硕士学位论文

基于 Java 字节码的多线程数据竞争检测方法研究及工具实现

DATA RACE DETECTION FOR JAVA BYTECODE IN MULTITHREAD PROGRAMS AND TOOL IMPLEMENTATION

史鹏宙

哈尔滨工业大学

2014 年 12 月

国内图书分类号：TP311.1　　学校代码：10213
国际图书分类号：004.4　　密级：公开

工程硕士学位论文

基于 Java 字节码的多线程数据竞争检测方法研究及工具实现

硕 士 研 究 生：史鹏宙
导　　师：黄荷姣
申 请 学 位：工程硕士
学　　科：计算机技术
所 在 单 位：深圳研究生院
答 辩 日 期：2014 年 12 月
授予学位单位：哈尔滨工业大学

Classified Index: TP311.1
U.D.C: 004.4

Dissertation for the Master's Degree of Engineering

DATA RACE DETECTION FOR JAVA BYTECODE IN MULTITHREAD PROGRAMS AND TOOL IMPLEMENTATION

Candidate： Pengzhou Shi
Supervisor： Prof. Hejiao Huang
Academic Degree Applied for： Master's Degree of Engineering
Speciality： Computer Technology
Affiliation： Shenzhen Graduate School
Date of Defence： December, 2014
Degree-Conferring-Institution： Harbin Institute of Technology

摘 要

随着计算机系统规模逐步扩大，用户数量不断增加，越来越多的软件系统需要考虑并发程序设计，而多线程技术是实现软件程序并发的常用方法。但是，由于多线程程序开发难度大，程序运行期间线程调度的不确定性，使得多线程程序很容易发生问题。数据竞争就是多线程程序中一种常见的并发问题。在多线程环境下的数据竞争是指多个线程同时访问同一块内存资源时，没有采取正确的同步方式，而且其中至少存在一个写操作。数据竞争问题会给软件程序带来很大的隐患，可能造成很大的危害。为了解决数据竞争问题，国内外研究者采用形式化的方法进行数据竞争检测。通常的数据竞争检测方法分为静态检测和动态检测两种。然而，静态检测方法存在错误预报，动态检测方法执行效率较低，而且不能覆盖全部的程序执行路径。本文针对 Java 多线程程序，提出了一种基于 Java 字节码的数据竞争检测方法，主要完成了以下工作：

通过字节码解析，建立了抽象程序模型，并提取出与程序分析相关的关键信息集合。在字节码指令的基础上，采用数据流分析方法，提出了基于字节码指令的控制流程图和函数调用图的构建方法。

根据数据竞争产生的条件，总结出基于字节码的检测数据竞争方法。通过使用别名分析方法，建立数据竞争检测步骤，即可达性计算、别名分析和锁集合计算，有效地发现数据竞争问题。

为验证方法的有效性，使用编程语言实现了基于字节码的数据竞争检测方法。通过实验证明，相比基于源代码的数据竞争检测方法，该方法具有较高的检测效率，并且能够准确地发现数据竞争问题。

关键词：程序分析；数据竞争；多线程；静态检测；字节码

Abstract

With the gradual expansion of the size of computer systems, the number of users is increasing continuously, and many software systems need to consider the concurrency. Nowadays, the multithreading technology is a common way to realize the concurrency. However, the multithreaded application developoment is difficult, and the programs run randomly during the thread switching. It is more prone to have problems. Data race is a common concurrency problem in multithreaded programming. The data race means many threads access a shared memory resource at the same time without ordering constraints enforced between them, and at least one access is a write. Data race problem can bring a lot of bugs in software, and sometimes it will be a huge disaster. In order to solve the data race problem, many researchers use the formal method to detect the data race problems. Data race detection is classified as either static or dynamic. Static race detectors have false results, while dynamic race detectors are running slowly and can't detect all paths. In this thesis, a method of detecting data race in Java bytecodes is proposed. The main contributions of my work are summarized as follows:

By analyzing the bytecode files, the abstract model of the program is built, and some key information about the program analysis is extracted. On the basis of bytecode instructions analyzing, it is used the data flow analysis method to propose a way of building the control flow graph and the call graph about the bytecode instructions.

According to the conditions of the data race problem, it is summarized a way to detect data race problem based on bytecode instructions. By using alias analysis method, it is found three steps in the process, and they are reachable analysis, alias analysis and lockset analysis. By using these steps, an effective data race detection method is built.

To verify the validity of the method, the method which is named data race detection based on bytecode is implemented by using a programming language. Experiments show that the tool runs efficiently compared with the method based on the source code. And it can find the data race problem correctly.

Keywords: program analysis, data race, multi-threading, static analysis, bytecode

目　录

第 1 章 绪　论

1.1 研究背景

计算机硬件多核技术的迅速发展，使得多核处理器成为主流。同时随着移动互联网和云计算技术的发展,越来越多的用户开始使用计算机软件系统。为充分地利用多核硬件资源，并且满足软件系统的要求，软件设计人员将并发程序设计作为提高系统性能和满足用户需求的关键技术[1]。

在复杂软件系统或大规模计算机系统中并发需求已经成为设计者考虑的基本因素，例如在分布式系统[2]、云计算服务[3]、文件存储系统等各类型软件系统中都需要考虑程序并发问题。各行各业的专业软件中越来越多地使用到并发程序技术。但是由于软件编程语言在设计初期并没有考虑并发特征，使得不同的编程语言实现并发的方法各不相同，再加上并发程序本身比较复杂，因此开发符合系统要求并且保证正确运行的并发程序的难度很大。

为保障软件系统正确运行，在软件工程中，通常使用软件测试的方法来保障软件质量。但是对于并发程序而言，在执行过程中存在并发调度的不确定性，粗粒度的测试用例并不能够覆盖程序运行的全部状态，从而可能会带来隐藏的软件漏洞。为快速有效地保障软件质量，研究人员通过采用形式化方法[4]研究并发程序中的问题，编写自动化检测工具来检测软件缺陷。

在众多的并发程序的缺陷中，数据竞争是最常见的问题。所谓数据竞争，是指多个并发执行单元在不正确的同步方式下同时访问同一块内存数据，并且其中某个访问操作为写操作。数据竞争的存在使得并发程序容易出错，而且会使软件程序表现出违反预期的行为。如果发生在已经交付用户使用的软件产品中，可能引发灾难性事故，例如 2012 年纳斯达克 Facebook 股票交易问题[5]。在上述事件中导致故障发生的根源是隐藏在并发程序中的数据竞争。

1.2 研究目的及意义

在并发程序设计中，多线程技术是最常见的方式[6]。多线程程序通过采用多个程序运行实体提高了系统的资源利用率，提升了应用程序的性能，并且带来了较好的用户体验。在客户端程序中，多线程技术使得程序能够在响应图形用户界面的同时运行完成数据处理任务，缩短了运行时间；在服务器程序中，多线程技术可以将多个用户请求同时处理，提高系统效率，加快了

处理请求速度。然而，由于多个线程执行期间代码的执行顺序不确定，使得多线程程序在开发过程中容易产生死锁、数据竞争等问题。因此并发程序的开发和缺陷检测成为软件工程的挑战[7]。

为了解决多线程程序的数据竞争问题，开发人员在代码完成后使用形式化检测工具来检测程序，记录程序的执行过程，提前发现存在问题的代码块，从而消除数据竞争问题。为此，各种检验数据竞争的形式化方法也应运而生。

采用形式化方法分析数据竞争问题的方式有很多，包括模型检测、数据流分析等。常见的数据竞争检测思路为：在程序源代码的基础上，根据数据竞争发生的必要条件，使用形式化方法建立分析模型；通过对程序中数据访问的语句进行分析，或是对程序运行流程分析，找出可能发生问题的位置。实验证明，形式化方法可以有效地解决数据竞争问题，并且提高并发程序的健壮性。

为研究问题的方便，所有的数据竞争检测研究必须针对某种编程环境或某种应用场景，有些研究者还针对特定的程序，如操作系统内核[8]、中断处理程序[9]等。本课题主要的研究对象为 Java 环境下多线程并发程序中数据竞争检测。课题通过对数据竞争检测方法的研究，全面总结数据竞争检测的形式化方法，掌握形式化方法的思路和解决方案，合理评估各种形式化方法的优缺点。通过本课题的研究探索多线程程序下数据竞争问题的解决方法，并且对检测方法的有效性做出验证。因此，本课题的研究对于实际工程应用具有一定的意义。

1.3 国内外研究现状

1.3.1 国外研究现状

数据竞争问题在国外研究较早，近几年成为研究热点之一，提出很多的检测方法，并针对不同应用场景和语言平台实现了自动化的检测工具[4,8-10]。国外研究者的早期研究集中在动态检测方法上，然而近几年来静态检测成为主要研究内容。目前使用的检测方法较多，重要研究成果如下所述。

Engler 和 Ashcraft 提出一种基于数据流分析的静态检测工具 RacerX[10]。该工具主要针对 Linux、FreeBSD 等操作系统代码进行分析。通过对程序的分析找出多线程执行的代码块，并分析上下文找出保护共享变量的锁操作，定位出对共享变量的危险访问操作。最后，按照程序控制流图中的路径对可能出现数据竞争的次数进行统计，出现次数较多的操作被认为是较危险的行

为。采用数据流分析的方法具有普通静态方法的缺点，容易出现误报。

Flanagan 和 Freund 通过类型推导理论提出一种静态检测的方法[11]。在分析每个程序中共享变量的同步操作后，建立推导规则，用来检测被访问的共享变量是否符合检查规则。但该方法需要在源代码中合适的位置添加正确格式的注释，并且注释内容会影响检测结果，增加了程序开发的难度。

Elmas 等基于模型检测理论提出一种检测方法[12]。该方法通过对程序中锁操作路径的分析，捕捉所有互斥同步操作，并且通过使用 happens-before 关系过滤结果。Kahlon 等结合模型检测和别名分析方法，将检测过程分为三个步骤：首先在数据流分析中发现共享变量；然后在访问共享变量时通过别名分析检查所拥有资源锁；最后分析操作记录并过滤错误预报[13]。

Ferrara 利用抽象解释理论分析 Java 程序语句，实现了静态检测工具 Checkmate [14-15]。该工具支持了 Java 多线程的大部分相关特性，如线程的动态创建、内存的动态分配、运行期间 monitor 的创建等。工具从给定类的 main 方法开始计算程序的近似语义，分析指令流程并建立函数内的控制流程图，最终完成对给定属性的检测。

Naik 和 Aiken 应用别名分析技术来判断两个资源锁对象是否不同，从而实现对数据竞争的检测[16]。若两个锁对象不指向相同的 monitor，那么对于共享变量的访问操作也一定不能指向同一块内存，从而满足了 race freedom 的条件。但是，该方法由于需要对每两个共享变量的访问操作进行判断，从而使得方法执行时间较长，算法效率较低。

上述的研究方法都是静态检测方法，而动态检测方法主要分为三种：基于发生序关系（happens-before 关系）、锁集合（lockset）算法和混合发生序和锁集合的方法。此外，还有文献[8]中使用的基于硬件断点的动态检测方法。

Flanagan 和 Freund 实现了一种基于发生序关系的动态检测工具[17]。该工具使用一种轻量级的表示方法来取代发生序关系中的向量锁机制，使得算法在运行期间只需要常量级别的内存开销和时间花费。该方法与实现了锁集合算法的 Eraser[18]工具相比，减少了错误预报，提高了算法效率，是目前最快的纯发生序的动态检测方法。但是，该方法对线程交替顺序敏感，会产生较多问题。

Yuan 等在混合发生序算法和锁集合算法的基础上，通过对运行期间访问变量和线程的锁集合的跟踪，动态地检测数据竞争问题[19]。该方法使用发生序关系分析每个变量在当前时刻的并发访问次数，并动态判断当前变量的访问操作是否满足并发访问条件。若发现对某个变量的操作不满足并发访问的

条件，则判断为发生数据竞争问题。但是该方法需要反复多次执行程序来检测数据竞争问题，因此运行效率较低。

Xie 等结合锁集合算法和改进后的发生序算法实现了一种新的动态检测工具 ACCULOCK[20]。该工具通过观测运行过程中加锁和解锁的发生序关系集合，减弱了对线程交替的敏感度。通过实验发现，该工具能够检测出比文献[19]更多的数据竞争问题，而且比锁集合算法误报率低。

文献[21]建立了应用程序中数据竞争缺陷库，从而以此来判断程序中是否存在数据竞争问题。然而该方法依赖于特定的测试用例库，需要经验丰富的测试人员来编写测试脚本。

1.3.2 国内研究现状

与国外研究相比，国内研究大多集中在已有的检测方法上。通过对已有方法进行改进和优化，取得了较多的成绩，主要的研究成果如下所述。

吴萍等在 JTool 编译器上实现了一个精确有效的静态检测框架。该框架的分析算法把问题分解为跨线程的控制流分析、以全局变量为中心的访问事件获取以及时序关系的约束求解[22]。分析过程中使用了上下文敏感分析和别名分析技术，并结合 Escape 分析有效地缩小了检测范围，提高了准确率。但是，该方法需要建立全局的分析信息，只能对规模较小的程序进行检测。

简道红在 happens-before 关系和 lockset 算法的基础上实现了一种基于抽象语法树的识别多线程程序的静态检测方法[23]。通过词法分析和语法分析等步骤实现对关键属性信息的提取，并采用 XML 形式描述数据竞争问题。该方法还提供一种共享资源读写访问树的机制，来定位源代码中存在数据竞争的位置。

富浩等采用增强型的锁集合算法，精确细化了数据竞争检测条件，进一步提高检测准确程度，并且减少了错误预报[24]。算法通过采用无干扰型的免插桩检测技术，将集合操作精确映射到位图操作，并避免了二进制代码插桩的不确定性。在片上多核处理器体系架构上实现了该算法，有效地减少了误报率，并具有较高的检测效率。

