计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2019,28(3):133−139 [doi: 10.15888/j.cnki.csa.006797]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

适用于分布式静态检测的 Java 代码依赖性分析技术①

韩承锋, 唐云善, 杨维永

(南京南瑞信息通信科技有限公司, 南京 210003)
通讯作者: 韩承锋, E-mail: kkwwqe87@vip.qq.com

摘　要: 为满足 Java 静态分布式检测系统对 Java 程序源代码解耦分包的需求, 解决代码检测单节点单进程运行耗时过长问题, 实现分布式检测系统单任务多节点并行运行的目的, 本文提出了 Java 源代码文件间依赖性分析方法. 该方法以生成源代码文件抽象语法树的方式抽取文件文本信息, 遍历分析抽象语法树, 获取文件与其他源代码文件类依赖关系, 再通过定位类所在的文件方式得到文件与文件之间依赖关系. 同时, 以无入边顶点的带环有向图表示文件间依赖关系图, 本文提出的方法基于该图进行了文件间解耦的分析. 最后, 通过对示例程序逐步剖析的实验以及对数个开源工具源代码解耦拆分的实验, 验证了本文提出的文件间依赖性分析方法的可行性.

关键词: 文件依赖性分析; Java 程序; JavaParser; 依赖关系图; 分布式静态检测

引用格式: 韩承锋,唐云善,杨维永.适用于分布式静态检测的 Java 代码依赖性分析技术.计算机系统应用,2019,28(3):133–139. http://www.c-s-a.org.cn/1003-3254/6797.html

Java Code Dependency Analysis Technique for Distributed Static Detection System

HAN Cheng-Feng, TANG Yun-Shan, YANG Wei-Yong

(NARI Group Corporation Information & Communication Technology Company, Nanjing 210003, China)

Abstract: In order to meet the requirements of Java static distributed detection system for decoupling and subcontracting Java program source code package, Java source code file dependency analysis method was proposed. This method is part of single-task multi-node parallel-running distributed detection system which could solve the problem that it takes long time for single-task single-node single-process code detection. The method extracted the Java source code file text information by generating its abstract syntax tree. Then, the method traverses and parses the abstract syntax tree to obtain the file's dependent classes which were not declared in the file. Finally, the dependency relationship between two files was obtained by locating the files where the dependent classes were declared in. The directed cyclic graph with no incident edge vertices was proposed to represent the dependency graph among files. The Java program source code package was decoupled by analyzing the dependency graph. At last, the feasibility of the proposed Java source code file dependency analysis method is verified by two experiments. The first experiment is the step-by-step result analysis of a sample program package. The second experiment is verifying the correctness of decoupling result of several open source tools' source code packages.

Key words: file dependency analysis; Java; JavaParser; denpendency graph; distributed static detection

源代码静态分析是软件安全漏洞检测的主要方法之一. 静态分析是指在不运行程序情况下, 对软件源代码进行扫描分析. 基于抽象语法树和程序控制流图等模型的数据流分析则是现阶段静态检测的主要使用技

① 基金项目: 国家电网公司科学技术项目 (调度自动化软件安全漏洞与风险实验能力提升关键技术研究)
Foundation item: Science and Technology Project of the State Grid Corporation (Research on Key Technologies of Automation Scheduling Software Security Vulnerabilities and Risk Experiment Capability Improvement)
收稿时间: 2018-08-31; 修改时间: 2018-09-26; 采用时间: 2018-10-08; csa 在线出版时间: 2019-02-22

术[1]. 使用静态检测技术实现的开源、商用代码检测工具已较为成熟, 如 PMD[2]、FindBugs[3]、Fortify SCA[4]等. 然而, 这些代码静态检测工具都是单机单进程运行, 实现对代码的检测. 由于静态分析需要遍历程序生成的中间语言, 所以随着程序代码规模的增大, 分析工具所需要的扫描时间也随之增加. 扫描单元级代码需要数十分钟甚至一小时, 扫描系统级代码需要高达数十小时, 这对项目开发进度产生较大的影响.

