在，因此给软件的稳定性带来很大威胁。出于对检测效率和精度的平衡，现有工具在工作原理和检测范围等方面各不相同，无法全面检测该类缺陷，大量的误报也降低了此类工具的实用价值。本课题基于 Soot 框架实现一种 Java 代码空指针引用缺陷检测工具，弥补了同类工具在某些方面的不足。此外还提出一种多个工具交叉验证的空指针引用缺陷检测方法，并在 SonarQube 平台上进行实践。

关键词： 空指针引用；静态检测；交叉验证；

Abstract

In order to exploit ⋯⋯.

Key Words: shape memory properties; polyurethane;textile;synthesis;application

目录

第 1 章　绪论

1.1　研究背景

随着互联网的迅猛发展，各种应用软件日益增多，软件规模越来越大，但是这些软件背后的软件安全及其稳定性的问题也日益突出。特别是当下人工智能技术的快速成熟，软件的智能化和自动化程度都上升到了新的高度，在越来越多的无人场景下，如无人驾驶，智慧医疗等领域，软件正逐步完全替代人类在一些重要领域的工作，这就对软件的安全性 [1] 提出了更高的要求。

任何的软件设计或者编码产生的安全漏洞 [2]，都可能成为潜在的安全隐患，可能会给社会带来巨大的损失，信息安全问题早就成为人们日益关注的焦点 [3]。近年来，重大网络安全事件层出不穷，如 2017 年 5 月，史上规模最大的一次勒索病毒攻击事件爆发，全球近百个国家的网络遭遇 Wannacry 病毒 [4] 的攻击，电脑被该病毒感染后文件会被加密锁定，支付黑客索要的赎金后才能解密恢复，受攻击对象甚至包括医院、高校等公益性机构。2013 年 6 月，前美国中情局 (CIA) 雇员斯诺登曝出一项由美国国家安全局 (NSA) 实现的棱镜计划 [5]，震惊全球。据斯诺登披露的资料显示，NSA 通过植入恶意软件感染了全球超过 5 万台计算机，用于窃取敏感信息；

由于软件的复杂性随着软件的规模和数量不断增大，软件开发的规模和数量不断增大，软件开发的难度也在增大，导致在开发过程中存在某些不确定性的错误或缺陷。另一方面软件开发人员的水平参差不齐，即使是富有经验的开发者在开发过程中也难以避免引入一些错误或缺陷。如图1.1所示，Eclipse3.0.1 中也会存在着一些明显的空指针引用错误 [6]。公开数据显示，对于有经验的程序员编写的代码，每 1000 行就有 50-250 个错误，平均每 1000 行会有 100 个缺陷，即使是经过软件故障控制管理培训的软件工程师，平均每 1000 行代码中存在 50 个故障 [7]。因此，整个行业在关注软件如何提升生产力的同时，也更加注重提高软件源代码编写的质量，加大了对源代码的检测力度，以期及早发现代码中潜在的安全隐患，避免或减少因为软件缺陷带来的损害。据统计，现在在软件开发总成本中，投入到软件测试中的资源约占到 25% 到 50%[8]，并贯穿在软件生命周期的各个阶段。

在数量繁多的软件缺陷中，空指针引用缺陷是相对比较常见的缺陷类型，广泛存在于不同的编程语言中。同时，它也是最影响软件系统可靠性和稳定性的故障之一。

```
// Eclipse 3.0.1
if (in == null)
  try {
     in.close();
  } catch (IOException e) {}
```

图 1.1　Eclipse3.0.1 中存在的空指针引用缺陷

通过人们对故障的总结，人们发现常见的比较大型的软件故障有数组越界，资源泄漏，空指针引用等，其中空指针引用问题出现的尤为频繁，根据 coverity 公司 2009 年针对 280 个开源项目的故障分析报告，空指针引用在所有类故障中所占比例为 27.81%，是所占比例最高的故障 [11]。

中国国家信息安全漏洞库 (CNNVD) 统计，2013 年共发现空指针引用引发的漏洞共 35 个，这些漏洞存在于操作系统、服务器应用程序等软件系统中，漏洞类型有拒绝服务、代码注入、信息泄露、一区溢出、数字错误等。这些漏洞一旦被恶意攻击者利用, 可能导致系统崩溃，服务器程序可能会拒绝服务, 或者机密信息泄露，这都将严重的影响软件的运行以及系统的安全。根据对国内航空航天、武器装备、金融、电信等数千万行国产软件应用 DTSC 的测试报告统计，在所有故障类缺陷中，空指针引用缺陷大约会占到 30% 左右，空指针引用缺陷的密度大致是 0.3/KLOC。

由此可见，空指针引用缺陷的清除对于程序的稳定性和安全性都具有巨大价值，而针对空指针引用缺陷的检测技术研究也就具备了重大的意义。

1.2　国内外研究现状及发展趋势

软件缺陷检测相关的研究几乎是伴随着软件的产生而出现的，随着程序设计语言的发展，软件缺陷类型也越来越多。通过人们对故障的总结，人们发现常见的比较大型的软件故障有数组越界，资源泄漏，空指针引用等，其中空指针引用问题出现的尤为频繁，根据 coverity 公司 2009 年针对 280 个开源项目的故障分析报告，空指针引用在所有类故障中所占比例为 27.81%，是所占比例最高的故障 [11]。

以 Java 语言为例，Null 关键字的广泛使用是 Java 代码中产生 NPE（Null Pointer Exception）故障的直接原因，Haidar Osman[12] 等人开发了 NullTracker 工具对 810 个开源 Java 项目进行简单的数据流分析，追踪代码中空指针检查语句的分布情况，以发现程序开发者使用 null 关键字的时机和目的，结果表明在所有的条件判断语句中，用

来进行空指针检查的语句平均占比为35%，类成员没有初始化，方法返回null值，以及向方法中传递null值是引发NPE的最常见的因素。其中，71%的空指针检查语句用来保证方法调用返回值的安全性。由于空指针检查语句的频繁使用，可能导致程序运行时的开销增加2%-10%，不仅如此，空指针检查的频繁使用还会降低代码的可读性和可维护性，而一旦缺失了这种检查，程序的稳定性便无法得到保障。

由于空指针引用故障在代码中广泛存在而又十分隐蔽，但是其一旦出现很大可能会导致程序崩溃，因此对程序的稳定性具有非常大的威胁。针对空指针引用缺陷的检测一直备受关注。

目前，代码缺陷检测的技术从大的层面上主要分为两大类，静态检测和动态检测。

动态检测 [9] 主要侧重于软件的性能、功能完善等方面，通过动态测试来进行漏洞的探测与发现，不仅仅要求测试人员对缺陷特性具有较深入的理解，测试过程中还需要大量测试用例，在目前软件规模愈加庞大，逻辑愈加复杂的情境下，这种方式必然带来大量的人力和物力的浪费。

静态检测 [10] 是指利用静态分析手段来探测程序中潜在缺陷的方法，不需要运行程序使用其他手段完成对程序结构分析的技术。相较于动态分析技术而言，静态分析成本较低，而且能有效的对代码中的缺陷进行精确定位。因此，对源代码的静态分析和缺陷检测是一个值得深入研究的方向。

1.3 本文研究内容

静态分析技术与工具具有较早发现缺陷、覆盖率高、低开销、自动化程序高等优点；同时静态分析技术也存在一定的局限性，不仅要在分析效率与精度中做出取舍，还需要在误报率与漏报率之间做出取舍。现有的一些静态检测工具（如Findbugs,PMD）都采用了不同的实现方法达到这样的平衡，由于采取的分析策略的不同，形成的检测结果也往往有较大差异。

基于当下Java代码空指针引用缺陷检测工具的特点，本文提出了利用交叉验证的方式整合不同工具的检测能力，以提高检测结果准确率的方法。研究内容如下：

(1) 开发多工具缺陷代码检测系统BIT-Detector，将多种Java空指针引用缺陷静态缺陷检测工具以插件的形式集成在SonarQube平台上，针对目标代码进行一次性地同时检测，将检测结果按照优先级排序，即缺陷被检测出的工具数量越多，排序越靠前，缺陷真实的可信度越高。

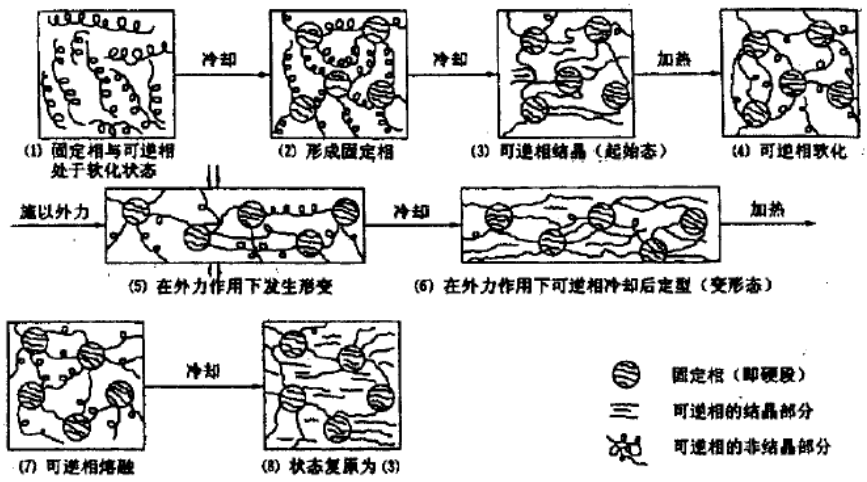

图 1.2　热塑性形状记忆聚氨酯的形状记忆机理示意图

(2) 提取缺陷代码特征，将缺陷代码的控制流图转换成向量，利用深度学习的方法搭建神经网络模型帮助选择目标代码适合的工具，进一步优化 BIT-Detector 的检测结果。