章隆兵等利用存储一致性模型提出增强发生序的概念，克服了维护发生序方法的缺点，提出一种新的基于锁集合的动态检测方法[25]。相比维护发生序方法，该方法需要检测的执行数目较少，执行效率高，但同时方法的开销很大，在应用过程中会降低软件运行速度。

宋东海等结合了静态检测和程序切片分析技术，提出一种针对 Java 程序

的静态检测方法[26]。该方法利用程序间函数调用层次关系构建函数调用链，对 Java 类进行分析，并通过使用静态切片分析缩小范围，根据数据竞争发生的必要条件，找出可能发生的位置并过滤错误结果。通过消除类上的数据竞争，达到修复程序的目的。

1.4 本文主要内容及组织结构

综上所述，目前国内外文献中存在的数据竞争检测主要有两种：静态检测和动态检测。其中静态检测具有较高的效率，能够全面检测数据竞争的路径，但是存在较高的误报率；动态检测的准确率高，但是不能全面验证所有线程的执行情况，而且运行代价较大。常用的数据竞争检测方法都不能完全检测所有数据竞争问题，并且都存在各自的缺点，本文的主要研究内容是在基于静态检测的数据竞争方法的基础上，结合 Java 程序的特点，提出一种基于字节码指令的数据竞争检测方法。为此，提出本文的主要研究内容：

（1）基于字节码文件，建立一种分析字节码文件的方法，包括对程序模型的抽象，对关键信息的提取等。并且结合提取到的集合信息，采用数据流分析方法和别名分析方法，建立一种基于字节码的程序分析方法。

（2）根据多线程程序数据竞争产生的条件，基于字节码文件分析的数据结果，提出判断数据竞争发生的形式化方法。

（3）使用编程语言实现提出的数据竞争检测方法，并通过实验验证方法的正确性和有效性。在实验过程中，分析实验结果，评价检测方法。

本文的组织结构安排如下：

第 1 章 绪论——首先介绍数据竞争检测的研究背景和意义，然后对国内外的研究现状做出分析，评价各种数据竞争检测方法的优缺点，同时重点介绍描述本文的研究内容及目的。

第 2 章 数据竞争及检测方法——首先介绍数据竞争问题定义以及在多线程程序中发生的例子，再介绍数据竞争检测方法的分类，最后对多线程程序中常用的程序分析技术进行介绍，如别名分析、数据流分析等。

第 3 章 基于 Java 字节码的数据竞争检测方法——首先介绍该方法的整体流程，接着详细对每个步骤进行介绍，包括字节码解析、程序流程建模、可达性计算、别名分析以及锁集合计算等。本章详细介绍了基于 Java 字节码的数据竞争检测方法的步骤和原理。

第 4 章 数据竞争检测工具实现及实验——首先介绍检测工具的整体流

程，然后介绍系统的功能结构设计和基于 Datalog 语言的算法实现，接着介绍各个模块系统的设计和分析；最后对数据竞争检测方法进行实验，分析检测结果，并对检测系统做出评价。

结论——总结本文设计的数据竞争检测方法,对研究工作进行评价总结，并提出今后需要完善的研究内容。

第 2 章 数据竞争及检测方法

2.1 数据竞争问题

2.1.1 数据竞争及产生原因

在多核操作系统中，线程可以并行执行，多个程序指令可能在同一时间被执行。然而每个线程中包含有自己的内存变量资源，而且线程之间的通信必然会涉及到共享资源的访问。如果某时刻存在多个线程并且对共享资源的读写操作被分配到不恰当的顺序执行，则就会发生数据竞争问题。由于数据竞争发生后不会立刻引起程序运行停止，因此调试和检测数据竞争都比较困难。数据竞争的产生使得程序产生不确定的行为，运行结果变得不可预测，因此对并发程序的可靠性和稳定性有一定的影响。

多线程环境下的数据竞争是指两个或两个以上的线程在某个时刻同时访问同一块内存（即同一个变量）时，没有使用合理的同步顺序，并且其中一个线程为写操作[27]。在 Java 程序中，虚拟机的内存模型定义了程序中访问变量的规则，即在虚拟机中如何保存和读取变量[28]。Java 内存模型规定：所有变量都存储在主内存中；每个线程拥有自己的工作内存，其中保存了该线程使用到的主内存的变量的拷贝；线程对变量的所有操作必须在自己的工作内存中进行，不能直接读写主内存中的变量；线程间无法访问不属于自己的工作内存，并且所有线程间变量的传递必须通过主内存完成。因此两个线程间共享变量的传递都要与主内存发生读写操作，所以在 Java 多线程程序中很容易发生数据竞争问题。

在多线程环境下数据竞争问题必须在满足一定的条件时才会发生，具体所需要的条件如下：

（1）至少有两个线程能够同时运行到产生问题的代码；

（2）多个线程对同一内存进行访问，且其中存在写操作；

（3）不存在合理的线程同步顺序。

假定程序中所有运行的线程为集合 T，程序语句集合为 P，变量集合为 V，访问数据的动作为 E（包括读操作 r 和写操作 w 两种），资源锁集合为 LS。数据竞争问题可以描述为五元组(T, P, V, E, LS)，即存在线程 t 和 t'在某时刻同时执行语句 p 和 p'时，对同一内存变量 v 进行访问，其中一个操作为写操

作，而且在访问时得到的资源锁 ls 和 ls'不相同。

采用(t, p, v, e, ls)表示线程 t 在某时刻执行语句 p，使用类型为 e 的操作访问变量 v，在执行语句 p 时得到的资源锁集合为 ls。数据竞争发生的条件表示为至少存在两对五元组(t, p, v, e, ls)和(t', p', v', e', ls')满足下列条件：

（1）$t \neq t'$，表示两个不同的线程；

（2）MHP(t, t')= true，表示线程 t 和 t'可能同时发生；

（3）Alias(v, v')= true，表示变量 v 是 v'的别名，二者指向同一变量；

（4）$(e = r \vee w) \wedge (e' = r)$或 $(e = r) \wedge (e' = r \vee w)$，表示至少有一种操作为写操作；

（5）$\forall a \in ls, \forall a' \in ls'$, Alias(a, a')= false，表示对于执行到语句 p 和 p'的资源锁集合 ls 和 ls'，任意元素中不存在指向相同资源锁的元素存在，即执行语句 p 和 p'时刻不会有等待资源锁的现象发生。

上述条件中，MHP 函数表示判断线程 t 和 t'是否能够同时发生的操作（may happen in parallel，简称 MHP），Alias 函数用来判断输入的两个变量是否指向同一块内存，即某个变量是否为另一个变量的别名。

2.1.2 多线程程序下的数据竞争

在 Java 环境中，进程中始终存在主线程，主线程运行结束则进程结束。多线程环境下，数据竞争主要发生在主线程与用户线程之间，或是在两个用户线程间。Java 多线程程序通过继承机制或实现接口完成新线程的创建，主要创建方式有两种[29]：一种是 java.lang.Thread 类继承，另一种是采用实现 java.lang.Runnable 接口，并在创建线程时传递该 Runnable 对象。通过对源代码的父类和接口关系进行分析，可以找出相应的线程类，从而判断程序是否为多线程程序。在多线程编程中最容易出现对共享变量访问的数据竞争，造成线程间共享变量的数据不一致，引发程序错误，如图 2-1 所示。

在图 2-1 所示代码中，为更容易地表现出数据竞争中共享变量数据不一致的问题，故意将写操作分为三条语句：（1）读取共享变量并保存在局部变量中；（2）改变局部变量数值；（3）写入共享变量中。在账户类 Account 中保存有私有成员变量 money，初始值为 100。Woker1 线程和 Worker2 线程在运行过程中得到 Account 对象的引用，其中 Worker1 线程完成对共享变量 money 加 13 的写操作，而 Worker2 线程则完成减 26 的写操作。由于主线程中没有对共享变量 money 进行任何操作，因此数据竞争发生在两个 Worker 线程之间。

```
class Worker1 extends Thread{
    private Account account;
    public Worker1(Account a){
        this.account = a;
    }
    public void run(){
        int local1 = account.getMoney();
        local1 += 13;
        account.setMoney(local1);
    }
}
class Account{
    private int money = 100;
    public void setMoney(int k){
        money = k;
    }
    public int getMoney(){
        return this.money;
    }
}
```

```
class Worker2 extends Thread{
    private Account account;
    public Worker2(Account a){
        this.account = a;
    }
    public void run(){
        int local2 = account.getMoney();
        local2 -= 26;
        account.setMoney(local2);
    }
}

// main
Account a = new Account();
Worker1 w1 = new Worker1(a);
Worker2 w2 = new Worker2(a);
w1.start();
w2.start();
```

图 2-1 数据竞争共享变量数据不一致问题

表 2-1 详细分析了 Worker1 和 Worker2 线程可能发生的全部情况。表中 R 和 W 分别表示 Worker1 线程的读写操作，即 run 方法中的 getMoney 和 setMoney 方法，r 和 w 代表 Worker2 线程的读写操作。分析过程中忽略其他语句的执行顺序，只考虑对共享变量 money 的读写操作。若两个线程的某些读写操作在同一时刻 T_i 发生，则将对应的读写操作写在一起。

表 2-1 Worker 线程执行结果表

时间长度	T_1	T_2	T_3	T_4	共享变量数值
2	R r	W w			不确定
3	R r	W	w		74
3	R r	w	W		113
3	R	r	W w		不确定
3	r	R	W w		不确定
4	R	r	W	w	74
4	R	r	w	W	113
4	r	R	W	w	74
4	r	R	w	W	113
4	R	W	r	w	87
4	r	w	R	W	87

从上表中可以看出，除去最后两次执行顺序得到正确的执行结果外，其他读写操作顺序下都为错误结果。因此，在多线程编程过程中，如果线程同步顺序安排不正确，可能无法得到正确的结果，产生数据竞争问题。而在目前的计算机软件系统中，并发程序应用广泛，进程间和线程间通信频繁。并发程序间通过文件系统、信号、共享内存、消息队列及网络数据包等方式进行通信，如果通信过程中同步顺序安排不当，出现对共享资源的不恰当访问操作，就可能出现数据竞争问题，带来不可预测的结果。

该例也同时说明，如果使用软件测试的方法，很难发现多线程程序中的错误。因为软件测试人员必须通过设计测试用例，将全部的可能的线程交互顺序覆盖后，才能根据实验结果发现数据竞争问题。

2.2 数据竞争检测方法

数据竞争检测方法主要分为静态检测和动态检测两种。静态检测是在不运行并发程序的基础上，通过分析源代码或编译后的中间文件来发现程序缺陷。动态检测是在代码运行期间，监测程序运行过程中变量的状态，从而判断是否发生数据竞争。此外，还有使用静态检测和动态检测结合的方法，以及其他的方法来检测数据竞争问题。

2.2.1 静态检测方法

相比动态检测，静态检测的优点是能够分析和检测程序执行过程中的全部执行路径，并且容易采用自动化工具实现，分析速度快。但是，由于缺乏程序运行期间的必要信息，使得检测结果中有较多的错误预报，检测结果的准确率低。目前，静态检测已经成功地应用在各种工具上，并且能够处理较大规模的程序，主要的静态分析方法如下：

（1）数据流分析。数据流分析[30]是指在程序编译期间，从程序源代码获取程序的语义信息，然后采用代数方法确定所观测变量的声明定义信息以及该变量的使用情况，最后通过分析程序中某个节点的数据流状态并结合程序的执行路径，从而发现数据竞争问题。数据流分析主要关注对共享变量的复制和引用操作，并被广泛使用在程序验证、编译优化等领域。数据流分析通过在特定程序点时分析所有赋值以及引用操作，查找软件故障，能够取得较好的效果。常用的数据流分析有上下文相关分析、路径分析等方法。

（2）类型推导。由于计算机编程语言本身具有一些编程规则，如变量的

定义方式、函数的使用等，因此可以使用类型推导方法。类型推导通过定义定型断言、推导规则和检查规则，保障程序的正确运行。当所监测的变量或函数不满足类型相关的规则时，就认为程序在违反规则的地方存在软件缺陷。其中，变量的初始类型由定型断言规定，推论系统的规则集合在推导规则中定义，检查规的目的是用来判断推论的结果是否为“良行为”[31]。静态检测方法中的类型推导适用于控制流不相关分析，在编译期间提前发现软件缺陷，常见的能够检测到的缺陷有：数组越界访问、函数返回类型错误、赋值操作中使用错误的数据类型等。

（3）抽象解释理论。抽象解释理论是在程序静态分析时通过构造和逼近程序不动点语义来进行程序分析的一种方法。该理论将程序语义（操作语义或指称语义）看作为在描述对象域上的计算过程或计算结果，从而程序的抽象解释就是指使用另外一个抽象对象域上的计算抽象逼近程序指称的对象域（具体对象域）上的计算，使得程序抽象执行的结果能够反映出程序真实运行的部分信息[32]。抽象解释理论通过在计算效率和计算精度两者之间的权衡，使得能够以较小的精度损失来获取计算的可行性，并使用多次迭代的计算方法，达到增加计算过程精度的目的。基于抽象解释理论的形式化方法被广泛使用在大规模的软硬件系统的测试、分析和验证方面，并且取得了有效的成果，如别名分析等。

（4）模型检测。模型检测的基本思想是采用状态转移系统（S）表示程序的行为，用模态逻辑（F）描述系统的特征，从而将“判断系统是否具有所期望特征性质”的问题转化为一个“状态迁移系统 S 中是否是公式 F 模型”的数学问题[33]，用公式表示为 S|=F。由于模型检测中状态转移系统可以自动执行，并能在系统不满足性质时提供反例路径，因此成为一种重要的自动化验证技术，而且在工业界广泛使用。通过模型检测将程序抽象为状态迁移系统 S，遍历所有可能的程序执行路径，同时监测系统 S 的状态转化，从而达到验证程序是否符合预定要求的目的。该方法对于有限状态的系统可以在有限时间内停机，但当状态空间比较大时，遍历所有执行路径会消耗较多的时间，出现状态空间爆炸。