国家电网南瑞集团信通公司已开发了基于云计算的分布式代码检测系统 QMAP2.0, 该系统以检测节点动态扩展的方式实现了多任务同时在线检测, 但仍是单任务单节点运行, 依然存在当任务包过大时, 检测实现过长的问题. 为解决该问题, 在 QMAP2.0 系统基础之上, 基于依赖性分包的 Java 分布式静态检测系统被提出. 该系统提出了单任务多节点并行检测的方法, 对单任务进行程序代码分包, 再进行多节点并行分布式检测, 达到缩短程序检测时间的目的.

现阶段, Leo Pruijt, Christian Köppe 和 Sjaak Brinkkemper 提出了在模块层级上的直接结构依赖和间接结构依赖[5]; Judith A. Stafford, Debra J. Richardson 和 Alexander L. Wolf 提出了软件系统在结构关系和行为关系两种关系产生的体系结构层级上的依赖关系[6]; Xinyi Dong 和 Michael W. Godfrey 提出了一种能够在系统层级上高度抽象面向对象系统的模型——高级对象依赖关系图模型[7]; 陈树峰, 郑洪源等人从类层级上对面向对象软件依赖性进行了分析[8]. Java 分布式静态检测系统的需求是对软件的源代码包进行文件间的解耦, 文献[5–7]是从更为粗粒度的模块和系统出发, 分析整个程序的模块和体系的依赖性; 文献[8]则是从比文件更为细粒度的类层次, 对程序类间的依赖进行分析. Java 文件间的依赖分析主要是从源代码文件层面出发, 去分析文件之间的依赖关系, 与模块层级相比更为细粒度, 而较之类层级又稍显粗粒度; 同时, 需要深入研究基于文件间依赖性关系的程序包进行解耦拆分方法.

为了实现文件间解耦功能, 本文主要针对 Java 分布式静态检测系统的前端分析模块——依赖性分析模块进行研究, 该模块主要针对输入的程序代码包进行文件间的依赖性分析, 给出任务分包结果. 该模块分析结果的准确性是保证缩短检测时间却不影响检测结果的重要前提. 本文进行的主要工作如下: 1) 阐述依赖性分析模块的设计思路; 2) 基于对该模块设计思路, 进行了模块的开发实现; 3) 针对实现后的模块进行准确性实验.

1 背景知识

下面简要介绍本文所用的开源工具 JavaParser 以及源代码分布式静态检测系统相关背景知识.

1.1 JavaParser 简介

JavaParser[9]是一个 Github 上的开源项目, 该库能够让开发者在 Java 环境中以访客支持的方式使用抽象语法树 (Abstract syntax tree) 与 Java 源代码进行交互. JavaParser 把语法树抽象为 CompilationUnit 类, 每一个该类的实例化都记录了一个 Java 源代码文件的所有信息. 图 1 给出了一段简单代码在 JavaParser 工具解析后生成的抽象语法树形状 (图中树的底部的三角形记号表示这部分已经被总结). 如图 1 所示, CompilationUnit 对象主要有三大类子树: 包声明 (PackageDeclaration) 节点、导入声明 (ImportDeclaration) 节点、类\接口声明 (ClassOrInterfaceDeclaration) 节点, 类的主要内容则在类\接口声明节点的子树中.

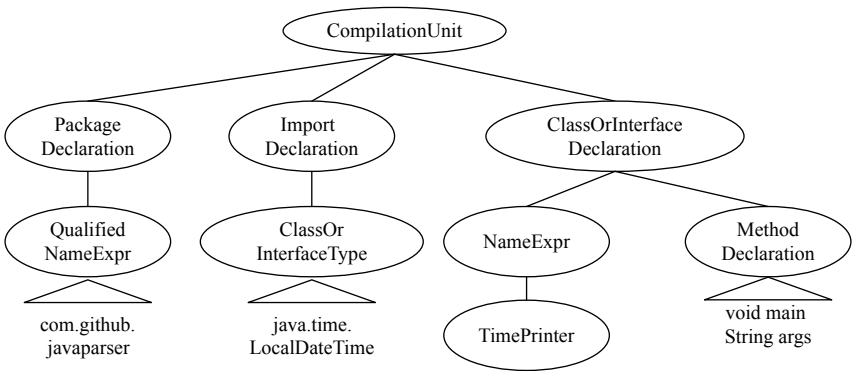

图 1 JavaParser 生成的抽象语法树

同时, 该项目还提供了一个 JavaSymbolSolver 库, 该库能够解析 JavaParser 生成的抽象语法树节点上各种符号的引用, 为开发者找到由该符号表示的变量声明, 参数声明或类型声明提供了便利.