1.4　论文结构

1.4.1　形状记忆聚氨酯的形状记忆机理

形状记忆聚合物（SMP）是继形状记忆合金后在 80 年代发展起来的一种新型形状记忆材料[1]。形状记忆高分子材料在常温范围内具有塑料的性质，即刚性、形状稳定恢复性；同时在一定温度下（所谓记忆温度下）具有橡胶的特性，主要表现为材料的可变形性和形变恢复性。即“记忆初始态－固定变形－恢复起始态”的循环。

固定相只有物理交联结构的聚氨酯称为热塑性 SMPU, 而有化学交联结构称为热固性 SMPU。热塑性和热固性形状记忆聚氨酯的形状记忆原理示意图如图1.2所示

1.4.2　形状记忆聚氨酯的研究进展

首例 SMPU 是日本 Mitsubishi 公司开发成功的……。

表 1.1　水系聚氨酯分类

类别	水溶型	胶体分散型	乳液型
状态	溶解 ~ 胶束	分散	白浊
外观	水溶型	胶体分散型	乳液型
粒径 $/\mu m$	< 0.001	$0.001 - 0.1$	> 0.1
重均分子量	$1000 \sim 10000$	数千 ~ 20□	> 5000

1.4.3　水系聚氨酯及聚氨酯整理剂

水系聚氨酯的形态对其流动性，成膜性及加工织物的性能有重要影响，一般分为三种类型[1]，如表 1.1所示。

由于它们对纤维织物的浸透性和亲和性不同，因此在纺织品染整加工中的用途也有差别，其中以水溶型和乳液型产品较为常用。另外，水系聚氨酯又有反应性和非反应性之分。虽然它们的共同特点是分子结构中不含异氰酸酯基，但前者是用封闭剂将异氰酸酯基暂时封闭，在纺织品整理时复出。相互交联反应形成三维网状结构而固着在织物表面。……

第 2 章　相关工作

为了保证软件的可靠性和稳定性，在大量研究人员长期不懈地努力下，出现了很多针对软件缺陷的检测方法。这些方法可以在软件开发周期的不同阶段介入，检测的效率和效果也大不相同，最终涌现出了一批相对成熟的代码缺陷检测方法和工具。另一方面，随着软件数量的日益庞大，以及数据挖掘技术在各个研究领域的广泛应用，利用机器学习的方式来解决软件安全问题也逐渐成为了研究热点。

2.1　程序静态分析技术

静态分析技术即是在不运行程序，不依赖程序输入的情况下对程序代码进行分析的一项技术。这种技术有助于开发人员对代码结构的理解，同时也能检测潜在的安全缺陷 (如 SQL 注入)，运行时错误 (如空指针引用缺陷) 以及部分代码逻辑错误。它一般需要配合利用自动化工具执行分析。采用的技术有数据流分析，机器学习，语义精简等。可检测死锁，空指针，资源泄露，缓存区溢出，安全漏洞，竞态条件等软件缺陷。具有快速，准确，伸缩性强等特点，能够在代码开发阶段找到并修复多种问题，从而节省大量人力成本和时间。下面对部分静态分析方法涉及的相关技术进行简要的介绍。

符号执行静态分析中较常用到的一种技术，它可以利用抽象符号描述程序执行过程中变量值，这种方法可以很好地模拟程序的运行过程。相对于传统方法无法确定程序真实执行下各变量值的情况，此方法在对程序进行路径敏感分析时十分有效。不过因为符号执行方法会追踪程序中所有变量的所有取值空间，所以在应用于大规模代码进行分析时，可能会导致分析的可能路径数量迅速增多，因此在应用该方法的时候，往往会采取优化路径数量的方法即选择部分可能性最高的路径进行分析，这样虽然可以避免状态爆炸的产生，但是也难免会导致分析精度的下降。

PREfix 是一种针对 C 语言的静态分析工具，它采用了符号执行的方法。该工具可以对程序每个可能的执行过程进行抽象建模，静态地模拟程序的多个可能执行路径，同时利用约束求解对程序分析过程中出现的约束集合进行检查。此工具能够做到路径敏感的缺陷检查，但是由于符号执行方法的特性，为了避免状态爆炸的情况出现，它只能选取部分路径进行分析，这就导致了分析精度的不理想。

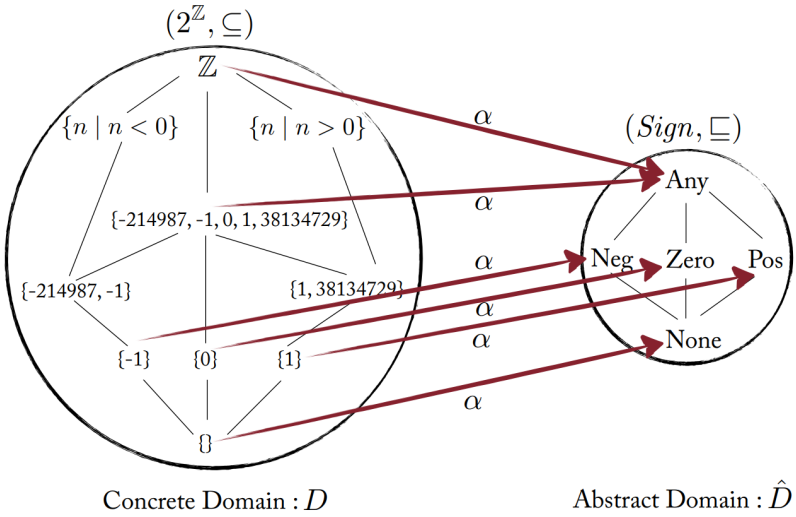

图 2.1　将整数空间的具体值域映射到抽象值域

模型检查也是一种常见的静态分析技术，通常的做法是构建有限状态机或者有向图等抽象模型，再对构造出的模型进行遍历来检验待检测系统的部分性质。SLAM 是一种具有代表性的基于模型检查的静态分析工具。它可以从待检测代码中抽象出一个布尔程序并加以验证。在得到的错误报告中逐个检查，找出所有误报，进而根据这些误报对抽象的布尔程序进行调优，经过不断迭代，最终可以取得很好的效果。

同样是应用于程序验证的技术，不同于模型检查，定理证明是基于语义的程序分析方法。但是由于采用了消解原理的定理证明器，而这种方法对整数域和有理数域相关的运算不是很好处理，所以应用在程序分析领域显得不是特别合适。基于这种问题，研究人员通常会选择各种判定过程来确定公式是否是定理。ESC 就是一个采用了定理证明技术的半自动化工具，它在分析程序的过程中需要外界指定所涉及的类不变量，过程不变量以及循环不变量。

除了上面提到的 3 种技术，抽象解释的应用更加广泛，它是数据流分析的理论基础。1977 年 P.Cousot 和 R.Cousot 共同提出了抽象解释的理论，应用该理论分析程序时就不需要拘泥于程序最底层的具体细节上可以在更高的角度上去观察和思考。传统解释器可以知道程序中每个变量具体的值从而得到具体值域，而抽象解释不同与传统解释的地方就是可以得到一个更高阶的抽象值域。如果将一个传统解释器迁移到抽象解释器，几乎等同于构造一个函数把具体值域映射到抽象值域。如图2.1所示，我们可以将无限的整数具体值域抽象成正，负，零三个抽象值域，这个过程只需要实现一个抽象函数 α 即可完成。

抽象解释理论实际上是从代码中抽象出一些能够刻画我们想要分析的问题所需要的特征，本质上还是为了提升分析的效率而损失部分分析精度的方法。但是如果应用得到，还是可以获得很大的收益，现在几乎所有的数据流分析方法都应用了该理论。

总的来说，静态分析技术在不运行代码的情况下进行可以有效避免了程序运行环境的苛刻要求，可以针对程序的规模采取灵活的分析方法，从而具备了较早发现缺陷，较低的分析成本，较高的覆盖率和自动化程序等优点，但是由于往往需要对分析精度进行部分舍弃，导致了分析结果的漏报率和误报率都无法达到特别理想的水平。

2.2 静态分析技术在空指针引用缺陷检测的应用

针对空指针引用缺陷，研究人员已经利用静态分析技术做出了很多实践并取得了一定成果。

Haidar Osman 等人利用数据流分析技术考察了 Java 语言空指针缺陷产生的情况，研究表明 Null 关键字的广泛使用是 Java 代码中产生 NPE（Null Pointer Exception）故障的直接原因。他们开发了 NullTracker[12] 工具对 810 个开源 Java 项目进行简单的数据流分析，追踪代码中空指针检查语句的分布情况，以发现程序开发者使用 null 关键字的时机和目的，结果表明在所有的条件判断语句中，用来进行空指针检查的语句平均占比为 35%，类成员没有初始化，方法返回 null 值，以及向方法中传递 null 值是引发 NPE 的最常见的因素。其中，71% 的空指针检查语句用来保证方法调用返回值的安全性。由于空指针检查语句的频繁使用，可能导致程序运行时的开销增加 2%-10%，不仅如此，空指针检查的频繁使用还会降低代码的可读性和可维护性，而一旦缺失了这种检查，程序的稳定性便无法得到保障。