2.2.2 动态检测方法

动态检测能够获取到程序运行期间的信息，如别名和变量赋值操作等信息，因此分析结果的准确率较高，并且在数据竞争检测工具中大部分使用了动态检测方法，是数据竞争检测方法的主流。但是，检测过程花费较多的时

间，而且某些程序执行路径并不能被覆盖，容易出现漏报[22]。

大多数的动态检测技术主要有三种方式：基于 happens-before 关系（发生序）的检测；基于锁集合（lockset）算法的检测；或是基于两者的混合。

（1）基于发生序的检测方法

Lamport 定义了事件的偏序关系 happens-before[34]，该偏序关系可以用来描述多线程间数据访问事件，具体定义如下：

（a）如果事件 a 和事件 b 在同一运行实体的程序流程中，并且 a 发生在 b 之前，则称作 a happens-before b，记为 a->b；

（b）如果事件 a 是一个运行实体内的发送消息事件，而事件 b 是在另一个运行实体内接受该消息的事件，则 a->b；

（c）如果存在 a->b，并且 b->c，则 a->c；

（d）如果事件 a 和 b 不具有 a->b 的关系，则称为 a 和 b 为并发的。

此外，事件 a 不存在自身的 happens-before 关系，即 happens-before 为反自反的。根据上述定义的的偏序关系，可以用来描述多线程间对共享变量的访问事件。访问变量的事件 a 和 b 满足 a->b 的条件是：事件 a 在程序中位于 b 之前，或事件 b 等待 a 释放资源锁，或从事件 a 通过 happens-before 的传递性可以到达 b。

通过定义 happens-before 规则可以用来描述对共享变量的读写操作，并将每个读写操作看做是一个事件来处理。分别对读写事件的顺序做出要求，从而实现对数据竞争的检测，例如规定 happens-before 一致性[35]。该规则规定对于一个对象的读操作 r 都能满足以下两个条件时，才能认为该读操作正确：（a）所有读操作不能发生在写操作之前；（b）没有一个中间的写操作操做运行期间不能有写操作事件正在发生且等。

但是，采用 happens-before 关系的动态检测方法要求严格，需要实时监测对变量的访问操作，而且该方法并不能够覆盖程序的全部执行路径，有漏报的情况发生。

（2）基于锁集合的检测方法

锁集合算法假设每个共享变量都存在锁机制保护，例如保护共享变量 v 的锁的集合为 C(v)。在程序运行过程中，若某个线程得到资源锁，则更新锁集合 C(v)为 C(v)和当前线程拥有锁集合的交集。当锁集合为空时，表示此事没有资源锁保护变量 v。此时，算法发出警告，表明在目前执行路径中可能存在数据竞争问题。算法的执行过程如图 2-2 所示。

图 2-2 中左图 a）中两个线程访问共享变量 v，分别使用锁对象 m 和 n

保护共享变量。在左图中锁集合算法能够正确地检测数据竞争发生的位置。当两个线程的获取锁的顺序如左图 a）所示时，保护在共享变量 v 上的锁集合 C(v)在空集时发出警报，有效地判断出数据竞争的发生。但是在右图 b）中，当共享变量 v 在线程 Thread1 中没有被锁保护时，锁集合 C(v)中仅有锁对象 m。按照右图的顺序 Thread2 在对共享变量 v 赋值后启动，此时 Thread1 和 Thread2 可能存在数据竞争。因为 Thread1 在启动之后存在对共享变量 v 的写操作。但是，算法不会发出警报，此时锁集合 C(v)不为空，发生数据竞争漏报的情况。此外，锁集合算法需要预先得到保护变量的锁集合，而采用动态分析方法时，该集合的获取通常是通过反复多次运行程序得到的，也存在漏掉锁集合元素的情况。

Thread1	Thread2	C(v)

lock(m);
 	
v = v + 1;
unlock(m);
	
	lock(n);
	
	v = v + 4;
	unlock(n);
		{m,n}
		{m}
		{}

a) 正确检测数据竞争示意图

Thread1	Thread2	C(v)

int v;
v = 5;
t.start();
v=v+1;
	
	lock(m);
	v=v+ 2;
	unlock(m);
		{m}
		
		{m}

b) 漏报数据竞争示意图

图 2-2 使用 lockset 算法进行数据竞争检测示意图

文献[18]提出了实现了锁集合算法的工具 Eraser，并且是第一次使用锁集合算法。但是由于该算法发出的警报并不能代表一定有数据竞争问题发生，而且对没有资源锁保护的共享变量无法做出判断，因此实验结果中存在较多的误报，准确率较低。

此外，还有研究者采用混合 happens-before 和 lockset 算法的检测。为结合两种常用的动态检测技术的优点，将两个方法结合考虑。在动态检测过程中，使用改进后的 lockset 算法检测共享变量的锁集合，并且跟踪读写事件的并发顺序，从而降低了错误率，检测到更多的数据竞争问题。

为综合静态检测和动态检测的优点，有的研究者使用混合静态检测和动态检测的方法。其中，简单的混合方法分为：首先通过静态检测方法找出所有可能发生数据竞争的位置，然后再使用动态检测方法进行验证，最后将其中错误预报过滤[36]。使用简单的混合方法的覆盖率由静态检测方法决定，主要的研究重点仍是静态检测，目的是能够覆盖程序中的全部路径。由于该方法在后期需要动态检测过滤误报，因此检测算法运行时间更长，不具有实际应用价值。文献[37]第一次提出了一种结合静态检测和动态检测的工具

JNuke。该工具在静态分析阶段将程序抽象为状态图，然后在动态分析阶段利用状态图分析数据竞争问题。通过将静态分析方法和动态分析方法抽象为一种广义的检测方法，使得该方法能在多种应用场景下运行。文献[38-39]在分布式系统的基础上，使用单个节点做静态分析并把动态分析过程分别交给多台机器执行。在执行动态分析的过程中，每个节点实时通知中心节点执行的位置。中心节点可以判断出重复的执行路径，并及时通知重复运行的节点重新开始执行，确保检测到全部的静态分析结果，减少数据竞争的误报。

2.3 多线程程序分析技术

2.3.1 字节码指令分析技术

在 Java 程序中，源代码经过编译之后生成 class 文件，在 class 文件中保存了反映程序流程的字节码指令。字节码指令通常有操作符和操作数组成，操作符描述了字节码指令的执行操作，操作数表述要处理的数据。但是由于字节码指令集采用的是面向操作数栈的方式，并不是汇编语言中使用的寄存器架构，因此大多数指令都不包含操作数。图 2-3 显示了左图的 Java 程序在编译后生成的 Java 字节码指令。

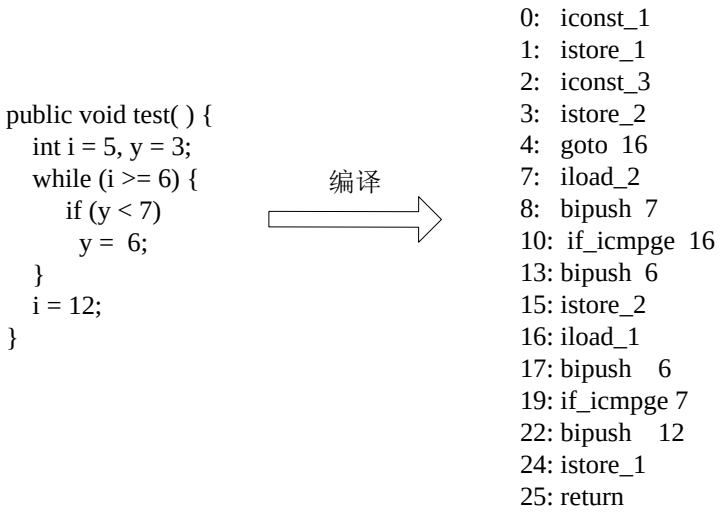

图 2-3 Java 程序编译过程示意图

在程序分析领域，较早的研究通常是基于源代码程序的分析。但是，近几年来，对于 Java 程序的分析，越来越多的研究集中在对字节码指令的分析上[40-41]。对于多线程并发程序而言，相对于在源代码分析上，采用字节码指令分析有以下优点：

（1）字节码指令更能体现出语句的详细执行步骤，便于程序分析。在源

代码分析阶段，通常都需要有语法分析、抽象语法树分析等步骤，目的是为了分析程序语句的语义。但是对于字节码指令而言，没有过多的语法内容。字节码指令的操作码明确表明了要进行的操作，而且更能反映出语句的执行细节。例如对于静态变量 i 执行语句 i+=3，在字节码指令中为表现为 getstatic、iadd 与 putstaic 三条指令。其中，getstatic 指令用于将静态变量存放在主内存中，iadd 指令将用于改变变量的值，putstatic 指令用于将结果写回到原变量的内存中。从以上步骤中可以清楚的发现，对于静态变量 i 采用了先读后写的操作。如果使用源代码分析，这一详细步骤将依赖于具体实现，再加上 Java 语言中语法规则的多样性，很容易出现分析错误。

（2）对于 Java 程序而言，使用字节码文件可以节省程序分析时间，并且简化程序分析步骤。在字节码文件中，保存有该类的父类关系、所实现的接口、方法的局部变量列表等信息，而且字节码文件格式固定，便于提取信息。如果使用源代码分析，不仅需要考虑面向对象的语法实现，而且需要自己分析函数的变量定义等，加大了程序分析难度。字节码文件中保存的信息屏蔽了语法细节，使得程序分析更加方便。

（3）基于字节码分析应用范围广泛，而且在并发程序分析上比源代码分析有优势。对于已经投入使用的计算机软件，不可能获取程序的源代码，但由于字节码指令文件是直接用来交付的最终产品。因此，使用字节码指令分析具有更广泛的应用背景和意义。近几年来，越来越多的软件形式化方法中使用字节码指令，如文献[40]中使用字节码的模块化验证的逻辑系统。此外，在并发程序的分析方法中也开始使用了字节码指令分析技术，如基于 Java 的数据流分析[41]，多线程程序的同步优化等。

2.3.2 程序分析技术

在 Java 多线程程序中，为保证共享变量的数据一致性，编程时通常采用线程同步机制。不同的编程环境下实现的同步机制也不相同。Java 语言提供的线程同步机制有：synchronized 关键字、wait 和 notify 同步方法、join 同步方法等。在数据竞争问题中，本课题主要关注 synchronized 关键字的同步机制。在多线程程序的数据竞争检测的程序分析方法中，本文中使用到的程序分析技术有别名分析、数据流分析和基于 Datalog 语言的程序分析。

（1）别名分析。别名分析又叫指向分析，是一种静态分析技术，用来决定哪个变量指向哪一块内存。别名分析的分为流敏感、流不敏感、上下文敏感和上下文不敏感四种[42]。流敏感的别名分析方法考虑函数内的控制流信

息，因此分析结果精度较高，但运行时间较长且空间复杂度高。流不敏感的别名分析方法不适用方法内的控制流信息，效率高但精度低。上下文敏感的别名分析方法考虑函数调用时的指针指向模式，如果模式不同，则分析结果也不同。而上下文不敏感的别名分析则不考虑此情况，因此同一函数的不同调用点总是返回唯一的结果。在数据竞争检测的过程中，本文中使用上下文相关的别名分析方法。通过别名分析用来判断并发的访问操作是否针对同一块内存变量，从而判断是否可能发生数据竞争问题。

（2）数据流分析。本文中使用到的数据流分析主要是用来构建函数内部的控制流程图和函数间的调用关系图。通过使用字节码分析技术，将字节码指令分块，并使用有向边连接，将其构建为函数内部的控制流程图。函数间的调用关系图主要是用来分析访问数据操作的可达性。

（3）基于 Datalog 的程序分析技术[43]。Datalog 是一门函数式编程语言，主要用来表达集合直接的逻辑关系，函数的定义由三部分组成：域（Domain）、关系（Relation）和规则（Rules）。域规定了语言的输入集合，关系是主要用来描述进行计算的有序对，规则定义了一种满足条件的特殊关系。算法可以按照规则进行集合运算，而且规则中列出的条件必须满足。为简单的描述 Datalog 语言的规则定义，举例如下。假定城市 1 至城市 4 之间的航线图如图 2-4 所示，现有两家航空公司 A 和 B，分别行驶不同的航线。

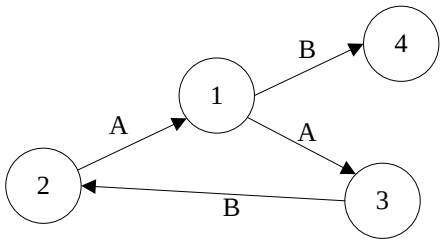

图 2-4 城市间航线示意图

某乘客在飞行过程中，先乘坐 A 公司航线，再乘坐 B 公司航线，那么，该乘客所有可能的路径可以使用规则 C(x,y)表示为 C(x,y) :- A(x,z), B(z,y)。其中乘坐 A 公司航线的规则是 A(2,1)或 A(1,4)；乘坐 B 公司的航线规则是 B(1,4)或 B(3,2)。任意两个城市之间的可达性可以用规则 D(x,y)表达如下：

D(x,y) :- A(x,y) (2-1)

D(x,y) :- B(x,y) (2-2)

D(x,y) :- D(x,z),D(z,y) (2-3)

上式的含义是 D(x,y)为真，当且仅当后面的所有式子都为真。两个城市

间的可达有两种情况：一种是直接通过某个公司的航线可达，另一种是通过在可达的城市 z 中转后，在下个目的地 y 可达。

通过使用 Datalog 语言，可以用来对算法进行形式化描述。在程序分析领域，使用 Datalog 语言成为一种常用的程序分析方法[44]。本文使用 Datalog 语言进行数据竞争检测算法的描述，通过使用该方法表达检测算法中关系集合的运算过程，如数据访问操作的可达性、访问操作是否访问同一内存变量以及访问路径中是否被同一锁对象保护。