1.2 源代码分布式静态检测系统简介

图 2 是基于依赖性分包的 Java 源代码分布式静态检测系统总体的工作流程图, 主要分为以下几个阶段.

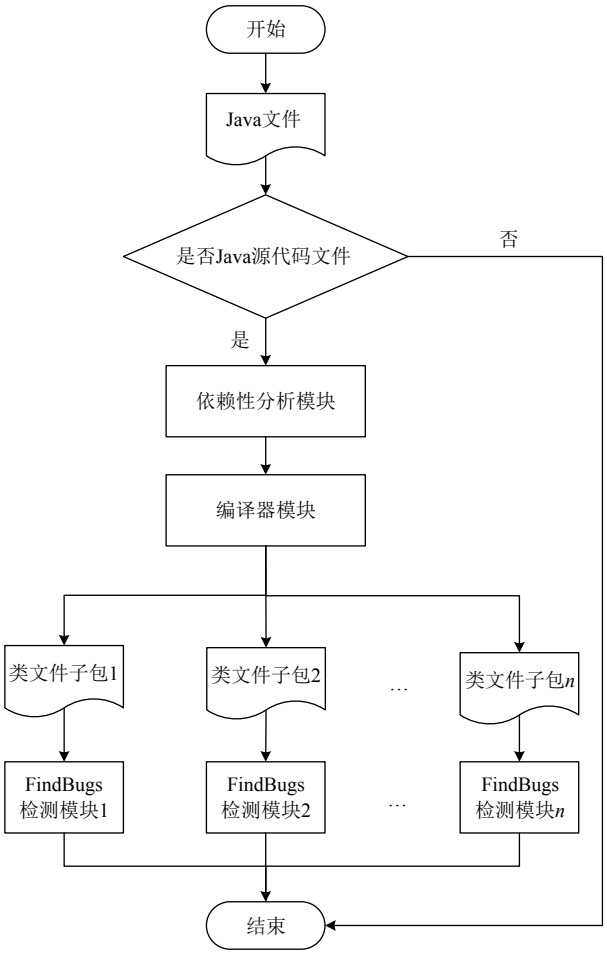

图 2 源代码分布式静态检测系统工作流程图

(1) 第一阶段, 系统读取程序文件, 若程序文件不是源代码文件则退出执行;

(2) 第二阶段, 文件依赖性分析. 首先针对输入程序代码包, 依赖性分析模块对源代码文件进行依赖性分析; 然后针对文件依赖性分析的结果进行分析, 得到互不依赖文件的集合; 再后根据系统要求, 合并互不依赖文件的集合, 得到若干个解耦后相互独立的文件集合;

(3) 第三阶段, 编译器模块对整个工程进行编译, 根据第二阶段的分析结果, 进行子包拆分;

(4) 第四阶段, Findbugs 检测模块获取到子任务包后, 对任务包进行检测;

(5) 第五阶段, 系统收到所有检测模块检测完毕信息后, 结束整个检测过程.

在整个系统中, 最重要的模块是 Java 源代码依赖性分析模块, 该模块生成的各源文件子包的大小决定了整个检测过程消耗的时间. 同时, 各源文件子包之间是否完全解耦也决定了分包前后的检测结果是否受到影响.

2 Java 源代码依赖性分析模块原理和实现

一个 Java 程序可以分为系统级、类层次级、方法级以及语句级. 依赖性分析与之相对应的分层是: 包间依赖、类间依赖、方法依赖、语句依赖[10]. 本系统依赖性分析模块主要进行源文件的文件间依赖进行分析.

2.1 依赖性分析模块原理

一个完整的 Java 源代码程序是由一个或数个以 Java 为后缀名的文本文件组成, 每一个文本文件至多属于一个包 (package), 一个包中可以含有一个或者多个文本文件. 在 Java 语言中, 每一个文本文件都是一个外部类 (public class), 该类包含一个或多个变量、方法与接口 (Interface), 接口与类功能类似, 但是接口只能实现 (Implenments), 不能被实例化.