研究人员利用静态分析技术在 Java 空指针引用缺陷上进行了大量的工作，产生了很多检测空指针引用的工具和技术，这些技术可粗略的分为指针引用验证和空指针引用 [13] 缺陷检测两大类。前者侧重于如何验证程序中的指针是否为空。后者侧重于如何尽可能多的发现程序中的空指针引用。指针引用验证技术是基于需求驱动的思想 [14]，一般是首先识别出指针，再沿着控制流后向的验证指针是否为空。空指针引用缺陷检测一般是在进行数据流分析 [15]、指针分析的基础上，根据一些规则基于控制流前向的检测。两者通常都需要进行数据流分析与指针分析。

Salsa[16] 是一个致力于验证 Java 代码中指针引用安全性验证的工具，通过定制的数据表示形式进行前向数据流分析，通过对传播深度和数据流传播路径数量的简单

限制来获得方法的可扩展性，同时依赖预先进行的必然别名分析来提高方法间数据流分析的准确性。由于一些空指针的引用需要经过多层方法调用链才有可能触发，这种验证方式会产生很多漏报的同时，效率也不理想。数据流分析技术具有十分灵活的特点，为了提高效率，Ravichandhran Madhavan[17] 等提出了一种过近似的最弱前置条件分析方法以验证 Java 程序中指针的安全性，该方法通过需求驱动的前向数据流分析大幅提升了单个引用的分析效率，该方法试图找到程序入口处可能满足被分析程序点的引用不安全的条件，如果存在这样的条件，则可以判定该引用不安全。此方法的数据流事实为有限的谓词集合，通过有选择地限制谓词集合的大小以及传播路径的数量，该方法可以做到低延迟的流敏感，上下文敏感的 sound 分析，利用 Wala[18] 程序分析框架，可以取得较好的验证引用安全的效果，但是过于追求针对单个引用的需求驱动分析，在对大规模代码中的引用进行批量分析时性能欠佳。

空指针检测相比于指针安全性验证更加具有实用性，而误报率和漏报率是检验工具实用性的重要指标，空指针检测工具大多不追求完美的正确率，而将较低的误报率和较高的召回率作为最重要的目标。

FindBugs[6][19] 是一个开源的针对 Java 代码的缺陷静态检测工具，通过分析 class 文件，在字节码层级进行简单的前向数据流分析，对程序中的每一个引用的是否为 null 值的不同情况，给定相应的标识从而在触发可能的空指针调用时给出不同的告警等级。对于指针引用 FindBugs 总结出了一些经验规则，对不可达路径、控制流汇合、指针赋值语句、断言等特定情况定制了专用的检测规则，在进行过程间分析时，其主要依赖特定故障模式以及用户编码时给出的注解来推断空指针是否可能发生，所以它只能检测出特定场景下的空指针引用。

检测工具 Xylem[20] 从每一个指针引用出发，进行基于需求驱动的后向数据流分析，并将谓词作为数据流事实，目标是能够高效的检测出最重要的空指针引用，在进行分析时采取的是不完全可靠的分析方法，检测结果存在较多漏报。

北京邮电大学的杨睿 [21] 提出一种 Java 中空指针引用故障的静态检测方法，将空指针引用问题抽象为一类故障模型，并以故障模式状态机来形式化描述此类故障模型，然后根据故障状态机的创建条件及待检测代码的语义信息确定是否创建该类型的状态机，并将创建的状态机示例置于控制流图入口，根据数据流分析的结果对故障状态进行迭代以检测空指针引用问题。

中国矿业大学的姜淑娟 [22] 提出一种空指针异常自动定位方法，该方法结合程

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。

总体来看，以上这些分析方法都有各自的优缺点，但是目前无法找到一种完美的静态检测方法可以兼顾缺陷检测的误报率和漏报率。这也正是静态代码分析的短板，不仅是将复杂的缺陷解释出来很困难，对于结果的高误报率，往往显得无能为力。

2.3 代码缺陷检测工具介绍

对于空指针引用缺陷，工业界已经产生了很多优秀的检测工具，这些工具具有不同的实现原理，对空指针引用缺陷的检测结果也不尽相同。

FindBugs[6][19] 是一个开源的针对 Java 代码的缺陷静态检测工具，通过分析 class 文件，在字节码层级进行简单的前向数据流分析，对程序中的每一个引用的是否为 null 值的不同情况，给定相应的标识从而在触发可能的空指针调用时给出不同的告警等级。对于指针引用 FindBugs 总结出了一些经验规则，对不可达路径、控制流汇合、指针赋值语句、断言等特定情况定制了专用的检测规则，在进行过程间分析时，其主要依赖特定故障模式以及用户编码时给出的注解来推断空指针是否可能发生，所以它只能在特定场景下检测出空指针引用缺陷。

Jlint 同样是一个开源静态代码检测工具，它通过执行数据流分析和构建锁图来查找缺陷，语义矛盾和同步问题。Jlint 有两个独立的程序来执行语法和语义验证。通过使用手写扫描器和简单的自顶向下解析器，Jlint 能够检测到一些代码缺陷，例如可疑地使用操作符优先级，没有切换代码中断，对构造体错误的假设等。同时，Jlint 执行本地和全局数据流分析，计算局部变量的可能值并捕获冗余和可疑计算。通过执行全局方法调用分析，Jlint 能够检测具有可能为“null”的形参的方法的调用，并且在没有验证“null”的方法中使用该参数。Jlint 还为类依赖项构建了锁依赖关系图，并使用该图来检测在多线程程序执行期间可能导致死锁的情况。除了死锁之外，当不同的线程可以同时访问相同的变量时，Jlint 能够检测到可能的竞争条件问题。Jlint 最大的特点就是检测的效率很高，但是由于使用的数据流分析十分有限，因此误报率也较高。

Infer 是 Facebook 的开发团队在代码提交内部评审时，用来执行增量分析的一款静态分析工具，在代码提交到代码库或者部署到用户的设备之前找出缺陷。由 OCaml 语言编写的 Infer 目前能检测出空指针访问、资源泄露以及内存泄露，可对 C、Java 或

Objective-C 代码进行检测。Facebook 使用 Infer 自动验证 iOS 和安卓上的移动应用的代码，bug 报告的正确率达 80%。Infer 通过捕获编译命令，把要被编译的文件转换为可用于分析潜在错误的中间语言格式。整个过程是增量进行的，意味着通常只有那些有修改过并提交编译的文件才会被 Infer 分析。Infer 还集成了大量的构建或编译工具，包括 Gradle、Maven、Buck、Xcodebuild、clang、make 和 javac。此外，Infer 根植于两大基本理论之上，其一是霍尔逻辑，一种用于推理计算机程序正确性的形式系统，另一个是抽象解释，该理论用于测度程序语义的逼近结果，此外还涉及其它一些研究成果，例如 Separation Logic 和 Bi-abduction。

Fortify SCA 是一款应用广泛的商业工具，由知名的惠普公司出品，是一个白盒的、静态的软件源代码安全检测工具。它通过内部的五种主要分析引擎：语义、结构、控制流、数据流、配置流等对应用程序的源码进行静态分析，在分析的同时与该工具特有的软件安全漏洞规则集进行全面地查找、匹配，进而找出源代码中存在的各种缺陷和漏洞，并整理和产出缺陷报告。Fortify 应用十分广泛，在世界范围内被大量公司用作内部源代码的质量安全检测工具。

Coverity 是美国 Coverity 公司提供的可配置的用于检测软件缺陷和安全隐患的静态源代码分析解决方案，该工具基于布尔可满足验证技术应用于源代码分析引擎，分析引擎利用其专利的软件 DNA 图谱技术和 meta-compilation 技术，综合分析源代码、编译构建系统和操作系统等可能使软件产生的缺陷。Coverity 是第一个能够快速、准确分析当今的大规模、高复杂度代码的工具，它解决了影响源代码分析有效性的很多关键问题，如编译兼容性，构建集成，高误报率，有效的错误根源分析等。

2.4 深度学习技术在软件安全领域的应用

深度学习是人工神经网络中一种多层级学习框架，试图通过构建深层网络模拟人脑感知抽象概念的能力。近年来，深度学习凭借着强大的特征学习能力，问题表达能力，数据容纳能力，掀起了又一次机器学习的浪潮，并在计算机视觉，语音处理，自然语言处理等众多领域取得了巨大进展，受到从学术界到工业界的广泛关注。现在，深度学习技术也开始渗透进软件工程的多个领域。

2.5 本章小结

本章首先介绍了静态分析涉及的相关技术背景，然后以空指针引用的检测为例，介绍了国内外研究人员利用静态分析技术在空指针引用缺陷检测方面的进展，随后对一些业界成熟的静态代码缺陷检测工具，如 Findbugs，Jlint，Infer，Fortify 等进行了简要的介绍，最后对当下热门的深度学习技术在软件安全领域的应用进行了讨论。

第 3 章　总体架构

本论文的目标是引入深度学习的方法针依据不同工具对代码的检测能力对大量缺陷代码进行分类，在使用多种检测工具对同一份代码进行检测时，可以有效得知这些工具针对这份代码的检测能力，然后利用这些工具的检测报告给出更精准的评判，从而提升检测精度。

3.1　设计背景

随着软件规模的增大，缺陷检测的难度和所需要的代价都越来越大，就 Java 语言的空指针引用异常的检测来说，目前业界存在着很多工具。如 Findbugs，Jlint，Infer，Fortify 等。他们在检测缺陷时使用了模式匹配，数据流分析，类型系统，模型检查等技术，由于不同的技术出于对检测精度和效率的权衡，他们所产出的检测报告往往各不相同，并且几乎都包含了大量的误报和漏报，开发人员在面对这样复杂的报告时，很难判断某条报告的准确性。NickRutar[A comparison of Bug Finding Tools for Java] 等人针对五种 Java 语言的缺陷检测工具做了比较，发现没有任何一个单一的工具是完美的，此外，不同工具所产出的报告之间也有不小的差异。