2.4 本章小结

本章首先介绍了数据竞争问题的定义、产生的原因以及形式化的表示，详细阐述了 Java 多线程程序中由于数据竞争问题带来的数据不一致问题；然后介绍了常用的数据竞争检测方法，对每种方法的优缺点做出了评价，并简单介绍了几种常用的数据竞争检测方法的思路；最后对本文所使用的程序分析方法进行了介绍，包括字节码指令分析方法、别名分析方法以及基于 Datalog 的程序分析技术，为下一章提出的基于 Java 字节码的数据竞争检测方法奠定基础。

第 3 章 基于 Java 字节码的数据竞争检测方法

3.1 检测方法流程

在研究国内外已有的数据竞争检测方法基础上，针对 Java 多线程并发程序，通过对字节码指令文件的解析，提出一种基于 Java 字节码的数据竞争静态检测方法。基于 Java 字节码的数据竞争检测方法使用数据流分析和别名分析等静态检测技术，对多线程程序进行抽象建模，并根据数据竞争产生的必要条件，一步步地检测数据竞争问题，检测方法的整体流程如图 3-1 所示。

图 3-1 数据竞争检测方法流程图

数据竞争检测方法的输入是 Java 字节码文件。在实际工程应用中，已交付使用的 Java 软件程序以 jar 包的形式存在，源代码的获取比较困难；而正在开发中的 Java 程序，能够获取源代码文件。为了能够检测更多的 Java 程序，系统将从编译后的字节码文件作为系统的主要输入数据。

相比程序源代码，数据竞争检测方法中使用字节码文件分析更加容易。其主要原因是源代码中涉及较多的语法规则，需要复杂的语法分析和语义分析后才能提取到所需要的信息，而编译后的字节码文件格式固定且结构简单，能够更准确地提取变量、函数调用等信息。

根据图 3-1 所示，将系统的数据竞争检测流程分为五个步骤：

（1）字节码解析。根据编译后的 class 文件格式，提取数据竞争检测所需要的信息，包括变量列表、具体实例类列表、函数列表、线程列表、函数调用点集合、程序语句集合、访问数据的指令、对象分配指令和函数调用指令等，并将这些信息抽象为集合的形式，保存在 xml 文件中。

（2）程序流程建模。基于提取出的集合信息，并通过对字节码文件中指令流程的分析，抽象为形式化模型，用以描述多线程程序的执行过程。此步骤中重点分析的内容有：控制流程图（control flow graph，简称 CFG 图），函数调用关系图等，为可达性计算、别名分析和锁集合计算提供数据。控制流程图主要是针对函数内部的处理逻辑，通过分析字节码指令的操作码，将程

序分解为几段代码块，并用有向边连接。函数间的调用过程使用函数调用关系图来描述，结合函数调用的上下文建立上下文敏感的调用关系图。

（3）可达性计算。通过判断两个访问操作是否可达，进而来判断执行这两个访问操作的线程是否可以同时发生，从而判断是否满足数据竞争的并发性条件。可达性计算是通过判断关键函数的可达性来实现的。如果从 main 方法中存在一条路径，使得线程产生的调用函数可达，则在程序执行过程中，可能会产生该线程。同理，如果从该线程的 run 方法开始，跟踪所有在 run 方法中调用的函数，如果这些函数中包含有访问数据的操作，那么这些操作也是可达的。在对两个线程的访问操作进行可达性计算之后，尽可以判断这两个访问操作是否可能同时发生。

（4)别名分析。通过别名分析保证两个线程的访问操作指向同一块内存。对于对象属性变量而言，需要找到访问操作的局部变量的指向集合，并判断两个集合是否有交集。对于静态变量而言，需要判断两个访问操作是否为同一个静态变量。

（5)锁集合计算。数据竞争的产生必须不能被同一个锁对象保护，因此，需要跟踪线程运行期间的锁操作，记录在路径上保持的锁集合。如果两个线程保持的锁集合交集为空集，则这两个访问内存操作没有被同一个锁对象保护，可能发生数据竞争。

3.2 字节码解析

由于 Java 是基于类的面向对象语言，需要考虑面向对象的特征，如继承、多态等，因此需要考虑程序中的数据类型、上下文环境等信息。在获取得到的类文件中，根据类文件的组织结构，读取基本信息；然后通过对字节码指令的分析，建立抽象程序模型。例如，可以根据字节码指令 getstaitc 和 putstatic 抽象得到访问静态变量的操作。字节码文件是整个系统的原始数据，需要考虑的字节码指令有以下几类：

（1）访问类 static 成员变量和实例变量的指令，如 getfield、putfield、getstatic 和 putstatic。

（2）创建类实例的指令：new。

（3）加载和存储变量指令：load 和 store。

（4）分支指令：if、if_cmp、goto 和 jsr 等。

（5）方法调用指令：调用实例方法的 invokevirtual 指令、调用静态方法

的 invokestatic 指令、调用接口方法的 invokeinterface 指令和调用动态方法的 invokedynamic 指令。

（6）同步指令：monitorenter 和 monitorexit。

整个字节码解析过程中所获取到的程序基本信息描述如表 3-1 所示：

表 3-1 程序分析过程获取的基本信息表

集合名称	标志	描述信息
抽象线程集合	A	程序中的所有的线程类
上下文集合	C	函数调用或对象分配时的字节码指令集合
访问数据指令集	E	函数中访问数据的指令集合
属性集合	F	类中定义的非静态变量集合
静态变量集合	G	类中所有的静态变量的集合
对象分配指令集	H	函数中使用对象分配的指令集合
调用点集合	I	函数调用时的指令集合
加锁指令集合	L	所有的获取锁的指令集合
函数集合	M	程序中全部定义的函数集合
字节码指令集	Q	函数中使用的字节码指令集合
释放锁指令集	R	所有的释放锁的指令集合
类型集合	T	程序中使用的类及基本数据类型集合
局部变量集合	V	函数中的局部引用变量集合

由于 Java 源代码在编译生成的 class 文件中，保存了常量表、父类描述信息、接口描述信息、属性列表、函数列表、局部变量列表等信息，因此可以方便的得到表 3-1 所示的基本信息，并将结果保存在中间文件 XML 中，例如描述抽象线程集合的 Alist.xml，所有函数方法的集合 Mlist.xml 等。如图 3-2 中显示了函数集合文件 Mlist.xml。

```
<Mlist>
<M id="M9700" sign="test.Worker.&lt;clinit&gt;()" file="test/Worker.java" line="4"></M>
<M id="M9701" sign="test.Worker.access$002(int)" file="test/Worker.java" line="3"></M>
<M id="M9702" sign="test.Worker$1.&lt;init&gt;()" file="test/Worker.java" line="7"></M>
<M id="M9703" sign="test.Worker$1.run()" file="test/Worker.java" line="9"></M>
<M id="M9704" sign="test.Worker$2.&lt;init&gt;()" file="test/Worker.java" line="12"></M>
<M id="M9705" sign="test.Worker$2.run()" file="test/Worker.java" line="14"></M>
</Mlist>
```

图 3-2 函数集合文件 Mlist.xml

字节码解析过程中，基本信息的处理情况都是直接来自与 class 文件的信

息。对于抽象线程，可以通过读取该类所实现的接口和继承的父类信息可以判断该类是否是线程类，从而决定是否将该类放入到集合 A 中。如果该类继承自 java.lang.Thread 类的子类或 Thread 的匿名类，则应该放入集合 A 中。同理，分析字节码指令的操作，将函数调用的 invoke 指令和分配内存的 new 指令加入集合 C，访问数据的字节码指令放入集合 E，分配内存的指令放入集合 H，函数调用指令放入集合 I，获取资源锁和释放资源锁的指令分别放入集合 L 和 R；通过读取局部变量列表、函数列表以及属性列表，形成集合 V、M 和 F 等。所有的字节码指令形成集合 Q，所有的数据结构形成集合 T。所有基本信息集合中，每个元素都具有唯一标记 id。此外，由于程序从 main 方法开始执行，main 方法的上下文具有特殊含义，定义为 ε。

3.3 程序流程建模

3.3.1 控制流程图的生成

由于 Java 字节码指令包括一个字节长度的操作码和随后的零个或多个操作数构成，为方便考虑程序的执行顺序，需要构建控制流程图（CFG 图）。流程控制图是通过图形化的方式将程序执行过程中可能执行的全部路径进行显示的一种表现方式[45]。通过使用 CFG 图，能够清楚地显示可能并发执行的程序语句，从而实现对检测程序的数据流分析，获取到相关的信息。

控制流程图由代码块和有向边组成。代码块（basic block，简称 BB）是指一组只能从该代码块的入口进入，出口结束的指令集合。每个代码块可以有多个出口和多个入口，但代码块中不能有多条执行路径，只能是从开始执行到结束。控制流程图中的有向边连接两个代码块，表示从一个代码块执行结束后转向另一个代码块继续执行，指出指令的执行顺序，如图 3-3 所示。

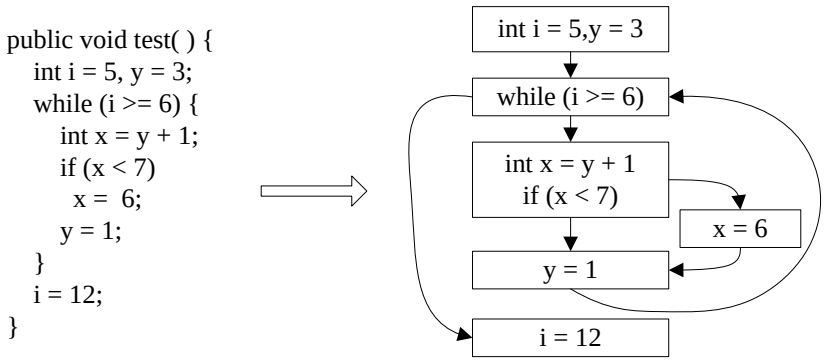

图 3-3 基于源代码程序的控制流程图

控制流程图 CFG 不同于程序流程图，只需要标明执行代码的流程即可，

代码块的形状一般为方形或圆形。为了减少控制流程图中的节点数量，系统中所使用的代码块都尽量包括较多的程序代码，如图 3-3 中将变量 x 的赋值语句与 if 语句放在同一个代码块中。

为减少控制流程图的复杂程度，本系统不考虑构建全局的控制流程图，只针对多线程程序中的每个方法构建程序流程图。控制流程图的生成有两个步骤：第一步是将字节码指令分割为尽可能少的代码块；第二步是将得到的代码块通过指令流程用有向边连接起来。

本系统中使用的是基于字节码的程序分析方法，因此，不直接从程序源代码分析，而是使用 class 文件中的字节码指令集合。将图 3-3 中左边的程序源代码编译后，得到如图 3-4 左边显示的标号为 0 到 31 的 22 条字节码指令，根据字节码指令的操作符将生成的控制流程图如图 3-4 所示。

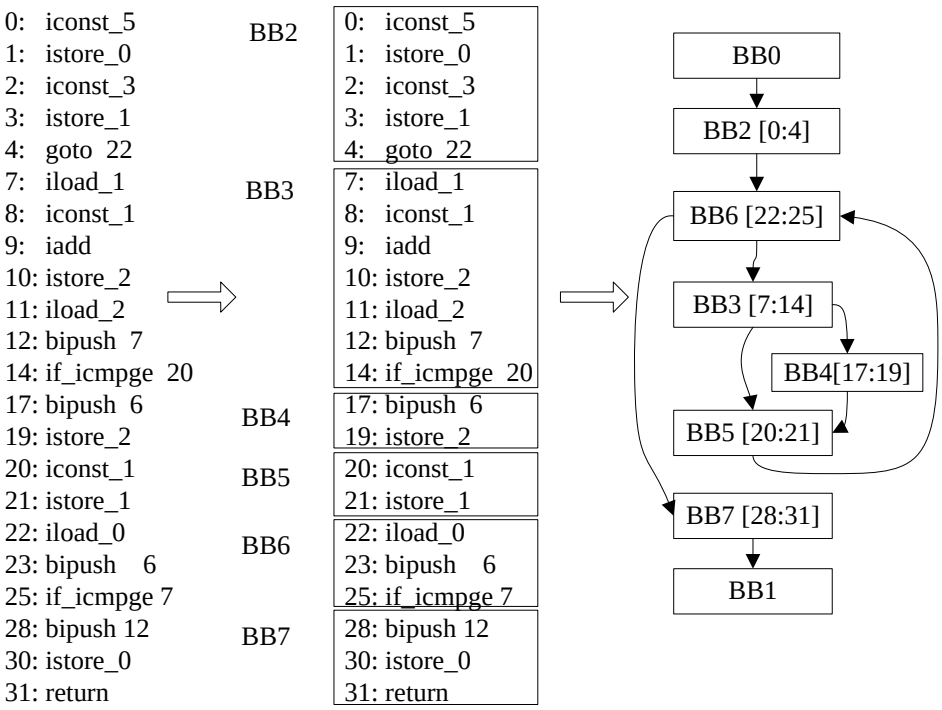

图 3-4 基于字节码指令的控制流程图

图 3-4 中所示的代码块中，代码块的编号根据指令出现的次序，其中 BB0 和 BB1 代表整个方法的入口和出口，方便分析程序处理。通过对 goto、ifcmp 和 if 等程序分支的字节码指令的分析，将字节码指令集合划分为代码块集合。其中 goto 指令是无条件跳转指令，只有一个出口；而 if 指令和 ifcmp 指令为有条件的跳转指令，可以存在两个出口。上图中 if_icmpge 7 指令的含义是，如果比较结果满足条件，将跳转到标号为 7 的指令开始执行，否则继续执行

下一条指令。通过分析分支指令的位置，可以将字节码指令集合划分为代码块集合，详细的划分算法如表 3-2 所示。

算法的输入是 n 条字节码指令集合 Inst，其中第 i 条指令用 Inst[i]表示。算法的输出是代码块的集合 Block，集合 Head 中的 Head[i]表示第 i 个的代码块 Block[i]的第一条字节码指令。