依赖性分析模块主要分析 Java 源代码程序每一个.java 文本文件之间的依赖性, 所以必须从 Java 程序类间依赖出发, 再获取文件间的依赖关系. Java 程序的类层次依赖主要是由继承、接口实现、类创建、接口实例化、内部类、类的静态域引用与以及静态方法调用等引起的类之间依赖关系[8].

Java 语言由于追求程序设计的灵活性, 其存在着动态特性—一个方法在父类中被定义, 若在子孙类中被重新定义, 则方法在调用的时候回涉及到方法的多态调用以及动态绑定. 同时由于 Java 语言对对象类型的绑定时机属于晚绑定, 若要精确的分析类的依赖性, 需要获得 Java 程序的运行时信息[11]. 由于, 静态检测的目的就是不运行程序, 所以依赖分析模块的实现也是在不运行程序的情况下进行以依赖性分析, 此时对 Java 程序的多态特性进行分析, 就是对对象的所有可能类型进行静态确定, 这在 Java 语言上是能够实现的[11].

2.2 依赖性分析模块实现

图 3 是依赖性分析模块的工作流程图, 该模块的工作流程主要分为以下阶段.

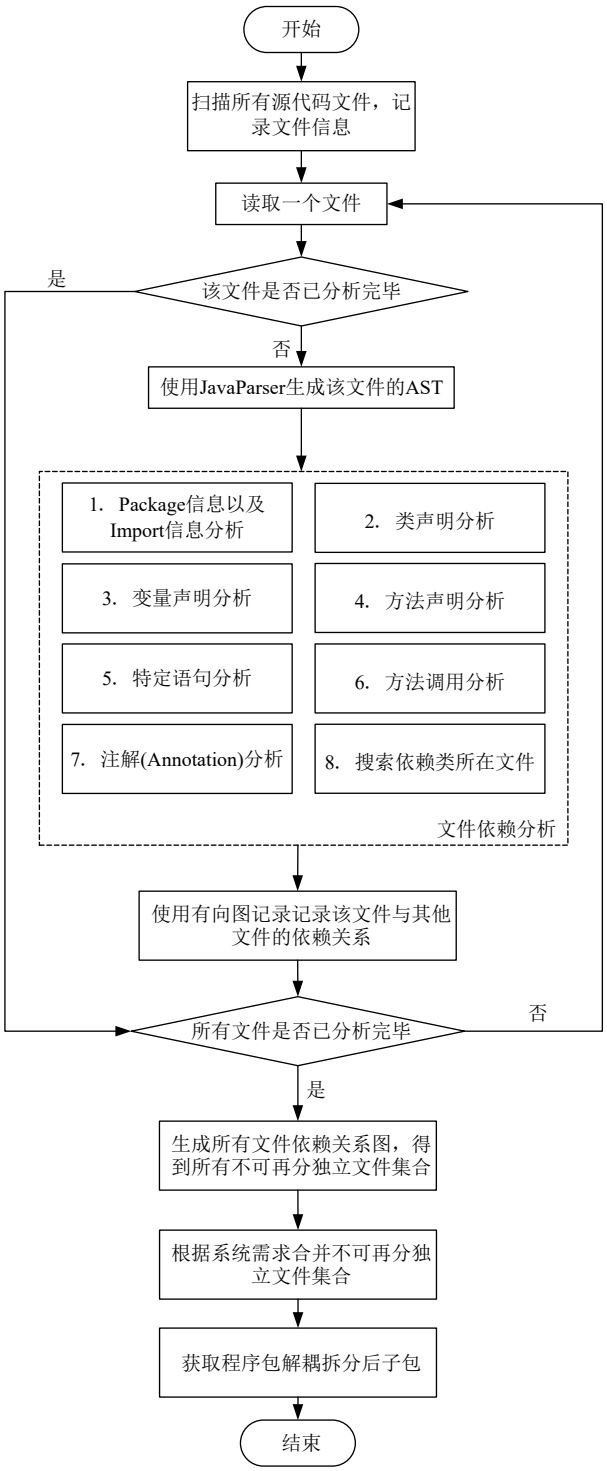

图 3 文件依赖分析模块流程图

(1) 第一阶段, 扫描该工程下所有文件并记录文件信息, 包括文件名、文件数量、文件是否已分析完毕等.