基于这种情况，可以设想将多种工具的报告汇总到一起进行交叉验证，如果多个工具同时给出了同一个位置出现同一种缺陷的报告，我们有理由相信这个缺陷是真实可信的。因此，本文基于 sonarqube 平台开发了插件 BIT-Detector，这个插件集成了 Findbugs，Jlint，Infer 和 Fortify 的能力，针对同一份待测代码，首先使用 4 种工具分别检测并给出报告，然后过滤出空指针引用缺陷，最后将 4 份报告的内容格式统一化从而进行比对，将不同工具同时检出的空指针引用缺陷作为 BIT-Detector 的输出，这样的结果理论上可以达到很高的准确率。

由于难以找到合适的空指针引用缺陷数据集，为了对这些工具进行合理的评测，本文采用一种特别的方式构建了一批可信的测试用例，这些用例的构造方法会在后面的章节详细说明。利用构建出来的 7429 个测试用例，我们针对上文提到的四种工具以及 BIT-Detector 进行了测试。图3.1反映了各个工具检出的空指针引用缺陷的重叠情况，表3.1给出了不同工具检测的精度信息，同时还给出了 BIT-Detector 的数据。通过对比不难发现, 各个工具的检测结果确实有较大差异，即使我们使用检测准确度最高

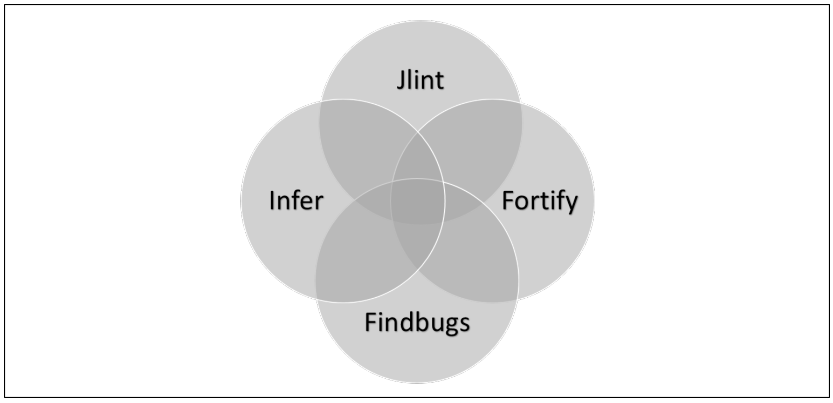

图 3.1　4 种工具在 7429 个测试用例上的检测结果

表 3.1　四种工具和 BIT-Detector 的测试结果对比

测试工具	正报	误报	准确率	召回率
Findbugs	5402	3169	63%	72%
Jlint	4933	9137	35%	66%
Infer	5051	3578	58%	68%
Fortify	3766	2211	63%	50%
BIT-Detector	3404	576	85%	45%

的 Findbugs，也会面临超过三分之一的误报，这些误报掺杂在检测报告中会给开发人员的缺陷修复带来很多困扰，可能很多时间和精力都会被浪费在验证缺陷的真实性上面。而保证报告的准确性应该是对检测工具的基本要求，即使不谈准确率，过多的漏报也会让人沮丧。显然，目前被广泛使用的各种检测工具还有很大的提升空间。

在检测过程中，我们也注意到选取所有工具共同检测缺陷作为输出结果的 BIT-Detector 在检测结果的准确率方面有着突出表现，相对于表现最好的 Findbugs 和 Fortify，BIT-Detector 有着 22% 的准确率提升, 而相对于准确率较低的 Jlint 而言，BIT-Detector 的准确率提升则是成倍的增加。即使召回率的表现不尽人意，但是如果我们不将除了四种工具同时检测的缺陷排除在外，而是将四种工具的检测结果按照优先级排序，召回率在某种程度上反而是提升的，如果按照这种方式处理检测报告，此时 BIT-Detector 的检测结果将自然地排在最前面，但是对其他结果的排序就成了一个棘手的问题，假如我们有四种工具，分别记为 T_a，T_b，T_c，T_d。如果工具 T 在被测代码的 L 位置成功检出了某个缺陷 D，则记为 $E(T, L, D) = 1$，反之则记为 $E(T, L, D) = -1$。给定位置 L_i，存在如下这种情况

$$\begin{cases} E(T_a, L_i, D) = 1; \\ E(T_b, L_i, D) = 1; \\ E(T_c, L_i, D) = -1; \\ E(T_d, L_i, D) = -1 \end{cases}$$

在这种情况下，开发人员很难判断位置 L_i 处是否存在缺陷 D，由上文已知，不同工具对同一份代码的检测结果不同，且它们的检测能力往往不能互相覆盖。所以，对于缺陷 D, 如果我们已知工具 T_a，T_b，T_c，T_d 对该缺陷的检测能力，那么对于检测结果的可信度就可以依据不同工具对于该缺陷的检测能力来确定。如果针对不同的缺陷类型，如空指针引用缺陷，资源泄露缺陷，跨站脚本引用缺陷等对不同工具进行能力评估是比较简单的，只需要计算不同工具在特定类型缺陷上的准确率就能大致评估在该类缺陷上不同工具的检测能力，但是如果只限定在单一的缺陷类型上，例如缺陷 D_1，D_2 都是空指针引用类型的缺陷，确定不同工具对缺陷 D_1，D_2 的检测能力就是比较棘手的事情了。我们需要找出缺陷 D_1 和 D_2 在代码结构和语义上的不同之处，还需要找出不同工具在检测策略上对这两种缺陷代码处理的不同之处。

显然，利用人工去分类所有空指针引用缺陷是相当困难的事情，分类的种类是否全面，粒度是否精确都会对评估不同工具的能力的结果造成很大影响。面对这种情况，深度学习具备相当不错的解决方案，如前文相关工作中提到的，研究人员已经利用深度学习方法在代码的分类上取得了相当不错的效果，受次启发，本文将利用深度学习的方法评估不同工具对空指针引用缺陷的检测能力。

3.2 设计思路

从上一节设计背景中的讨论可知，不同工具对空指针引用缺陷的检测能力是不同的，而这种能力也并不能很容易地进行评估，这就造成 BIT-Detect 不能对多个工具的检测报告进行很完美的交叉验证，在检测报告中对于同一位置的缺陷，不同工具可能会给出不同的判定结果，本文提出使用深度学习的方法，在大量缺陷代码数据集的基础上，利用深度神经网络根据代码和工具检测的结果训练出判定不同工具对特定代码检测能力的模型。

深度学习的方法需要大量的训练数据才能得到表现良好的模型，所以首先应该解决开源空指针引用缺陷用例缺乏的问题，然后为了方便后面的数据训练，需要将代码

的结构特征转化为数学空间的向量特征，在这个工程中需要解决代码的控制流提取以及代码特征抽取问题，之后可以利用 Structure2Vec 算法完成图到向量的转化。最后需要给这些数据打标，即通过不同工具的检测得到这些工具在用例上的检测结果，最后将标注过的数据作为模型的输入，完成整个训练过程。

(1) 数据集构建

数据集的构建是模型训练所必须的工作基础，在开源项目中很难找到可用的空指针引用缺陷用例。为了达到良好的训练效果，用例应该具备完整的语义，正确的逻辑，多样的代码结构，至少一条可达路径上必然会出现空指针引用缺陷等特点，考虑到后期的处理和训练，作为用例的程序不应过大，恰好包含满足空指针引用缺陷产生的上下文最佳。加上这些要求，在开源环境下直接获得可用的训练数据集就更困难了。

考虑到 LeetCode，ACM 等具备竞赛性质的项目下可以获得大量代码完整，逻辑正确同时工程精简的代码集，数据集的构建可以通过改造这些代码来完成。为了增加代码的多样性，还可以加入 OWASP，ONIST 等跟代码缺陷检测相关的项目下的数据集，另外也可以通过人为构造部分数据集作为补充。

(2) 控制流图生成

代码的结构化特征可以通过控制流图很好的表达出来，不同程序之间的结构化差异很大程度上可以通过控制流图反映出来，所以生成正确的程序控制流图是后续模型训练的重要保证。

在具备相当数量的缺陷用例数据集后，需要将这些用例的控制流图提取出来。这个过程可以利用现有的工具来完成，例如 SOOT，LLVM 等工具，它们可以很方便地构造出 Java 代码的控制流图和方法间调用图，利用这两方面的信息，基本可以提取到代码所有的结构化特征。从本文关注的角度来看，只需要提取出空指针缺陷发生的上下文信息即可，即提取从变量赋值为 null 到变量被解引用产生异常为止的控制流子图。

(3) 代码特征抽取

控制流图只能反映出程序的结构化信息，对于空指针引用缺陷的检测来说，工具需要更多地理解代码的语义信息，一个赋值语句的差别可以很容易地决定是否能够产生空指针引用缺陷。如果要体现程序的语义信息，需要尽可能多地抽取出每一句代码的静态特征，如是否将 null 赋值给了变量，是否针对 null 值进行了检查，是否调用了其他方法等。这些特征将附着在控制流图的每一个结点上，为后续的模型训练提供更