表 3-2 基于字节码指令的代码块划分算法

算法 3-1 代码块划分算法

```
输入：n 条字节码指令集合 Inst
输出：代码块头节点集合 Head、Block 集合
1.  Head = {0}     //初始化 Head 集合为第 0 条指令
2.  for i from 1 to n
3.      if Inst[i] is a branch inst     //如果指令 inst[i]是分支指令
4.          Flag = getDest(Inst[i]) //取得指令 inst[i]的分支集合
5.          Head = Head∪Flag
6.      endif
7.  endfor
8.  j = 0
9.  foreach h in Head
10.     Block[j] = {h}     //将头节点放入 block[j]集合
11.     k = h + size(h)     //头指令之后的下一条指令标号
12.     while k <= n and k∉Head
13.         Block[j] = Block[j]∪{k}
14.         k = k + size(k)     //指向下一条指令
15.     endwhile
16.     j = j+1
17. endfor
```

根据表 3-2 所示算法，将图 3-4 的字节码指令集作为输入，遍历每条字节码指令，当遍历到标号为 4 的 goto 指令时将 22 并入 Head 集合。同理，分别在标号为 14 和 25 的 ifcmp 指令时，将 17、20、7 和 28 并入 Head 集合。最终形成的 Head 集合为{0，7，17，20，22，28}，从而完成对字节码指令的分割，形成 8 段代码块（包括入口和出口）。接着，第二次遍历整个指令集合，按照分割点将指令存放在各自的 Block 集合中。

完成后第一步的代码块的划分之后，开始进行第二步的处理，将代码块用有向边连接，形成 CFG 图。算法的输入是代码块集合 Block，算法的基本思路是遍历 Block 集合中的每个代码块，如果当前代码块的最后一个字节码指令为分支指令，则将有向边从当前代码块指向可能产生分支的代码块。

算法中并没有对产生程序分支的字节码指令进行详细分类，一般而言，goto 字节码指令无条件指向标号所指向的位置，而 if 指令和 ifcmp 指令则根据判断条件，产生两个程序分支，详细的算法描述如表 3-3 所示。

表 3-3 基于字节码指令控制流程图生成算法

算法 3-2 控制流程图生成算法

```
输入：含有 m 个元素的代码块集合 Block
输出：控制流程图 CFG(Block, Edge)
1.    for i = 1 to m
2.        inst = getLastInst(Block[i]) //得到代码块最后一条指令
3.        if inst is a branch inst
4.            Target = getDest(inst) //取得指令 inst 的分支集合
5.            for trgt in Target
6.                CFG.CreateEdge(inst, trgt) //连接两个代码块
7.            endfor
8.        else
9.            // 连接该代码块与下一个代码块
10.           CFG.CreateEdge(inst, getNext(inst))
11.       endif
12.   endfor
```

3.2.2 函数调用关系图构建

多线程程序中每个方法的控制流程图是数据竞争检测的基础，根据基本信息集合，分析每个函数的控制流程图，可以得到很多关键的信息，如函数调用关系图等（call graph，简称 CG 图）。

函数调用关系图是通过在控制流程图的基础上，分析 invoke 字节码指令，得到调用函数和被调用函数的信息，并结合程序中的调用点集合 I，采用广度优先遍历算法，得到的静态分析图。

函数调用关系图算法的思路是：首先，保存一个函数集合 mlist，在算法开始初始化为程序中可能的函数执行的起点，如 main 方法、各个类的构造方法等；然后，从函数集合 mlist 中取出任意一个函数 source，并获取函数 source 的 CFG 图中的指令集合；接着遍历指令集合中的每条指令，如果该指令是 invoke 调用函数的指令，则对该指令的被调用函数进行解析，获取可能调用的所有被调用函数的集合；最后，将 invoke 指令的调用者和所有被调用者直接使用有向边连接，形成函数调用关系图 callgraph。算法的详细步骤如表 3-4 所示。

表 3-4 基于字节码指令的函数调用关系图生成算法

算法 3-3 函数调用关系图生成算法

```
输入：函数集合 Method，集合 mlist
输出：函数调用关系图 callgraph
1.   mlist.initialize() //初始化集合 mlist
2.   while mlist is not empty
3.       source = mlist.removeNext()   //从集合中取出一个函数
4.       foreach inst in source          //遍历每一条字节码指令
5.           if inst is a invoke inst   //如果某指令为 invk 指令
6.              destination = resolve(source, inst) //查找函数
7.              foreach dest in destination
8.                  mlist.add(dest)
9.                  callgraph.addEdge(source, dest)
10.             endfor
11.         endif
12.      endfor
13.  endwhile
```

算法 3-3 中使用的 resolve 函数，根据调用函数 source 和被调用函数的签名 dest，判断出可能执行的所有函数列表。resolve 函数主要考虑的情形有继承关系下的函数调用，同名函数的重载等语法细节。如图 3-5 所示的 Java 多线程程序，图中标记了每个方法中函数调用点，从 i1 一直到 i10，并且在产生新对象的语句上使用 h 标记。

```
    public  class C extends Thread {
      private B f1;
      public C(B v1) {this.f1 = v1;}
      public void run() {
 i1:    if (f1.get() == 0)
 i2:      f1.set(5);
      }
      public static void main(String args[]) {
i3,h1:   B v2 = new B();
         for (int i = 0; i < 5; i++) {
i4,h2:     C v3 = new C(v2);
 i5:       v3.start();
         }
 i6:     if (v2.get() == 0)
 i7:       v2.set(1);
      }
    }

    public  class B {
      private A f2;
      public B() {
i8,h3:   this. f2 = new A();
      }
      public int get() {
 i9:    return f2.get();
      }
      public void set(int i) {
i10:    f2.set(i);
      }
    }
    public  class A {
      private int f3;
      public int get() {return f3;}
      public void set(int i) { f3 = i;}
    }
```

图 3-5 Java 多线程程序

在上图所示的程序中，从 main 方法开始，调用的函数有类 B 的构造方法、类 C 的构造方法、线程启动方法 start 以及类 B 的 get 和 set 方法。而在类 C 的 run 方法中调用类 B 的 get 和 set 方法，类 B 的 get 和 set 方法又调用类 A 的相应的方法，整个函数的调用过程图如图 3-6 所示。

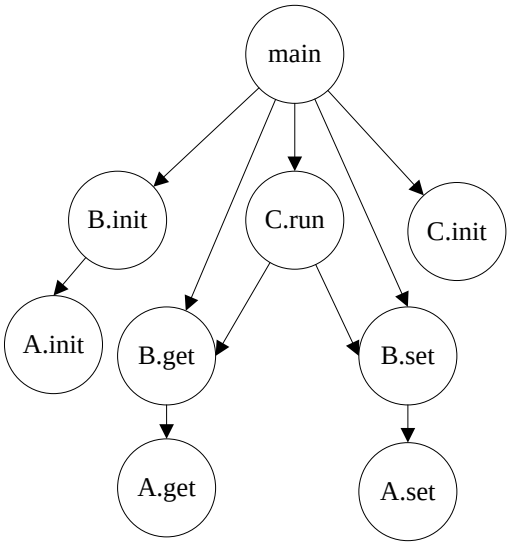

图 3-6 Java 多线程程序函数调用关系图

在函数调用关系图的基础上，考虑上下文切换过程，关注被调用函数的上下文环境，在 CG 图中做出标记，从而能够表示在哪些上下文环境中有哪些的函数被调用，形成上下文敏感的函数调用关系图（context sensitive call graph，简称 CSCG 图）。分析 CG 图可以得到函数的可达性，并根据可达性判断任意两个函数中的访问变量操作是否可以同时发生。CSCG 图可以表示为四元组 CICM(c,i,c',m)，表示在上下文为 c 的调用点 i 处调用函数 m，并且调用后的上下文为 c'。例如，分析图 3-5 中的程序，该程序的 CSCG 图使用四元组 CICM 表示如下。

CICM = {(ε，i3，[h1]，B.<init>)，(ε，i4，[h2]，C.<init>)，(ε，i5，[h2]，C.run)，(ε，i6，[h1]，B.get)，(ε，i7，[h1]，B.set)，(h2，i1，[h1]，B.get)，(h2，i2，[h1]，B.set)，(h1，i8，[h3:h1]，A.<init>)，(h1，i9，[h3:h1]，A.get)，(h1，i10，[h3:h1]，A.set)}。

上述集合中，main 方法的上下文为 ε，如(ε，i3，h1，B.<init>)表示在 main 方法中的 i3 调用点调用了类 B 的构造方法，并且调用之后的上下文为 h1。而且，由于 Java 多线程中线程启动的 start 方法会自动调用 run 方法，所以在集合中用 run 方法取代 start。集合中的上下文使用对象调用语句来标记，

并采用对象调用序列的集合表示。CICM 四元组关系可以用来描述函数的调用过程，即上下文相关的函数调用图。

在程序分析过程中，还需要通过控制流程图和基本信息进行运算来得到一些关系集合。这些关系集合主要用来进行函数判断，便于以后的数据竞争检测时使用，其中主要的关系如下所示。

（1）MI(m, i) 表示在函数 m 中在存在调用指令 i，调用了其他函数。

（2）ME(m, e) 表示在函数 m 中存在访问内存变量指令 e。

（3）IC(i, c) 表示调用点 i 与上下文集合的有序对的集合。

（4）MC(m, c) 表示函数 m 与上下文 c 的有序对。

3.4 可达性计算

3.4.1 访问内存操作分析

由于在 Java 的程序中，类中变量的定义可以为类的静态 static 变量或为对象的实例变量。因此将 F 定义为所有类的非静态实例变量集合，将 G 定义为所有类的静态变量集合，将 E 定义为访问变量的操作集合，且不区分读和写操作，将访问变量的操作所处的上下文环境定义为 C 集合，函数的调用点定义为 I 集合，所有函数定义为 M 集合。

在程序中使用 y = x.f的形式对实例对象的属性进行读操作的语句集合定义为 Fread；同理，使用 x.f = y的形式对实例对象的属性进行写操作的集合定义为 Fwrite。Gread 集合表示使用 y=ClassName.f 的方式对 static 属性的进行读操，而对使用 ClassName.f=y 的 static 属性的写操作定义为 Gwrite。

根据以上对程序属性的描述，可以得到以下表达形式。

$e \in E = E_f \cup E_g$ 表示对变量进行访问操作有两种，一种是对类的实例变量进行访问，另一种是对静态变量进行访问。$Fread, Fwrite \subseteq E_f \times F$，表示对于实例变量的读写操作为访问实例变量的动作与特定的某个实例变量 F 的笛卡尔积。同理，$Gread, Gwrite \subseteq E_g \times F$。

计算访问内存操作组合(e_1, e_2)是否满足数据竞争问题中至少有一个为写操作的条件。判断(e_1, e_2)的访问内存操作是否至少有一个为写操作的条件表示如下：

$$FRW = \cup_{f \in F}\{(e_1, e_2) \mid (e_1, f) \in Fread \cup Fwrite \ \wedge \ (e_2, f) \in Fwrite\}$$

$$GRW = \cup_{g \in G}\{(e_1, e_2) \mid (e_1, f) \in Gread \cup Gwrite \ \wedge \ (e_2, f) \in Gwrite\}$$

$$RWPairs = FRW \cup GRW$$

若访问操作(e_1, e_2)满足式子 RWPairs 则表示中至少有一个为写操作。

3.4.2 线程调用函数过程分析

对本步骤分析过程中可能使用到的函数关系，需要做出的函数定义如下所示。

（1）定义函数 FindM。根据程序分析过程中定义的 ME(m, e)、MI(m, i)关系，定义函数 FindM，即对于任意给定的 $i \in I$ 或 $e \in E$，该函数能够返回方法 $m \in M$，并使得元素$(m, e) \in ME$ 或 $(m, i) \in MI$。该函数的含义是根据特定的函数调用点或访问内存的动作，找到包含有该调用语句和访问内存语句的函数，即对于 i 或 e，使得$\exists m\{m \mid (m,i) \in MI \vee (m,e) \in ME\}$条件满足。

（2）定义函数 FindIC。对于给定的调用点 i 和函数调用者上下文 c，返回所有被调用的函数 m 和对应的上下文 c'的有序对，即查找满足调用者条件的 CICM 元素，FindIC(i, c)={(m,c')|$\exists$ m,c':(c,i,c',m)$\in$CICM}。

（3）定义函数调用过程。某个线程从调用点为 i 且上下文关系为 c'开始，能够到达方法 m，并且此时上下文为 c，用符号表示此调用转化过程为$(i,c') \to (m, c)$。如果在该函数调用过程中，线程没有发生转换，即没有发生线程交替的行为，则表示为$(i,c') \Rightarrow (m,c)$，形式化描述如下。

$(i,c') \to (m,c)$等价于$(m, c) \in \text{FindIC}(i, c')$，即线程通过调用点 i 到达函数 m。

$(i, c') \Rightarrow (m, c)$等价于$(i,c') \to (m,c) \wedge i \notin I_{fork}$，$I_{fork}$ 为产生线程的调用点。

（4）定义函数 m 的可达性。某函数 m 在上下文 c 中是可达的，是指存在一条以(i,c')为起点的函数调用路径，该路径表示为 $(i,c') \to^{*} (m,c)$。同理，如果在调用过程中，没有经过产生新线程的调用点 I_{fork}，则为 $(i,c') \Rightarrow^{*} (m,c)$。经过 n+1 个调用点到达函数 m，表示为 $(i,c'') \to^{n+1} (m,c)$，该式子的意思是存在一个通过 n 个调用点到达的中间函数 m'，并且在函数 m'中存在调用点 i'，使得通过调用点 i'能够到达函数 m。即

$$\exists m', i', c' : (i,c'') \to^{n} (m',c') \wedge \text{FindM}(i') = m' \wedge (i',c') \to (m,c)$$

如果在经过的调用点中，都不存在 I_{fork} 的元素，则表示为 $(i,c'') \Rightarrow^{n+1} (m,c)$，即等价于 $\exists m', i', c' : (i,c'') \Rightarrow^{n} (m',c') \wedge \text{FindM}(i') = m' \wedge (i',c') \Rightarrow (m,c)$。