(2) 第二阶段, 选取一个文件, 若该文件并未进行依赖性分析, 则使用 JavaParser 库生成该文件的抽象语法树; 若该文件已经进行过依赖性分析, 则跳过该文件, 进行第五阶段检查.

(3) 第三阶段, 遍历生成的抽象语法树, 通过下面 8 个步骤对该文件对其他文件的依赖性进行分析:

1) 第一步, 分析本文件所在的包以及 Import 产生的包依赖, 在接下来的步骤中直接遍历依赖的包中的类以及 Import 依赖的类, 缩小遍历范围. 同时, 记录 Import Static 引入的静态变量;

2) 第二步, 类声明分析. 针对该文件的 public 类以及类的内部类进行分析, 若这些类存在父类或者实现的接口类, 则记录下来;

3) 第三步, 变量声明分析. 分析所有变量声明语句, 记录所有是类对象变量所属的类; 若这些变量语句是初始化语句, 同时检查初始化的值是否是类的静态成员变量, 是则记录下该成员变量所属的类;

4) 第四步, 方法声明分析. 包括类的构造方法以及普通方法. 分析方法的返回类型、入参类型以及抛出类型, 记录这三处地方存在的类;

5) 第五步, 特定语句分析. 针对可能存在类符号的语句进行分析, 例如赋值语句, 类对象创建语句, for 的条件语句, catch 的条件语句等. 遍历所感兴趣的语句, 记录语句中存在的类或者类的静态成员;

6) 第六步, 方法调用分析. 首先, 分析调用方法的输入参数, 若参数类的静态成员变量, 则记录下该变量所属的类; 然后, 分析该方法是否是静态成员方法, 若是则记录下该成员方法所属的类; 最后, 若该方法不是静态成员方法, 查询该方法是否属于该文件 public 类或者内部类, 若不是则查询并记录下该方法所属的外部类.

7) 第七步, Java 注解 (Annotation) 分析. 分析文件中注解, 若存在自定义注解, 则把该注解类记录下来.

8) 第八步, 根据前 7 步所获取的该文件依赖的类, 定位这些类所在的文件, 得到的文件为该文件所依赖的文件.

(4) 第四阶段, 根据第三阶段所得结果, 生成表示该文件对其他文件依赖的有向图.

(5) 第五阶段, 检查所有文件是否已经分析完毕. 若分析完毕, 则合并第四阶段所得的每个文件的依赖有向图, 获得所有文件之间依赖关系的有向图; 若尚未分析完毕, 则跳回第二阶段进行迭代.

(6) 第六阶段, 对所有文件之间依赖关系的有向图

进行分析, 获取不可再分的文件集合; 然后, 根据系统需求 (比如拆分后文件包大小等), 合并不可再分文件集合, 得到若干个相互独立的文件集合, 这些相互独立文件集合为解耦拆分子包的文件集合.

2.3 依赖性分析模块实现

程序源文件的分包结果对检测的准确性和检测时间有较大影响, 所以依赖性分析模块除了对文件依赖性进行分析之外, 如何根据生成的文件依赖关系图进行对源程序进行分包也是该模块的一个重要步骤.

图 4 是一个程序源文件依赖图示例. 图中箭头指向表示依赖关系, 例如图中 A→G 表示表示 A 文件依赖于 G 文件, 文件 D、G、E、F 之间形成的依赖环称为文件集. 在处理图过程中, 所有的文件集被简化为一个独立节点对外进行分析.