多的代码特征信息。这些工作同样可以利用 SOOT，LLVM 等工具来完成。

(4) 数据标注

对于一个给定的程序片段，模型理想的输出结果为不同工具分别是否具备检测该段代码的能力，即这些工具检测结果的可信度。那么本方案中数据的标签即是不同工具对该用例实际检测的能力，即是否能够检测出该用例设计的缺陷。这个标注过程可以利用上节提到的 BIT-Detector 插件来批量完成。

(5) 模型训练

模型训练大致可以分为两部分，第一部分利用 Structure2Vec 算法对上述步骤中产生的包含特征向量信息的代码控制流图向量化，第二部分将这些向量化后的图向量和相应的标签一起针对不同的工具分别进行分类训练。每个模型都可以针对给定的代码判断该工具能否正确检测出缺陷。

(6) 结果验证

最后，可以利用不同工具的模型得到它们对于待测代码能否检测出缺陷的能力判断，综合这些工具在该代码上实际的检测结果，可以得到该代码是否具备缺陷的最终判定。

3.3 整体架构

本文的最终目的是基于 sonarqube 平台开发一种空指针缺陷引用检测系统，下面简单介绍一下该系统的运行模式，如图3.2所示，SonarQube 平台可以非常方便地与开发人员使用的 IDE，代码版本管理工具以及持续集成框架集成在一起。将这些工具整合到一起之后的工作流大致有如下步骤：

(1) 用户在 IDE 中完成代码编辑之后推送代码至 SCM（SoftWare Configuration Mangement）仓库，选用的工具可以是 SVN，Git 等。

(2)Jenkins 通过配置可以与核心仓库中的代码进行同步，并能感知仓库代码是否生了变动。

(3)Jenkins 在感知到核心仓库的代码发生改变之后，自动触发代码构建动作，构建工具可以选用 Maven，Gradle 等。代码成功构建之后，Jenkins 会触发 SonarScanner 扫描代码的操作。

(4)SonarScanner 首先加载配置文件记录的代码位置及目录结构信息，然后由插件

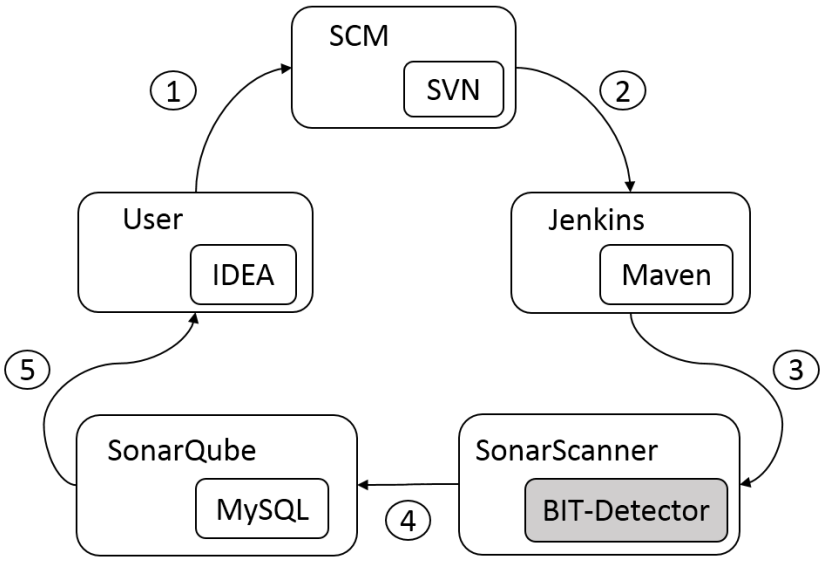

图 3.2　SonarQube 平台工作流程

负责对代码进行全面检测，并生成检测报告。最后检测报告将按照特定格式传递给 SonarQube 服务器。

(5)SonarQube 服务器会处理 SonarScanner 传递的代码检测报告，并将报告数据存储在 MySQL 等数据库中，一个工程同时可以拥有多份不同批次的检测报告，最后这些报告将会以合适的方式可视化给用户查阅。

本文重点介绍 BIT-Detector 的部分设计开发工作，BIT-Detector 是以插件的形式在 SonarScanner 中承担着检查代码的工作。它的主要工作流程如图3.3所示。首先，用户需要针对特定的工程配置好 sonar-scanner.properties 文件，文件中记录了工程的名称，SonarQube 项目的代号，版本等基本信息，还有工程的源码，测试代码及编译后的字节码文件所在目录和依赖库等信息的位置。这些信息将提供给插件中的代码检测工具使用。在获得必要的信息之后，调用不同的工具对代码进行检测，然后根据不同工具的特性收集相应的检测报告。不同的工具所产生的报告格式也是不一样的，在交叉验证阶段，需要将这些报告的格式统一处理，同时还需要过滤掉不需要的缺陷类型，例如本文重点关注空指针引用缺陷，只需要留下空指针引用缺陷类型即可。后面就可以根据汇总的缺陷数据对检测报告进行粗略的分类，将所有工具都能检测出来的缺陷选取出来，作为最高优先级的缺陷，其他缺陷需要经过模型进行进一步的验证。需要模型验证的用例需要确定分析域范围，即空指针产生的上下文，这些信息可以通过代码检测工具得到，然后提取出这部分的控制流图，抽取出代码特征，在通过模型进行向量化之后可以利用训练好的分类器得到不同工具对该代码的检测能力评价，即工具

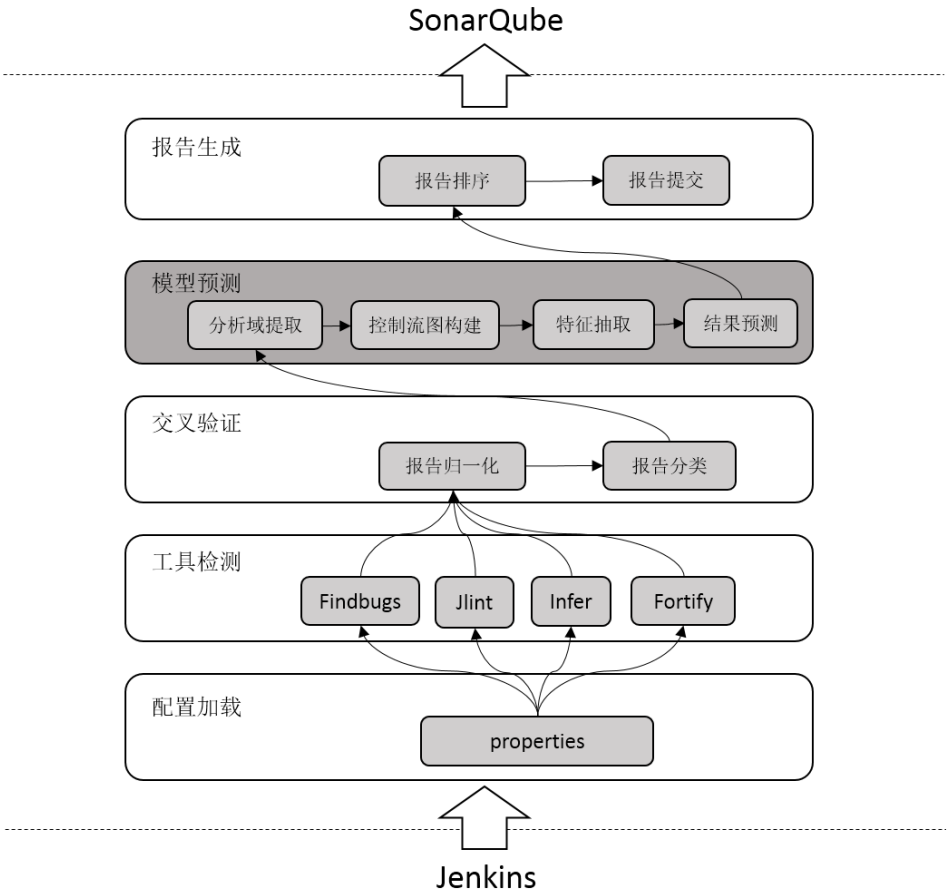

图 3.3　BIT-Detector 工作架构图

的置信度。利用工具的置信度可以对工具检测的结果进行适当修正，并根据可信程度对缺陷进行进一步的排序。最后将报告提交给 SonarQube 进行可视化输出。

利用模型评价工具的检测能力是本文重点研究的部分，后续的章节将会重点介绍模型训练相关的工作，其他关于插件开发的工作不再过多阐述。

3.4　本章小结

本章介绍了 BIT-Detector 的设计背景，详细讨论了该工具的设计思路，特别交代了在工具中引入深度学习方法的原因。之后介绍了基于 SonarQube 平台的代码检测流程，重点介绍了 BIT-Detector 的架构设计和工作过程，最后阐明全文研究的重点聚焦在深度学习模型的实现和训练上面。

第 4 章　数据集的构建和预处理

上一章节提到了验证工具的检测能力需要大量的测试用例，由于合适开源用例的稀缺，合理利用现有无缺陷代码来生成空指针引用缺陷是一种可行的方法。但是，为了方便后续的处理工作，测试用例的选择应该从多维度慎重考虑。然后，为了模型训练的顺利进行，还需要对这些数据进行预处理，构建出正确的控制流图，提取出合适的代码特征。这些工作都是神经网络模型训练所依赖的重要基础。

4.1　数据集构建

4.1.1　数据来源

如果采用从正常代码中构建空指针引用缺陷的方式，首先面对的问题就是选择构建用例的合适的原材料。为了便于后期处理，用例应该包含程序入口，具备语义完整，结构多样化，代码规范简洁等特点，只要程序包含常见的语法结构和调用关系，程序规模不应过大，恰好包含空指针引用陷产生的上下文最佳。依据这些条件，本文选择了部分开源代码数据集作为构建空指针引用缺陷的原始资料，如图4.1所示。