（5）定义线程能够被创建的条件。假设在程序中产生新的线程的调用点为 i 且上下文为 c，则包含该调用点的函数为 FindM(i)，那么该线程能够被程序运行时产生的条件是：在 main 函数中存在调用点 i'，并且存在一条以(i', ε)为起点的函数调用过程路径，该路径的终点为(FindM(i), c)，将此定义为 ReachIC，即

$$ReachIC=\{(i,c)|i\in I_{fork} \wedge \exists i':FindM(i')=main \wedge (i',\varepsilon)\rightarrow^{*}(FindM(i),c)\}$$

ReachIC 关系所描述的路径中，可以发生线程交替行为。因为在程序中新线程可以有主线程产生（主线程运行 main 方法），也可以由其他线程产生，因此，使用在定义过程中使用单横线箭头。

根据以上定义，描述两个线程的访问内存操作 e_1 和 e_2 可达的条件：

（a）两个线程的创建函数是可达的；

（b）从线程创建函数到的读写操作 e_1 和 e_2 所在的函数必须是可达的，并且函数调用过程中不能产生线程交替；

（c）访问内存的操作中至少有一个为写操作。

假设两个线程的创建函数调用点分别为 i_1 和 i_2，上下文为 c_1'和 c_2'，它们是可达的，即(i_1, c_1')和(i_2, c_2')满足关系 ReachIC。根据函数 FindM 定义，访问内存操作 e_1 和 e_2 所在的函数为 $FindM(e_1)$和 $FindM(e_2)$，并假设 e_1 和 e_2 的上下文为 c_1 和 c_2，而这些函数也是可达的，即满足下列式子：

$$\exists(i_1,c'_1)\in ReachIC \wedge (i_1,c'_1)\Rightarrow^{*}(FindM(e_1),c_1)$$

$$\exists(i_2,c'_2)\in ReachIC \wedge (i_2,c'_2)\Rightarrow^{*}(FindM(e_2),c_2)$$

综上所述，两个线程的访问内存操作的可达性 ReachPairs 定义如下：

$$ReachPairs=\{(e_1,c_1,e_2,c_2)\,|\,(e_1,e_2)\in RWPairs \wedge \exists(i_1,c'_1,i_2,c'_2)\in ReachIC^2:$$
$$(i_1,c'_1)\Rightarrow *(FindM(e_1),c_1)\wedge(i_2,c'_2)\Rightarrow *(FindM(e_2),c_2)\}$$

四元组 ReachPairs(e_1, c_1, e_2, c_2)表示两个线程分别在上下文为 c_1 和 c_2 的函数调用过程中，进行的变量读写操作 e_1 和 e_2 的程序语句可达，并至少其中一个为写操作。式子中 $ReachIC^2$ 表示 ReachIC 的笛卡尔积。

3.5 别名分析

在数据竞争问题中，对于两个线程访问内存变量的操作而言，如果该变量是在堆内存上分配，则需要判断被访问的是否是同一个对象的实例变量；如果该变量是在全局的数据区分配，则需要判断是否为同一个静态或全局变量。为此需要使用别名分析的方法。

别名分析是用来判断哪个程序变量指向哪一块内存。在数据竞争问题中，需要保证两个操作指向同一块内存，即其中一个变量是否为另外一个变量的别名。在数据竞争检测中，可以使用上下文相关的别名分析或上下文无关的别名分析方法，本系统中使用上下文相关的别名分析方法。上下文相关的别名分析是在上下文无关的别名分析的基础上，结合上下文相关的函数调用图，

添加上下文的信息。

Java 程序的字节码指令中，访问堆内存实例变量的操作是 putfield 和 getfield，赋值操作为 assign，访问静态变量的操作为 putstatic 和 getstatic。为研究问题方便，对实例变量的操作使用 store 和 load 代替。通过对字节码指令的分析，提取相关的信息。首先对需要的函数和操作进行抽象，定义如下：

（1）定义局部变量 v 和内存对象 h 的指向关系 VP_0。如果$(v, h, c)\in VP_0$，则表示在程序中存在一条对象产生语句，该语句在某个调用点调用了对象 h 的构造方法，并且此时上下文为 c，即存在语句 v = new H()。

（2）定义对对象属性的写操作 store。存在元素 store(v_1, f, v_2)的含义是在某个函数中存在对 v_1 变量指向对象的成员属性 f 进行写操作的语句，即程序中存在函数语句 $v_1.f = v_2$。

（3）定义对对象属性的读操作 load。存在元素 load(v_1, f, v_2)的含义是在程序中存在对 v_1 指向对象的属性 f 的读操作，即 $v_2 = v_1.f$。

（4）定义对对象属性的赋值操作 assign。存在元素 assign(c_1, v_1, c_2, v_2)的含义是变量 v_1 在上下文环境为 c_1 时可能指向上下文为 c_2 的变量 v_2 所指的对象，即 $v_1=v_2$。由于 assign 操作中可以包括函数调用期间的传递参数动作和返回值的传递，因此存在上下文切换的过程。

（5）定义二元关系 EF(e, f, c)，表示访问操作 e 在上下文为 c 时访问静态变量 f。通过分析 Java 字节码指令 putstatic 和 getstatic，能够提取出访问的静态变量 f。

（6）定义二元关系 EV(e, v, c)表示访问对象属性的操作 e 通过变量 v 进行。如对实例对象的读操作 load 为 v' = v.f，写操作为 v.f = v'，赋值操作为 v = v'。该二元关系主要用来保存变量 v 和访问动作 e 的对应关系。

通过以上操作的定义，数据竞争检测中的别名分析需要求的问题是在程序中任意一个变量 v，判断其是否可能指向的给定的对象 h。别名分析方法对于的处理步骤如下所述。

（1）对于在程序中出现的内存分配指令 new，直接将 VP_0 的元素(v, h)放入集合 VP 中，即 $VP=\{(v, h, c)|(v,h,c)\in VP_0\}$。

（2）如果在赋值 assign 操作中出现 $v_1=v_2$，并且 v_2 指向对象 h，则变量 v_1 也可能指向 h，即 $VP=\{(v_1,h,c_1)|(v_2, h, c_2)\in VP \wedge (c_1, v_1, c_2, v_2)\in assign\}$。

（3）在写入对象 v_1 的属性 f 时，即 $v_1.f = v_2$，并且 v_1 指向对象 h_1，v_2 指向对象 h_2，则对象 h_1 的属性 f 指向对象 h_2。即 $HP=\{(h_1,f,h_2)|(v_1,f,v_2) \in store$

$\wedge$ $(v_1,h_1,c)\in VP$ $\wedge$ $(v_2,h_2,c)\in VP$ }。

（4）如果在读取对象的属性 f 的过程中出现 $v_2 = v_1.f$，并且 v_1 指向对象 h_1，而对象 h_1 的属性 f 又指向对象 h_2，则 v_2 指向对象 h_2。即 $VP=\{(v_2, h_2, c)|(v_1, h_1, c)\in VP \wedge (h_1, f, h_2)\in HP \wedge (v_1, f, v_2)\in load\}$。

根据以上四条规则，分别对每个函数的控制流程图中的字节码指令进行处理，可以得到程序中的所有指向关系 VP 和 HP。为描述问题方便，定义函数关系 PointH，该函数输入访问操作 e，返回在使用该操作过程中可能执行的对象集合，即 $PointH(e, c) = \{h|(v, h, c)\in VP \wedge (e, v, c)\in EV\}$。对于静态变量而言，同样定义 PointS 关系，即 $PointS(e,c)=\{f|(e, f, c)\in EF\}$。

因此，对于堆内存的变量而言，在数据竞争发生时，两个访问操作 e_1 和 e_2 所采用的局部变量 v_1 和 v_2 的所指向的对象集合一定有交集，即 $PointH(e_1, c_1)\cap PointH(e_2, c_2)\neq\varnothing$。

在数据竞争发生时，两个访问操作 e_1 和 e_2 所访问的静态变量一定为同一个静态变量，即 $PointS(e_1, c_1)\cap PointS(e_2, c_2)\neq\varnothing$。

结合两种情况，通过别名分析判断了两个内存的访问 e_1 和 e_2 是否指向同一块内存变量，即满足条件 AliasPairs，即

$$AliasPairs=\{(e_1,c_1,e_2,c_2)|PointS(e_1, c_1)\cap PointS(e_2, c_2)\neq\varnothing \vee PointH(e_1, c_2)\cap PointH(e_1, c_2)\neq\varnothing\}。$$

3.6 锁集合计算

在数据竞争检测中，需要对同步顺序做出判断。如果两个访问操作同时被同一个锁对象保护，则对于任意给定的两个访问操作一定不会发生数据竞争。为此，需要对锁的操作进行分析，定义以下函数和集合。

（1）使用集合 Path 代表在程序中的全部执行路径，集合 Sync 代表锁同步操作，包括加锁和解锁的字节码指令。集合 H 代表所有对象集合。

（2）函数 FindSync 为查找锁操作函数，输入为某个函数调用点或访问内存变量的操作，返回锁操作集合。函数 FindSyncObj 为查找锁对象函数，输入为有序对(s, c)，表示在上下文为 c 时的锁操作语句为 s，输出为对应的锁对象集合。

（3）函数 FindPath 为查找某个线程的以(i, c')为起点，终点为(m, c)的全部路径。该函数是通过在上下文相关的调用关系图 CICM 中，找到以调用点 i 和上下文 c'为起点，能够到达上下文为 c 的函数 m 的全部路径。

（4）函数 FindPathR 的含义是：以某个线程从 main 方法到达的上下文函数调用点(i, c)为起点，能够到达终点(m, c)的全部路径。该函数的目的是找到特定线程的以(m, c)为终点的可以执行的全部路径。这些路径的起点必须满足条件 ReachIC 关系，即保证该线程能够从 main 方法开始能够执行到路径的起点，而起点之后步骤满足 FindPath 函数关系。

以上函数和关系的形式化描述如下，字母 P 代表该集合的子集的集合。

FindSync：$(I \cup E) \rightarrow P(Sync)$

FindSyncObj：$(Sync \times C) \rightarrow P(H)$

FindPath：$(I \times C \times M \times C) \rightarrow P(Path)$，即

FindPath(i,c',m,c)={path| path 满足条件$(i,c') \Rightarrow^{*} (m,c)$}

FindPathR：$(M \times C) \rightarrow P(Path)$，即

$FindPathR(m,c)=\bigcup\{path|\exists i,c':(i,c') \in ReachIC \wedge path \in FindPath(i,c',m,c)\}$

根据以上定义的函数关系，定义某个线程在执行某条路径上保持有锁对象 h 的含义是，满足以下两个关系中任意的一个：

（a）访问操作 e 被 Java 程序中的 synchronized 关键字包围，并且锁变量 l 在上下文 c 中指向锁对象 h；

（b）在该路径执行过程中，存在函数调用点 i''，在上下文为 c''时，该调用点 i''被 synchronzied 关键字包围，并且在上下文为 c''的锁变量 l 指向的锁对象为 h。

形式化描述如下，保持锁的关系 LockHeld 为：

LockHeld：$(Path \times E \times C) \rightarrow P(H)$

$LockHeld(path,e,c)=\{h|\exists s \in FindSync(e):FindSyncObj(s,c)=\{h\}\} \cup$

$\{h|\exists i'',c'':$路径 path 内包括$(i'',c'') \wedge \exists s \in FindSync(i'):$

$FindSyncObj(s,c'')=\{h\}\}$

在数据竞争过程中，两个线程间没有保持相同的锁对象才可能发生，因此，定义两个线程操作的锁集合 UnlockedPairs，定义如下。

$UnlockedPairs=\{(e_1,c_1,e_2,c_2)|\exists path_1 \in FindPathR(FindM(e_1),c_1),$

$\exists path_2 \in FindPathR(FindM(e_2),c_2):LockHeld(path_1,e_1,c_1)$

$\cap LockHeld(path_2,e_2,c_2)=\varnothing\}$

该表达式的含义是，两个线程的访问动作 e_1 和 e_2，存在两条可达的路径，能够到达访问操作所在的函数，并且在此路径过程中两个线程获得的锁对象的集合交集为空集，即不会发生线程互斥现象。

通过以上章节描述的字节码解析、程序流程建模、可达性计算、别名分

析和锁集合计算五个过程，定义数据竞争发生的有序对 RacePairs(e_1,c_1,e_2,c_2) 如下。

$$RacePairs(e_1,c_1,e_2,c_2)=\{(e_1,c_1,e_2,c_2)|\ ReachPairs(e_1,c_1,e_2,c_2)\wedge AliasPairs(e_1,c_1,e_2,c_2)\wedge UnlockedPairs(e_1,c_1,e_2,c_2)\}$$

其中 ReachPairs 保证了两个访问变量的操作可以同时发生并且只是有一个操作为写操作，AliasPairs 保证了访问的是同一块内存变量，UnlockedPairs 保证了在整个路径的调用过程中没有出现锁互斥的现象。因此，多线程的数据竞争问题可以用 RacePairs 来表示。

3.7 本章小结

本章主要是具体详细地介绍了基于 Java 字节码的数据竞争检测方法。首先，通过字节码解析，建立程序的抽象模型，提取关键的集合信息；接着在程序流程建模中，建立每个方法的控制流程图，在方法间构建函数调用关系图，并且结合上下文环境进行标记；然后通过可达性计算，确保两个线程的访问操作可以并发运行，并且其中至少存在一次写操作；通过别名分析，确保访问操作对同一个内存变量发生；最后通过锁集合计算，保证两个线程在访问内存变量的过程中，没有被同一个锁对象保护，从而有效地检测出数据竞争问题。

第 4 章 数据竞争检测工具实现及实验

4.1 系统流程设计

根据上一章描述的数据竞争检测方法，将编译后的 Java 字节码文件作为输入，通过信息提取、程序流程建模、可达性计算、别名分析和锁集合计算之后，检测到数据竞争问题产生的位置。本章将对该检测方法进行实现，系统的整体流程图如图 4-1 所示。