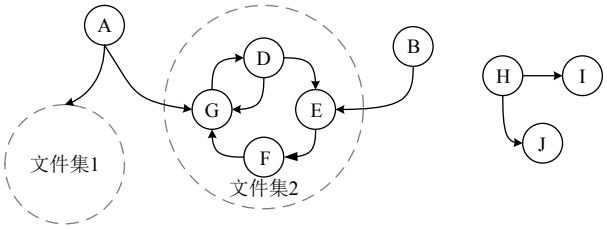

图 4 源文件依赖关系图

针对文件集对关系图简化后, 此时所有节点之间都是单向依赖关系. 由文件间依赖性关系的特点可知, 简化后的关系图必然存在一个没有入边的顶点. 所以, 在对文件依赖关系有向图进行简化之后, 首先寻找图中所有无入边顶点, 如图 4 中的 A、B、H 节点, 以这些节点作为遍历依赖图的起始节点. 同时, 以所有没有出边的节点为终结节点, 遍历整个依赖图, 得到的所有节点集合则为最小不可分的依赖文件集合. 如图 4 中, 以 A 为起始点, 遍历整个依赖图, 得到所有终结点文件集 1 和文件集 2, 可到一个不可再分文件集合 a(A, 文件集 1, 文件集 2); 同理, 以 B 为起始点可得到不可再分集合 b(B、文件集 2), 以 H 为起始点可得到不可再分文件集合 c(H、I、J).

得到所有不可再分集合之后, 根据系统实际要求的分包大小, 把不可分集合进行合并. 具体合并规则为: 1) 有交集的两个集合先合并; 2) 多个集合同时存在交集, 交集大的先合并.

如图 4 所得到的 3 个不可再分集合, 若两两相合并之后的大小都满足系统要求, 因为集合 a 和集合 b 存在交集文件集 2, 所以在合并过程中应该先合并集合 a 和集合 b, 得到独立文件集合 SetA, 若独立文件集合 SetA 和不可再分文件集合 c 合并后大小超出系统要求, 则不可再分文件集合 c 则被视为独立文件集合 SetB. 把不可再分集合合并之后得到若干独立文件集合就是最终程序分包的实际文件集合, 如图 4 所得 SetA 和 SetB 就是最终所得的两个程序子包的文件集合.

3 依赖分析模块准确性实验

为了评估依赖分析模块的准确性, 本文通过两个步骤进行验证: 1) 针对自行编写的简单示例程序, 验证依赖分析 8 个步骤分析结果的准确性; 2) 选择了三个 Java 开源项目作为实验对象, 验证模块是否存在普遍准确性.

3.1 实验环境

实验软件环境: JDK8. 实验对象为自行编写的简单示例软件、Apache Commos 项目下 26 个独立组件、JfreeChart 开源软件以及 Findgbus 检测工具. 实验对象基本情况如表 1 所示.

表 1 实验软件基本情况

实验软件	源代码规模	类文件个数
简单示例程序	2.08 KB	7
Apache Commons Components	31.78 MB	3910
JfreeChart	7.68 MB	629
Findbugs	7.24 MB	1116

3.2 验证模块单步准确性

如表 1 所示, 实验使用的的示例程序共有 7 个类文件, 其中代码规模和文件关系最复杂的就是 A.java 文件, 所以本次实验主要针对依赖关系最复杂的 A 文件进行分析, 验证模块单步的准确性.

图 5 是主要表示了 A.java 文件与其他源代码文件的依赖关系图. 图中每一个小圆圈代表着一个源代码文件; 虚线框表示源代码文件所处的包, 如 A.java 文件是处在 com.dependencyfiles.package1 包中; 与图 4 类似, 箭头表示着文件的依赖关系.

如表 2 所示, 每一个步骤都节选 A.java 文件中部分该步骤分析的语句, 并给出相应结果, 每步分析结果具体过程如下:

(1) 在第 1 步分析时, 模块首先分析 package 语句, 可得到文件属于 com.dependencyfiles.package1 包, 同时通过扫描的方式获取包下其他文件; 然后, 模块分析

import 语句, 如表 2 中所示, 得到文件可能依赖的类为 com.dependencyfiles 包下 A3 类的类.