其中，OWASP[+] 和 ONIST[+] 分别为代码缺陷检测相关领域的项目，从这些项目下可以获得部分标准的空指针引用缺陷用例，同时也可以得到很多具备其他缺陷

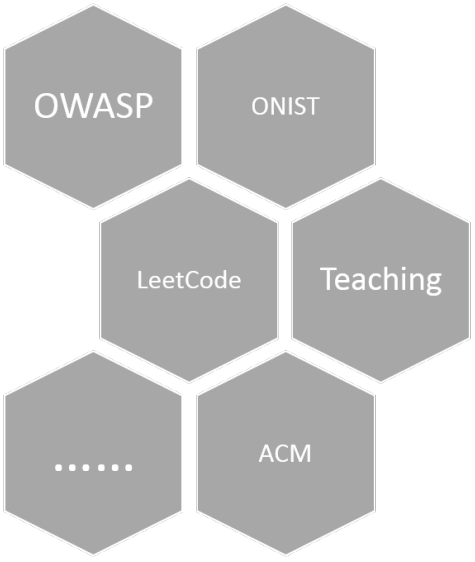

图 4.1　测试用例来源

的测试用例，由于这些用例的代码编写较为规范，并且经过了合理分类，还往往包含说明文档等辅助理解代码的信息，大多都可以用来生成空指针引用缺陷。除此之外，LeetCode 和 ACM 作为编程竞赛性质的项目下也包含大量可以利用的代码，这些代码根据题目的难易级别具备着不同的复杂程度，包含了多样的代码结构，同时，代码的规模往往不大，是作为测试用例生成的良好材料。另外，一些供教学使用的代码也是十分合适的空指针缺陷构造来源，作为补充，本文还添加了部分人工编写的测试用例。

4.1.2 测试用例生成

在取得构造测试用例的原始代码后，需要对这些代码进行检查，确保代码有正确的程序入口，并且可以正确执行。对于 OWASP 和 ONIST 项目下的代码，很多用例缺乏 Main 方法的入口，这会导致后续控制流图提取的困难，这时需要在源代码层级加入合适 main 方法，调用合适的方法驱动程序的执行。此外，对于 ACM 和 LeetCode 项目下的代码资源，虽然所有用例都包含正确的程序入口，但是往往需要合适的输入数据程序才能正确执行。获得这些用例的输入数据并不困难，只需要按照相应题目下的输入输出样子给予输入数据即可驱动程序执行，这些数据的获取可以通过爬虫程序取得，输入数据则可以通过重定向程序的输入流来完成。

空指针引用缺陷的产生必然需要一个产生 Null 值的缺陷源，在程序的某个位置，变量被赋值为 Null 随后沿着控制流图向前传播，在遇到解引用时便会触发空指针引用异常。这个变量便是缺陷源，只要在程序中的合适位置构造缺陷源，则有一定的可能会在程序的下文中触发空指针引用缺陷。当用例的可用性得到确认后，需要对用例程序进行语法分析，寻找合适的空指针产生点并构造缺陷源。

对 Java 文件进行语法分析可以使用 Eclipse JDT 下的 AST 来完成，该工具可以在 Eclipse 环境下获得，利用它能够对 Java 文件进行解析，生成相应的抽象语法树，并且能够任意修改 Java 代码的结构和内容。在 AST 中，Java 代码的每一个语法结构都有对应的 AST 结点表示，这些结点具有完整的层次结构，可以表示整个程序对象到具体方法的某个具体变量。如图4.2所示，一个 for 循环的代码片段按照 Eclipse AST 的标准解析出抽象语法树, 表4.1表示部分结点在 AST 树中对应的名称。

在生成用例的抽象语法树后，只要找到合适的点位，就可以通过修改合适的操作数为 Null 来构造空指针引用缺陷源。通常这些点位都和赋值表达式有关，但是在过程间调用的上下文中，方法的参数和返回值都可以是合适的构造点位。可以利用的修

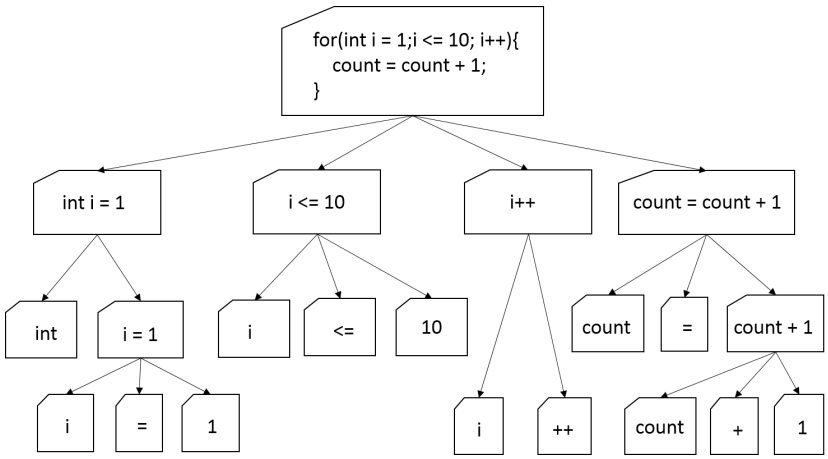

图 4.2　抽象语法树示例

表 4.1　AST 中的结点信息

子节点	子节点名	依附于父结点的角色
int i = 1	VariableDeclarationExpression	INITIALIZERS
i <= 10	InfixExpression	EXPRESSION
i++	PostExpression	UPDATERS
count = count + 1	Block	BODY

改位置有：

(1) 类的属性成员。

(2) 方法内的局部变量。

(3) 方法的参数。

(4) 方法的返回值。

其中（1）中的属性成员包含被初始化的非 Null 的普通属性和静态属性。（1）和（2）需要找到相关的赋值表达式，通过修改右操作数为 Null 来生成空指针引用缺陷源。（3）需要判断被调用方法的参数列表中属性的类型，将引用类型的实参修改为 Null 即可。（4）需要判断该方法的返回值类型，只有返回值为引用类型才可以修改。

图4.3为测试用例构建的流程图，在抽象语法树的基础上进行修改获得缺陷源后，需要对程序进行编译并执行才能确定能否真正构建出空指针引用缺陷用例，如果没有通过编译或者运行后没有发生空指针引用缺陷，则用例构造失败，需要重新寻找新的构建点位。重复此步骤直到成功产生空指针引用缺陷。最后，成功构建的缺陷用例需要在代码中添加代码信息的注解表明该缺陷的缺陷源和发生空指针解引用的位置。运用注解的方式是为了后续代码信息抽取的工作顺利进行，Java 程序可以很方便地抽

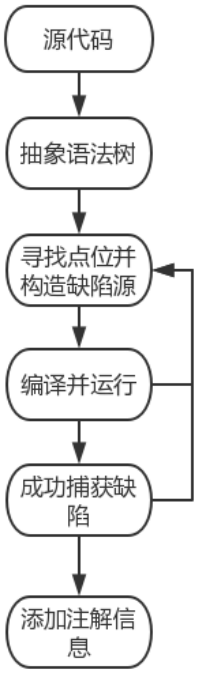

图 4.3　测试用例构建流程

取代码的注解信息，而这些自定义注解不会对代码的实际语义产生影响。记录空指针解引用的位置是为了验证工具检测的结果，加上缺陷源的位置可以很方便的确定该空指针引用缺陷发生的上下文范围，确定分析域。这些注解信息需要放置在测试用例的 Main 方法所在类中，方便代码信息抽取时统一处理，如果缺陷的产生位置不在 Main 方法所在类的方法中，就需要在注解信息中表明该缺陷产生位置所在的类。

下面的代码片段就展示了一个成功生成的测试用例，在代码的第 8 行将“sb = new StringBuilder(””)”修改为“sb = null”，随后在代码的第 11 行即对 sb 进行解引用操作，触发空指针引用缺陷。在第一行的注解标注了该缺陷涉及的上下文代码行号及变量名，这表示了分析域的范围。这个例子非常简单，实际上产生的代码在复杂度上是满足要求的，选用的原始代码往往都具备跨方法和跨文件的调用关系。

```
1  @Context(start = 8, end = 11, var = “sb”)
2  public static void main(String[] args) {
3      Scanner in = new Scanner(System.in);
4      int k = in.nextInt();
5      if(k > 36){
6          System.out.println("-1");
```

```
7         }else{
8             StringBuilder  sb = null;  //source
9             int  mul = k/2;
10            while(mul--> 0){
11                sb.append("8"); //npe
12            }
13            if(k%2 == 1){
14                sb.append("4");
15            }
16            System.out.println(sb.toString());
17        }
18    }
```

4.2 控制流图提取

控制流图（Control Flow Graph，CFG）是一个程序或者过程的抽象表现，代表了程序执行过程中所有可能经历的路径信息，能准确刻画一份代码的结构信息。空指针引用缺陷测试用例生成完毕后，需要生成代码的控制流图。

4.2.1 Soot

Soot【生存手册第 13 个引用】是一种 Java 字节码优化框架，凭借着对 Java 语言强大的分析能力，已经被广泛地应用于很多针对 Java 语言的分析优化项目。Soot 框架最大的特点就是它提供了四种不同的代码中间表示形式：Jimple，Baf，Grimp 和 Shimple，它们考虑到不同的分析场景，对代码进行了不同程度的抽象表示。同时，Soot 还构建了数据结构来表示待分析的项目，如表4.2所示。这些特有的表示方法使得代码的分析过程变得更加简单和灵活。Soot 的工作过程如图4.4所示。