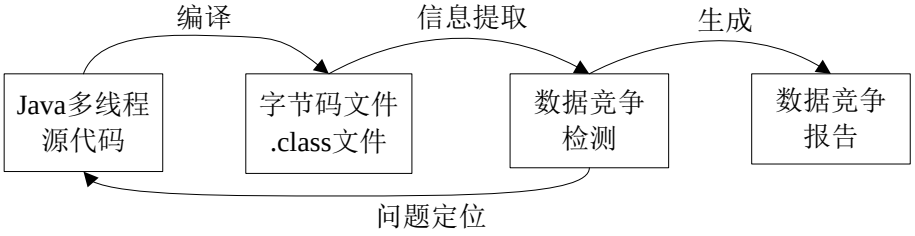

图 4-1 数据竞争检测系统流程图

根据上图所示，将系统的数据竞争检测流程分为三个步骤：

（1）获取 Java 程序字节码文件。对于获取到的 Java 源代码，经过 javac 编译之后得到字节码文件，而 jar 包形式的文件则通过 jar 命令解压后生成相应的字节码文件。

（2）数据竞争检测。根据编译后的字节码文件，经过字节码解析、程序流程建模、可达性计算、别名分析和锁集合计算之后的到数据竞争问题列表，数据以 XML 文件形式保存。在字节码解析过程中，需要分析 class 文件内容，提取基本信息。程序流程建模则需要根据字节码指令流程构建控制流程图和调用关系图。最终经过可达性计算、别名分析和锁集合计算，保证数据竞争发生的条件，成功地检测到数据竞争问题。

（3）数据竞争报告生成。通过数据竞争检测之后，系统得到可能发生数据竞争的位置。但是由于系统采用字节码指令分析，并不是程序源代码中的语句，因此需要将检测到的字节码指令映射到相应的 Java 源代码文件中。在程序分析过程中，记录了每个字节码指令对应的源文件代码行数，因此通过读取该分析结果，能够方便的生成数据竞争报告，便于软件开发人员进行修改源代码。

由于系统中使用的可达性计算、别名分析以及锁集合计算的方法中包括有大量的集合运算。为便于实现，系统中使用基于二叉判定图的 Datalog 语

言[44]来进行算法描述，并通过脚本的形式来运行程序，得到分析结果。整个系统最终以 html 文件的形式来报告数据竞争发生的位置，并包括发生数据竞争的变量名称和函数调用过程等。

4.2 系统功能设计

本系统是基于 Java 多线程程序的字节码的静态检测工具。系统的整个分析过程是：首先通过字节码解析模块产生基本信息文件，保存在 xml 文件中。然后，在字节码文件分析的基础上，使用程序流程模块构建多线程程序的函数内的控制流程图和函数调用关系图。接着，在程序流程建模的基础上，进行可达性计算，遍历函数的调用关系图，观察两线程所进行的访问操作是否可以同时发生；同时，在别名分析模块，对可能进行并发访问操作进行别名分析，判断是否访问同一块内存；在最后一个分析阶段，锁集合计算过程中判断同步方式，确保在数据访问过程中访问内存变量的操作没有被同一个锁对象保护。最终，在使用最终的分析结果生成数据竞争检测报告。具体的系统功能的结构图如图 4-2 所示。

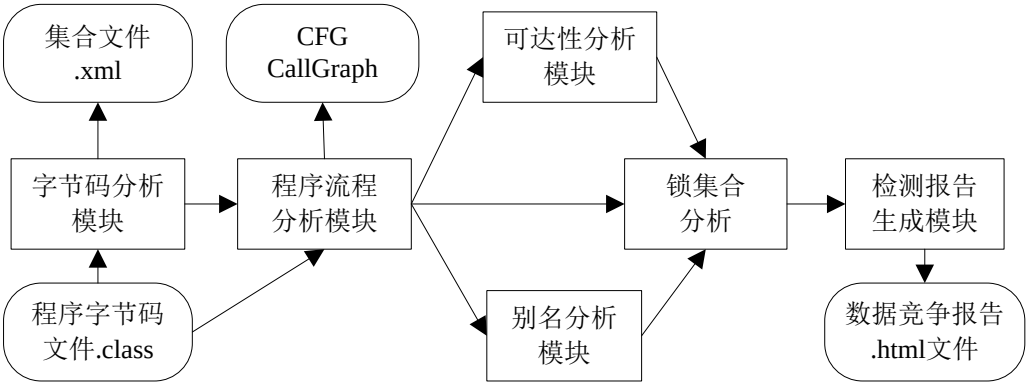

图 4-2 数据竞争检测系统功能结构图

系统总共有六个模块，包括有字节码解析模块、程序流程建模模块、可达性计算模块、别名分析模块、锁集合计算模块以及检查报告生成模块。其中，字节码解析模块主要用来对于字节码文件进行抽象，主要的工作内容包括提取程序的基本信息集合，对字节码指令进行归类，分析字节码文件结构，构造程序的抽象模型等；程序流程建模模块则使用字节码解析模块得到的信息，分析字节码指令流程并生成的 CFG 图和函数调用关系图，生成的这些信息直接被以后的分析模块使用，在整个系统中处于中心的位置；别名分析模块、可达性模块以及锁集合计算模块都是单独的 Task 类的子类，每个模块加

载各自的 Datalog 分析脚本程序，按照顺序运行分析结果；最终程序将分析结果交给检测报告生成模块，显示最终结果。整个系统的数据竞争检测的详细步骤如图 4-3 所示。

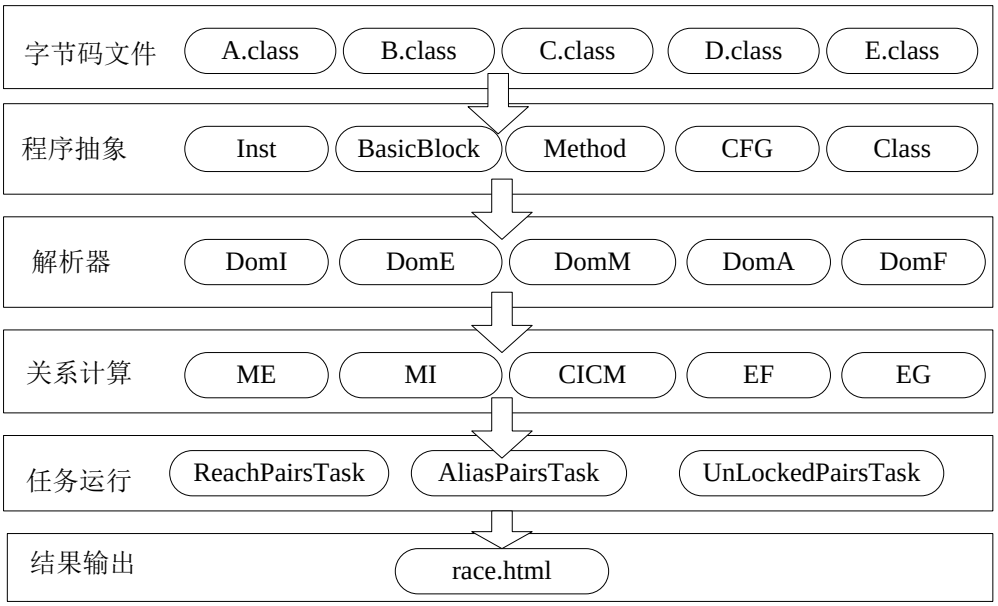

图 4-3 数据竞争检测详细步骤图

在数据竞争检测的步骤中，首先通过程序抽象类分析字节码文件，将文件中的信息建立起抽象模型；然后通过解析器，分析抽象程序模型，将关键信息保存在集合中，并负责计算函数的程序流程图和调用关系图；接着通过关系提取类，计算简单的关系，如在可达性计算中使用的表达函数中访问动作操作的二元关系 ME、表达控制流程的关系 CICM、访问类中属性的关系 EF 等；其次通过运行 Datalog 语言描述的可达性计算、别名分析、锁集合计算的算法，计算后得到可能的数据竞争有序对(e_1, e_2)；最后通过和程序抽象过程中的程序模型结合，找到访问操作所在的语句的位置并输出检测报告。

在整个数据竞争检测过程中，各个步骤起到的作用各不相同。程序抽象步骤和解析器分析步骤属于字节码解析和程序流程建模，这两个步骤的目的是提取和分析数据。关系提取步骤主要是为计算 Datalog 语言描述的算法进行数据准备，如计算得到 MI 关系等。任务运行步骤则通过调用接口进行 Datalog 脚本运算，得到计算结果。

在系统实现方面，集合信息使用 Java 类库中的 Map 和 List 作为数据容器，存放程序的基本信息，包括函数列表 DomM、上下文集合 DomC 等。关键的字节码指令也在程序抽象步骤中形成各自的指令类 Inst，如访问对象属性变量和静态变量的读写操作指令、函数调用指令等。关系的实现使用多维

数组，通过保存 DomM 对象和 DomI 对象的 id 标识符来记录两者的对应关系。在任务运行阶段，每个分析过程继承 ITask 接口并实现加载各自需要的有序对关系，最终完成 Datalog 算法的计算。

4.3 基于 Datalog 的算法设计

4.3.1 可达性计算算法设计

在上一章中描述的可达性计算 ReachPairs 中，判断两个线程对变量的读写操作是否可以同时发生，ReachPairs 表达式如下所述：

$$\begin{aligned}\text{ReachPairs} = \{(e_1,c_1,e_2,c_2) \mid (e_1,e_2) \in \text{RWPairs} \wedge \exists (i_1,c'_1,i_2,c'_2) \in \text{ReachIC}^2 : \\ (i_1,c'_1) \Rightarrow *(\text{FindM}(e_1),c_1) \wedge (i_2,c'_2) \Rightarrow *(\text{FindM}(e_2),c_2)\}\end{aligned}$$

ReachPairs 采用的思路是：（1）首先找到访问操作所在的函数，通过判断包含有访问操作的函数 m 的可达性来代替对访问操作的可达性；（2）判断线程的可达性，即是否存在一条函数调用路径使得某个线程可以达到函数 m；（3）必须保证调用函数 m 的线程也是可达的，即可以从 main 方法开始，到达产生线程的调用函数 java.lang.Thread.start()方法。除此之外，还需要保证两个访问内存的操作 e1 和 e2 中至少有一个为写操作。

在系统实现过程中，首先从字节码文件中提取中相对应的函数集合，如线程类的集合 A，程序函数的集合 M 等；然后通过集合函数的控制流程图，计算一些简单的有序对关系，如保存有函数 m 中的所有访问内存的操作的二元关系 ME(m, e)，对全局变量进行访问的关系 EG(e, g)；接着在一些二元有序关系的基础上，完成对问题的描述，使用 Datalog 语言描述所需要的集合关系，并通过调用命令完成处理过程，将结果保存在中间文件中。使用 Datalog 语言描述的算法实现如下。

算法 4-1 计算访问操作的可达性。

输入关系：cscg: C×I×C×M，EG: E×G，EF: E×F，ME: M×E，
MI: M×I，RWPairs: E×E，wr: E

输出关系：reaches: C×C×M，ReachPairs: A×C×E×A×C×E

规则：

reaches(tmain,ε,M_{main}) (4-1)

reaches(t,c,m):- reaches(t,c',m'), MI(m',i), cscg(c',i,c,m),m≠M_{start} (4-2)

reaches(c, c, M_{start}):- reaches(-,c',m'), MI(m', i), cscg(c',i,c,M_{start}) (4-3)

ReachPairs(t,c,e,t',c',e'):-reaches(t,c,m),reaches(t',c',m'),ME(m,e),ME(m',e'),

EF(e,f),EF(e',f),RWPairs(e,e') (4-4)

ReachPairs(t,c,e,t',c',e'):-reaches(t,c,m),reaches(t',c',m'),ME(m,e),ME(m',e'),
EG(e,g),EG(e',g),RWPairs(e,e') (4-5)

ReachPairs(t,c,e,t',c',e):-reaches(t,c,m),reaches(t',c',m),ME(m,e),wr(e) (4-6)

ReachPairs(t,c,e,t',c ,e):-reaches(t,c,m),reaches(t',c ,m),ME(m,e),wr(e) (4-7)

上述算法中，二元关系 EG(e,g)的含义是访问操作 e 访问静态变量 g，EF(e,f)的含义是访问操作 e 访问对象的属性 f。ME(m,e)的含义是方法 m 包含有访问操作 e，MI(m,i)的含义是方法 m 在调用点 i 有函数调用的指令 invoke。cscg 为函数的上下文敏感的调用关系图。RWPairs(e,e')代表访问操作 e 和 e'至少存在一个写操作。关系 wr(e)表示该操作为写操作。tmain 代表主线程，ε为 main 方法的上下文，M_{start} 为线程启动方法，代表 Thead 中的 start 方法。

公式（4-1）到（4-3），描述的是 reaches 关系，该关系描述函数的可达性。首先在公式（4-1）中表示，主线程 tmain 在 main 方法中是可达的，此时上下文为 ε。接着，公式（4-2）中表示，如果某个线程 t 在某个方法 m'中，存在函数调用点 i，而且以上下文为 c'的函数调用点 i 调用了上下文为 c 的函数 m，那么函数 m 也是可达的。此条件考虑的是不发生线程转化时的可达性，所以此时的方法 m 不能为线程的启动方法 start。最后，公式（4-3）中描述启动线程的调用函数 start 的可达性，即存在函数 m'的调用点 i，使得在方法 start 可达。规则中的横线代表该条件可以被忽略。