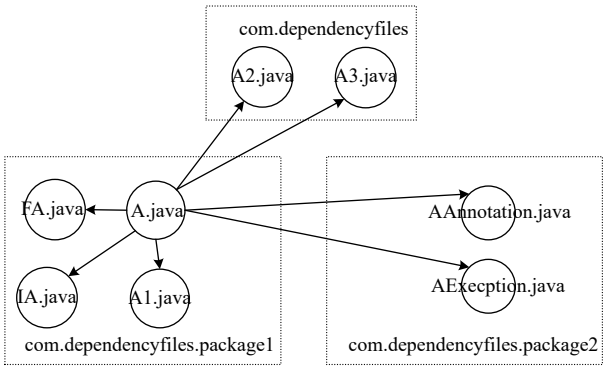

图 5 示例程序的依赖关系图

(2) 在第 2 步分析时, 模块需要分析所有类声明的语句, 如表 2 示例语句所示, 获取到该文件外部类 A, 有父类 FA, 实现了接口 IA.

(3) 在第 3 步分析时, 模块需要对类和类成员方法中变量声明语句进行分析. 如表 2 示例语句所示, 该语句是带初始化的变量声明语句, 模块分析得到结果为: 变量是属于 List 类, List 泛型为 String, 变量初始化为 ArrayList 类.

(4) 在第 4 步分析时, 模块针对方法声明的返回类部分、入参部分以及抛出错误部分进行分析. 如表 2 示例语句所示, 分析该方法声明语句块可得到: 返回类型属于 A1 类; 形式参数属于类 Map, Map 的泛型为 String; 抛出错误类型为 AException 类.

表 2 模块对 A.java 文件的部分单步分析结果

分析步骤	待分析文本 (部分)	分析结果
第 1 步	package com.dependencyfiles.package1; import com.dependencyfiles.A3;	所属包 com.dependencyfiles.package1 可能依赖于 com.dependencyfiles 包下 A3 类
第 2 步	public class A extends FA implements IA	实现接口 IA, 父类 FA
第 3 步	List<String> list = new ArrayList<>();	带初始化的变量声明语句, 依赖类 List、String、ArrayList
第 4 步	public A1 aMethod1(Map<String,String> map) throws AException { return new A1();}	方法声明返回类 A1, 形参 Map、String, 抛出类 AException
第 5 步	for (Set<String> set : setArrayList) {} try {} catch(AException e) {}	For 语句条件: Set 类、String 类 Catch 条件: AException 类
第 6 步	a2.a2Method(new A3());	方法调用语句, 调用方法属于 a2 所属类 A2, 参数为 A3 类
第 7 步	@AAnnotation	注解: AAnnotation 类
第 8 步	定位前 7 步获取的依赖类所在文件	A3 类属于 A3.java 文件; 接口 IA 属于 IA.java 文件; 父类 FA 属于 FA.java 文件; A1 类属于 A1.java 文件; AException 类属于 AException.java 文件; AAnnotation 属于 AAnnotation.java 文件

(5) 在第 5 步分析时, 模块针对关心的兴趣点语句进行分析, 如对象创建语句、返回语句、for 循环的条件语句、catch 块的条件语句等. 如表 2 中示例语句所示, 模块分析可得 for 语句块的条件变量属于 Set 类, Set 的泛型属于 String 类; 同理, try 语句块的 catch 条件属于类 AException 类.

(6) 在第 6 步分析时, 模块针对所有的方法调用进行分析. 如表 2 中示例语句所示, 模块通过分析可得, 调用方法属于变量 a2, 由上下文查找得到变量 a2 的初始化语句, 分析得属于类 A2; 同时, 方法调用的入参为类型 A3 类.

(7) 在第 7 步分析时, 模块针对注解进行分析. 如表 2 中示例语句所示, 模块通过分析可得注解属于 AAnnotation 类.

(8) 在第 8 步分析时, 模块对前 7 步所获取到的依赖类进行定位, 确定 A.java 文件所依赖的源代码文件. 如表 2 中示例所示, 第 2 步分析所得类 FA 属于 FA.java 文件, 接口 IA 属于 IA.java 文件; 第 3 步分析所得类 List、String 以及 ArrayList 类均不存在于示例源代码文件中, 所以这些类不产生文件间依赖关系; 第 4 步分析中, 方法声明中返回类型 A1 属于 A1.java 文件, 抛出错误类型 AException 属于 AException.java 文件; 在第 6 步分析时, 方法调用的入参 A3 类属于 A3.java 文件; 第 7 步分析时, 注解 AAnnotation 类属于 AAnnotation.java 文件. 至此, A.java 文件所依赖的其他源代码文件被找出.