表 4.2　Soot 中表示项目的数据结构

类名	描述
Scene	表示整个分析环境
SootClass	表示一个 class
SootMethod	表示一个 Method
SootField	表示一个类成员属性
Body	表示一个方法体

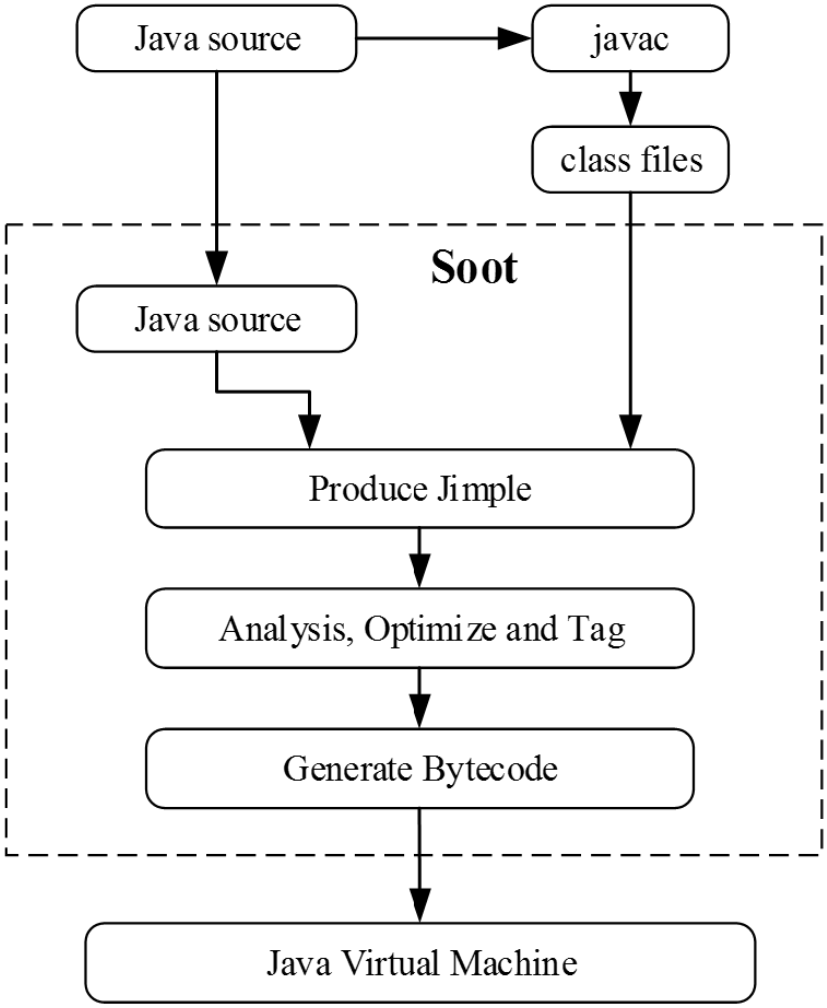

图 4.4　Soot 工作流程

Jimple 是 Soot 框架中最主要的代码中间表示形式，采用了典型的基于三地址的语句表达形式，它可以由 Java 源代码或者 Java 字节码转换得到。其中，通过字节码的转换通过为隐式的堆栈变量引入新的局部变量以及清除 jsr 指令来完成。在代码转换的过程中，Jimple 中的局部变量会被加上被推断出来的类型信息，同时部分没有用到的局部变量也会在优化阶段被清除掉从而不会出现在 Jimple 代码中，最重要的是代码中所有的表达式都会经过线性化的处理，最终 Jimple 将由若干三地址表达式组成，这些表达式至多包含三个变量或者常量信息，这种规则对于执行代码优化是非常方便的。另一方面，Jimple 表示形式所包含的语句类型只有 15 条，相比于 Java 字节码中多达 200 多条的指令类型，Jimple 的表示形式是足够简洁的，这些精简之后的指令类型也给本文后续的代码特征提取提供了极大的便利。

除了简洁准确的表示形式，利用 Soot 还可以进行复杂的别名分析和数据流分析。

另外，每个待分析方法的方法体，即表4.2中的 Body，都包含了一个 Units 链，Unit 是 Soot 中的一个接口，具体实现则是一个三地址表达式，这些表达式实际上会对应到 Jimple 中的某个具体语句类型，即 Stmt。这些 Unit 已经包含了互相之间的结构关系，根据这些结构关系可以很方便地生成方法内的控制流图。同时，对于包含调用方法语句类型的 Unit，通过 Soot 提供的 API 还可以很方便的得知该调用的对象方法，这对后面过程间调用图的构建非常重要。

4.2.2 全局控制流图构建

上一节介绍了 Soot，利用它可以构建出方法内的控制流图和过程间的调用图，进而得到全局控制流图。

方法内的控制流图可以从以 Jimple 表示的方法体中包含的 Units 链得到。通过遍历 Units 链中的 Unit 元素，可以得到它们的前驱和后继结点，这些信息实际上就表示了该方法的控制流图，不过它的结点是一个 Jimple 表示的 Stmt，在 Jimple 中表示了一种语句类型。如下面的代码片段所示，不同的语句对应着 Jimple 中不同的 Stmt 类型，这些类型有 15 种之多，是 Jimple 能够表示的最基本的语法单位。

```
1  public int foo(java.lang.String){// locals
2      r0 := @this; // IdentityStmt
3      r1 := @parameter0;
4      if r1 != null goto label0; // IfStmt
5      $i0 = r1.length(); // AssignStmt
6      r1.toUpperCase(); // InvokeStmt
7      return $i0; // ReturnStmt
8  label0: // createdbyPrinter
9      return2;
10 }
```

从该代码块中可以抽取出 foo 方法包含的 Units 链，进而得到其结构信息，如图4.5所示，结点的序号为代码块中语句对应的行号。从图中任意一个结点都可以找到其相邻的前驱和后继结点，这种关系不是静态代码顺序的关系，其考虑了代码执行逻辑的语序。

过程间调用图的构建需要依赖方法体内的方法调用语句，如上文代码块中的第 6 行语句，该 Stmt 的类型为 InvokeStmt。实际上 InvokeStmt 是一个接口，在 Jimple 包

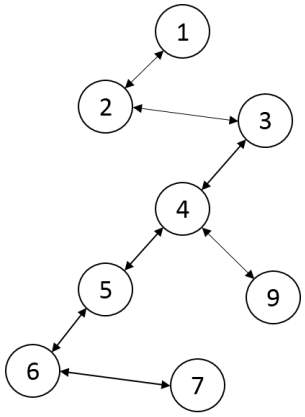

图 4.5　示例代码的 Units 关系图

含的 Stmt 中，实现 InvokeStmt 接口的类有五种，它们表示五种调用方法的类型：

(1)invokestatic：调用静态方法。

(2)invokespecial：调用实例的私有方法、父类方法或构造器方法。

(3)invokevirtual：调用实例中的虚方法。

(4)invokeinterface：调用接口方法。

(5)invokedynamic：调用动态方法。

其中，前四种调用方式可以由静态分析得到，而 invokedynamic 调用方式所调用的目标必须在运行时才能动态确定，所以本文只需关注前面四种调用方式，通过对调用目标的解析建立方法之间的调用关系，从而构建出完整的过程间调用图。

这里定义过程间调用图为G,G 包含两个数据集合 $Caller(U, S_m)$ 和 $Callee(M, S_u)$, 前者表示 Unit 结点与其调用的 Method 集合的映射关系，后者表示 Method 与调用它的 Unit 结点集合的映射关系。由于 Unit 和 Method 在 Soot 分析环境中具有唯一性，并且 Unit 和 Method 还包含了其结构的上下文关系信息，即利用 Unit 可以得到其所属的 Method 信息，甚至 Class 信息，所以不需要构建 Method 与 Method 之间的调用关系。利用这两个数据集合可以方便地找出某个 Unit 调用的 Method 集合 S_m 以及调用 Method 的 Unit 集合 S_u。在 Soot 开始加载待分析工程时，会将工程中的所有 Class 转换为 Soot 自定义的 SootClass，同时按照表4.2的对应关系，将每个 Class 中包含的子结构逐一封装成 Soot 中的相应类型对象。在 Soot 加载完毕后，逐一遍历 Scene 下所有的 SootClass，并分析其中的 Jimple 代码，在遇到调用语句解析其调用对象，即可构建 $Caller(U, S_m)$ 和 $Callee(M, S_u)$ 数据集合。具体构建过程如算法1描述。

以本节上面给出的代码片段为例，在对 foo 方法的 $Units$ 进行遍历时, 在第 6 行遇

Algorithm 1: 过程间调用图构建算法

Input: Soot 加载的 Class 集合 S_c
Output: 过程间调用图 G
$Caller \leftarrow \emptyset$
$Callee \leftarrow \emptyset$
foreach $c \in S_c$ **do**
  foreach $m \in c$ **do**
    foreach $u \in m.units$ **do**
      if $uinstanceofinvoke$ **then**
        $r = u$ 中执行调用的对象变量
        $mehodName = u$ 中被调用的方法的名字
      $className = backAnalysis(r)$
      if $u.invokeinstanceofinvokespecial$ **then**
        $invokeType = invokespecial$
      else if $u.invokeinstanceofinvokestatic$ **then**
        $invokeType = static$
      else if $u.invokeinstanceofinvokevirtual$ **then**
        $invokeType = invokevirual$
      else if $u.invokeinstanceofinvokeinterface$ **then**
        $invokeType = invokeinterface$
      $m' = getMethod(className, methodName, invokeType)$
      $Caller \leftarrow Caller \cup \{(u, m')\}$
      $Callee \leftarrow Callee \cup \{(m', u)\}$
$G \leftarrow \{\{Caller\}, \{Callee\}\}$
return G

到方法调用语句，可以得到其调用类型为 $virtualinvoke$。然后从该 $unit$ 中解析出被调用的对象 $r1$，这里无法得到 $r1$ 的类型，需要对 $r1$ 进行逆向分析，即 $backAnalysis$ 方法，直到解析到它是由该方法的第一个参数传递进来，从而得到它的类型为 $java.lang.String$。然后调用 $getMethod(className, methodName, invokeType)$ 方法，该方法传入的参数分别为 $r1$ 的类型的名称，被调用方法名字及调用方式，$getMethod$ 方法可以获得被调用方法在 Soot 环境下的实例。然后将 $unit$ 和 $Method$ 实例的映射信息加入到 $Caller$ 和 $Callee$ 这两个数据结构中。

本文需要将一个程序的结构刻画出来只用方法内的控制流图和过程间的调用图是不够的，为了形象地反映 Null 值在整个程序的传播路径，需要将空指针引用缺陷产生的上下文所涉及的方法构建成全局控制流图。在一个方法的调用语句处，将被调用方法的控制流图拼接进来，最终得到的全局控制流图将由若干个 unit 结点组成，结点之间的边表示不仅表示方法内语句的跳转关系，一些边还表示方法间的调用关系。用

一张图表示整个程序的控制流结构有利于反映直观地反映整个程序的复杂程度，也有利于代码特征的抽取和程序之间差异性的比较。

```
[21] public static void main(String[] args) {
[22]         ClassA varA = new ClassA();
[23]         Object object = null;
[24]         int varB = varA.foo();
[25]         if (varB == 0) {
[26]             object = new Object();
[27]         } else {
[28]             System.out.println("do nothing");
[29]         }
[30]         System.out.println(object.hashCode());
[31] }
```

```
[12] private int foo(){
[13]         return 1;
[14] }
```

上面的两个代码片段分别包含了 main 方法和 foo 方法的实现，在 main 方法的第 23 行处，*object* 对象被赋值为 null，在第 30 处发生了对 *object* 变量的解引用，此时必然会触发空指针引用缺陷。根据前文介绍，首先需要提取出分析域的范围，即第 23 行代码到第 30 行代码之间的控制流信息。同时，在 main 方法的第 24 行处存在对 foo 方法的调用，因此分析域应该包含 foo 方法内的控制流图。前文已经介绍了过程间调用图的构建方法，在过程间调用图的支持下，只需要在 main 方法的第 23 行处将 foo 方法内的控制流结构拼接起来就可以生成该分析域的全局控制流图，如图4.6所示。同理，即使方法的嵌套调用层级很多，也一样可以构建出分析域的全局控制流图。

4.3 代码特征抽取

分析域的全局控制流图可以表示空指针引用缺陷相关上下文代码的结构信息，但是代码不仅包含结构信息，还包含语义信息。本文构建的全局控制流图的结点为 Jimple 表示的 Unit，它表示了一个抽象的语法层面的语句类型，以 Unit 为基本单位，本文选取了 6 个维度的信息进行编码作为该结点的特征附加在控制流图结构上，这些维度分别如下：

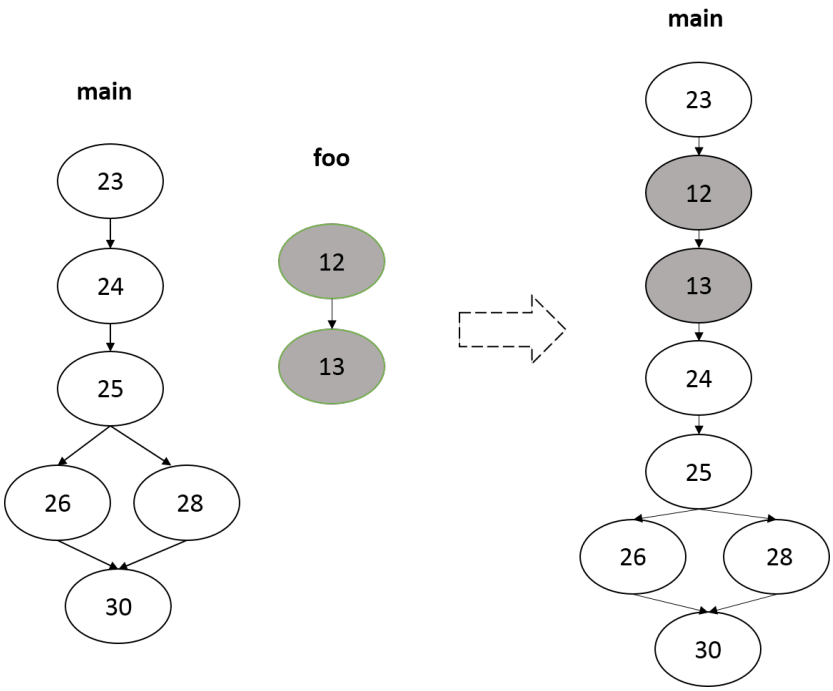

图 4.6　全局控制流图构建示例

1. 前驱结点数量

以上文构建的全局控制流图为基础，针对图中的每个结点获取其前驱结点的数量，作为该维度的特征值。

2. 语句类型

Unit 表示抽象的语句类型，实际上它包含了 15 种具体的 Stmt 语句类型，这些具体的语句类型和编码如表4.3所示。

3. 调用语句类型

调用语句表明该程序涉及到了跨过程的分析，在 Jimple 包含的 Stmt 中，实现 InvokeStmt 接口的类有五种，它们表示五种调用方法的类型，它们的编码4.4如表所示，如果该结点不包含方法调用语句，标记为 0。

4. 使用的操作数数量

该维度特征表示了指令对操作数使用的密集程度，过多的操作数使用往往也代表着算术指令的密集使用。

5. 空指针传递情况

如果当前节点涉及到了 Null 值的传递操作，可以说明下文中有更多可能会触发空指针引用异常。空指针的传递也有不同的方式需要编码，如表4.5所示，如果没有发

生空指针传递，编码为 0。

6. 空指针检查情况

空指针检查可以很好的避免下文中对空指针的引用，该语句通常会影响到工具检测结果的判定。如果该结点对 Null 值进行了检查，标记为 1，否则为 0;

通过六个维度的特征抽取，不仅可以体现全局控制流图的结构特征，而且可以体现图中每个结点即 Jimple 指令的语义特征。但是 Java 代码在转换为 Jimple 中间表示的过程中，为了使得到的语法单位都是标准的三地址表达式，将 Java 基本语句进行了大量的拆解重组，使得 Jimple 的基本语句的粒度比 Java 基本语句的粒度要小很多，这就使得生成的全局控制流图所能体现的特征更加不明显。为了让在模型训练的过程中更加容易的对代码分类，需要将生成的全局控制流图进一步压缩，使得图中每个结点接近基本块的维度。

4.4 数据标注

4.5 本章小结

表 4.3 Jimple 中 Stmt 语句类型及特征编码

语句类别	Stmt 名称	编码
核心指令	NopStmt	1
	IdentityStmt	2
	AssignStmt	3
方法内控制流指令	IfStmt	4
	GotoStt	5
	BreakPointStmt	6
	TableSwitchStmt	7
	LookUpSwitchStmt	8
方法间控制流指令	InvokeStmt	9
	ReturnStmt	10
	ReturnVoidStmt	11
监视器指令	EnterMonitorStmt	12
	ExitMonitorStmt	13
处理异常指令	ThrowStmt	14
	RetStmt	15

表 4.4　Jimple 中调用语句的类型及特征编码

调用指令	说明	编码
invokestatic	调用静态方法	1
invokespecial	调用私有方法、父类方法、构造方法	2
invokevirtual	调用抽象方法	3
invokeinterface	调用接口方法	4
invokedynamic	调用动态方法	5

表 4.5　Jimple 中空指针传递类型及特征编码

空指针传递指令	传递方式	编码
ReturnStmt	方法返回值	1
InvokeStmt	方法调用传参	2
definitionStmt	直接赋值	3

结论

（结论作为学位论文正文的最后部分单独排写，但不加章号。结论是对整个论文主要结果的总结。在结论中应明确指出本研究的创新点，对其应用前景和社会、经济价值等加以预测和评价，并指出今后进一步在本研究方向进行研究工作的展望与设想。结论部分的撰写应简明扼要，突出创新性。）本文采用……。（结论作为学位论文正文的最后部分单独排写，但不加章号。结论是对整个论文主要结果的总结。在结论中应明确指出本研究的创新点，对其应用前景和社会、经济价值等加以预测和评价，并指出今后进一步在本研究方向进行研究工作的展望与设想。结论部分的撰写应简明扼要，突出创新性。）

参考文献

[1] 姜敏, 彭少贤, 郦华兴. 形状记忆聚合物研究现状与发展 [J]. 现代塑料加工应用, 2005, 17 (2): 53–56.

附录 A　***

附录相关内容…

攻读学位期间发表论文与研究成果清单

[1] 高凌. 交联型与线形水性聚氨酯的形状记忆性能比较 [J]. 化工进展, 2006, 532 一 535.（核心期刊）

致谢

本论文的工作是在导师……。

作者简介

本人…。