通过对每一步分析过程的的展示, 在示例程序上, 依赖分析的最终结果与图 5 实际结果相同, 模块的分析结果的准确性达到了实验的预期.

3.3 验证模块普遍准确性

依赖分析模块分析结果主要是通过使用编译器对分包后的源文件集进行编译的方法进行准确性验证. 因为编译器在对源文件进行编译时, 需要对源文件进行语义检查, 若模块分析模块对程序进行分包后得到的子包有文件缺失时, 编译器的语义检查则得不到通过, 会导致编译无法完成. 因此, 以把程序分包后得到的不同子包进行编译的方式来验证模块分包的准确性是可行的.

因为依赖分析模块是先获取所有不可分依赖集之后再进行合并, 每个不可分依赖集都是独立的, 所以验证分析结果的准确性时, 只需要对所有的不可分依赖集进行验证即可.

经过依赖性分析模块分析之后, Apache Commons 组件被拆分为 906 个不可分集合, JfreeChar 工具被拆分为 143 个不可分集、Findbugs 工具被拆分为为有 290 个不可分集. 所有这些不可分集合在使用 JavaC 编译器进行编译时都能顺利编译通过, 验证了依赖性分析模块是可行性的.

4 结语

本文为满足 Java 源代码分布式静态检测系统需求, 提出了一种适用于该系统的 Java 源代码包文件间依赖分析技术. 该技术在文献[8]的类依赖关系分析基础上, 使用了类定位文件的方法, 完成了源代码文件间依赖性的分析; 同时, 提出了用于表达文件间依赖关系的有向图, 并基于该图提出了程序包解耦拆分的方法. 最后, 本文基于 Javaparser 开源工具, 设计和实现了该系统的 Java 源文件依赖分析模块, 并针对该模块的准确性进行了验证. 实验验证结果证明, 该模块可以较好的适用于 Java 源文件程序包的文件间依赖性分析和程序文件间解耦分包, 能够适用于类似代码静态检测等程序源代码文件解耦拆分的场景.

参考文献

1 吴世忠, 郭涛, 董国伟, 等. 软件漏洞分析技术进展. 清华大学学报 (自然科学版), 2012, 52(10): 1309–1319. [doi: 10.16511/j.cnki.qhdxxb.2012.10.001]

2 PMD Open Source Project. PMD. https://pmd.github.io/. [2018-03-29]

3 Hovemeyer D, Ayewah N, *et al*. FindBugs. http://findbugs.sourceforge.net/index.html. [2018-03-29]

4 Micro Focus Company. Fortify static code analyzer. https://software.microfocus.com/en-us/products/static-code-analysis-sast/overview. [2018-03-29]

5 Pruijt L, Köppe C, Brinkkemper S. On the accuracy of architecture compliance checking support accuracy of dependency analysis and violation reporting. Proceedings of the 21st International Conference on Program Comprehension. San Francisco, CA, USA. 2013. 172–181. [doi: 10.1109/ICPC.2013.6613845]

6 Stafford JA, Wolf AL. Architecture-level dependence analysis for software systems. International Journal of Software Engineering and Knowledge Engineering, 2001, 11(4): 431–451. [doi: 10.1142/S021819400100061X]

7 Dong XY, Godfrey MW. System-level usage dependency analysis of object-oriented systems. Proceedings of 2007 IEEE International Conference on Software Maintenance. Paris, France. 2007. 375–384. [doi: 10.1109/ICSM.2007.4362650]

8 陈树峰, 郑洪源. 面向对象软件的依赖性分析与回归测试. 计算机应用, 2009, 29(11): 3110–3113.

9 Nicholas Smith, Danny van Bruggen and Federico Tomassetti. JavaParser: Visited. https://leanpub.com/javaparservisited. [2018-03-29]

10 余斌. Java 程序分层及概率依赖性分析[硕士学位论文].南京: 东南大学, 2006.

11 陈振强. 基于依赖性分析的程序切片技术研究[博士学位论文]. 南京: 东南大学, 2003.