关系 ReachPairs 的描述中添加了线程的集合，使用抽象上下文 C 代表当前的线程对象，满足可达性要求的共有四种情况。公式（4-4）描述的是两个不同的线程 t 和 t'在不同的方法 m 和 m'中存在的访问对象的属性 f 的操作 e 和 e'可达，而且其中有一个是写操作。公式（4-5）描述的是两个线程 t 和 t'在不同的方法 m 和 m'中访问同一个静态变量，同样至少存在一个写操作。公式（4-6）表示如果存在两个线程 t 和 t'在不同的上下文环境，但是在同一方法 m 中都可达，并且此方法 m 中存在写操作 e，则该访问操作也是可达的。公式（4-7）则考虑上下文相同时，存在两个线程执行相同的函数的写操作。

通过以上算法描述，通过使用 reaches 关系表达了方法的可达性，进而表达访问操作的可达性。最终结果 ReachPairs 中包括任意两个的线程可能并发的变量访问操作，并且至少存在一个写操作。

4.3.2 别名分析算法设计

在数据竞争检测中的别名分析阶段，主要目的是判断线程访问的静态变

量或对象的属性变量是否为同一块内存区域。别名分析的表达式和使用Datalog表达算法的规则如下所示。

$$AliasPairs(e_1,c_1,e_2,c_2) = \{(e_1,c_1,e_2,c_2)|PointS(e_1,c_1) \cap PointS(e_2, c_2) \neq \varnothing \vee PointH(e_1, c_2) \cap PointH(e_1, c_2) \neq \varnothing\}$$

算法 4-2 别名分析

输入关系：ptsV: C×V×H，EG: E×G，EV：E×V，

ReachPairs：A×C×E×A×C×E

输出关系：AliasPairs：A×C×E×A×C×E

规则：

AliasPairs(t,c,e,t',c',e'):- ReachPairs(t,c,e,t',c',e'), EG(e,g), EG(e',g) (4-8)

AliasPairs(t,c,e,t',c',e'):- ReachPairs(t,c,e,t',c',e'), EV(e, v), ptsV(c, v, o), EV(e',v'),ptsV(c',v',o) (4-9)

上述公式（4-8）和（4-9）的含义是，在可达性的基础上，确保两个访问操作访问同一个静态变量 g 或是访问同一个对象 o。其中二元关系 EV 表示的意思是访问操作访问的变量为 v，如赋值语句中读取 v 的变量 x=v 或是写入变量 v 的内存 v=x。关系 ptsV 的含义是变量 v 指向对象 h，该关系是上下文相关的。公式（4-8）则表示两个线程的访问操作访问同一静态变量 g。公式（4-9）的意思是：如果存在两个访问操作 e 和 e'，分别在上下文环境为 c 和 c'时，访问变量 v 和 v'，并且恰好都指向相同的对象 o。关系 ptsV 的构建方法在 3.5 小结中已经介绍，主要思路是将字节码指令转化为有序关系，采用步步分析的方法表达为指向关系 ptsV，算法采用 Datalog 描述如下。

算法 4-3 指向关系 ptsV 的计算

输入关系：vP_0：V×H，cscg：C×I×C×M，store：V×F×V，load：V×F×V，assign：C×V×C×V

输出关系：ptsV：C×V×H

规则：

ptsV(c,v,h):- vP_0(v,h), cscg(c,h,-,-) (4-10)

ptsV(c,v,h):- assign(c,v,c',v'), ptsV(c',v',h) (4-11)

hP(h,f,h'):- store(v,f,v'), ptsV(c,v,h), ptsV(c,v',h') (4-12)

ptsV(c,v,h'):- load(v,f,v'), ptsV(c,v,h), hP(h,f,h') (4-13)

上述算法中，vP_0 表示使用 new 指令分配对象，并用局部变量 v 指向分配的对象 h。store、load、assign 都是字节码指令，分别表示存储、读取和修改变量。公式（4-10）表示，如果使用 new 指令分配了一个对象 h，而且在函

数调用图中显示调用该对象的构造函数时上下文为 c，则满足(c,v,h)关系。公式（4-11）的含义是，如果在使用 assign 指令时，将上下文为 c 的变量 v 赋值给上下文为 c'的变量 v'，而此时 v'指向对象 h，则变量 v 也执行对象 h。这种情况主要发生在函数调用时参数的传递过程。公式（4-13）的含义是将变量 v 的属性 f 的结果读到变量 v'中，此时如果变量 v 执行对象 h，并且对象 h 的属性 f 指向另一个对象 h'，则(c,v,h')满足 ptsV 关系。

4.3.3 锁集合计算算法设计

锁集合计算是要计算 UnlockedPairs 的元素，其中保持锁的关系 LockHeld 函数由关系 guarded 维护。使用的 Datalog 语言描述的算法如下。

算法 4-4 UnlockedPairs 的计算

输入关系：guarded：A×C×E×H，ReachPairs：A×E×A×C×E

输出关系：UnlockedPairs：A×C×E×A×C×E

规则：

NoRaces(t,c,e,t',c',e'):- ReachPairs(t,c,e,t',c',e'), guarded(t,c,e,o),
guarded(t',c',e',o) (4-14)

UnlockedPairs(t,c,e,t',c',e'):- ReachPairs(t,c,e,t',c',e'), !NoRaces(t,c,e,t',c',e') (4-15)

算法 4-4 表示，如果两个线程的访问变量操作 e 和 e'被同一个锁对象 o 所保护，那么不可能发生数据竞争，因此，在最终结果 UnlockedPairs 中去掉。关系 guarded 保证线程 t 在上下文为 c 时，进行某个访问操作 e 的路径上，保持有锁对象 o。综合以上的算法，可以得到检测数据竞争的整体的 Datalog 算法描述如下。

算法 4-5 数据竞争检测算法

输入关系：UnlockedPairs：A×C×E×A×C×E，AliasPairs ：A×C×E×A×C×E

输出关系：RealRaces：A×C×E×A×C×E

规则：

RealRaces(t,c,e,t',c',e'):- AliasPairs(t,c,e,t',c',e'), UnlockedPairs(t,c,e,t',c',e') (4-16)

由于 AliasPairs 和 UnlockedPairs 都是在可达性的基础上添加的条件，因此，ReachPairs 的可达性要求包含在两者内。所以，只要满足了 AliasPairs 和 UnlockedPairs 的条件，则被认为是会发生数据竞争问题。

4.4 系统模块设计与实现

4.4.1 字节码解析模块

字节码解析模块主要完成的功能是从编译后的 class 文件中提取关键信息，建立抽象程序模型。抽象程序模型主要包括的数据结构是：字节码指令类 Inst、代码块类 BasicBlock、方法 Method 类、控制流程图 CFG 类以及 Class 类等，各个类之间的关系如图 4-4 所示。

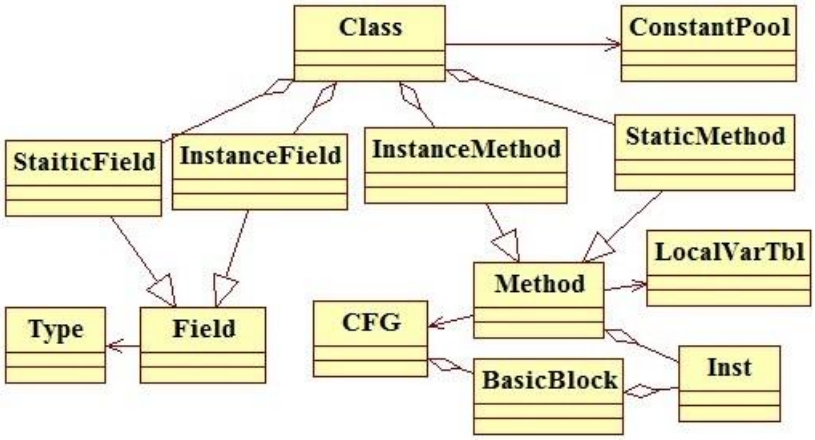

图 4-4 抽象程序模型类图

其中，Class 类则保存了类的静态变量、实例变量、静态方法、动态方法，以及常量表等信息。每个字节码文件至少可以抽象出一个 Class 类。方法 Method 类则记录了方法对应的指令集合、源代码文件中的行号信息表、函数的返回值、局部变量表等，该类中还完成了构建控制流程图 CFG 的操作。Method 类有两个子类，一个是实例方法类 InstanceMethod，另一个是 StaticMethod 静态方法类。指令 Inst 类中包括的属性有标识符 id，该指令的操作码 operator，操作数集合，指令所属的代码块 BasicBlock 的引用等。控制流程图 CFG 类中保存了代码块 BasicBlock 类的集合。BasicBlock 类则包括标识符 id，该代码块所属的方法的引用，代码块中的指令集合以及该代码块的前驱和后继节点等。变量 Field 类保存有执行类型 Type 的引用，该类也存在两个子类，一个是 StaticField 保存了类中的所有静态变量，另一个是 InstanceField 保存了类中的所有非静态的属性变量。

字节码解析模块中，对于指令集合的分析是最关键的步骤，分析过程中需要对每条字节码指令的功能进行判断，将每条功能的指令解析为相应的类，并保存与该指令相关的信息，该模块需要抽象的指令的如表 4-1 所示。

表 4-1 字节码指令与 Inst 子类对应关系表

Inst 子类	对应的字节码指令	描述信息
ObjValAsgnInst	new	分配对象操作
InvkInst	invkvirtual、invkstatic、invkspecial、invkinterface	调用函数指令，包括静态方法、实例方法等
ObjVarAsgnInst	assign	局部变量赋值 v=v'形式
StatFldRefInst	putstatic getstatic	读写静态变量
InstFldRefInst	putfield getfield	读写对象的属性变量
AcqLockInst	monitorenter	获取对象锁的操作
RelLockInst	monitorexit	是否对象锁的操作

表中列出了在别名分析过程中主要关注的指令对应的子类，这些类中都封装了对应操作的关键信息。抽象所有的指令类之后，生成的类图如图 4-5 所示。

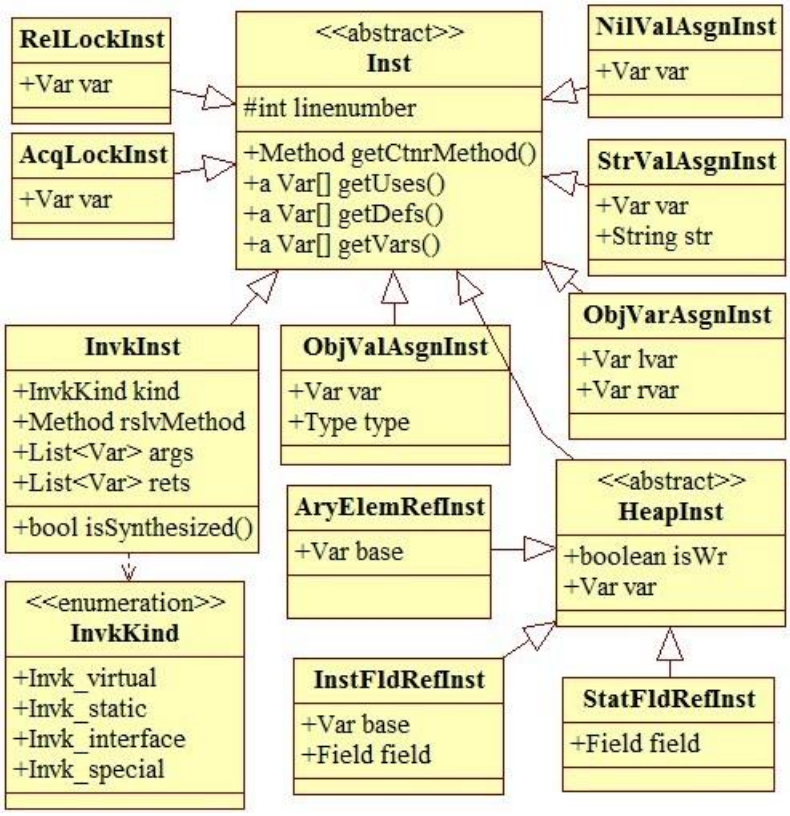

图 4-5 字节码指令类类图

从上图可以看出，对局部变量赋值的 assign 指令类 ObjVarAsgnInst 类中，需要保存左右操作的局部变量的引用；在分配堆对象指令 new 指令中，需要

保存新对象的类型和指向的变量；在访问静态变量的 putstatic、getstatic 指令中，需要保存读写操作类型、进行操作的变量和被访问的属性；在访问实例变量的 putfield、getfiled 指令中，需要保存对访问对象的引用、操作的变量以及被访问的属性等；在调用函数的 invk 指令中，需要保存调用函数的类型、被调用者的方法、调用者的参数等。

图 4-5 中所有具体的指令类都继承自抽象的基类 Inst。抽象基类 Inst 中保存了该字节码指令在源代码文件中处于的行号，便于以后产生检测报告时使用。Inst 类的 getImmediateCtnrMethod 方法返回该指令所在的函数 Method，此外，可以根据 getUses 得到语句中读操作的变量集合，getDefs 返回语句写操作的变量集合。此外，还有一些其他操作，如对数组元素的访问指令对应的 Inst 子类 AryElemInst 等。

4.4.2 程序流程建模模块

程序流程建模模块主要任务是构建每个函数的控制流程图，生成函数调用图，并利用字节码解析模块提供的信息封装成相应的 Domain 类。该模块中主要的执行类是 TaskParser 类，该类完成了控制流程图 CFG 对象的构建和 Domain 对象的封装，该类的执行流程过程如下。

（1）遍历程序中所有的 Method 对象，调用其 buildCFG 方法中，构建该方法的 CFG 对象。

（2）针对当前的方法，遍历其中的所有字节码指令和局部变量表，并根据指令类型封装到相应的 Domain 类中。例如，将静态变量的读写对象 StatFldRefInst 封装到 DomE 类中；将函数调用对象 InvkInst 封装到 DomI 类中；将加锁对象 AcqInst 封装到 DomL 类中；将局部变量对象封装到 DomV 类；将产生新对象的 ObjValAsgnInst 对象封装在 DomH 类等。

（3）遍历字节码解析模块的类 Class 对象，提取每个类的方法集合和属性集合，将 Field 和 Method 封装到 DomF 和 DomM 类中，同时将 Field 的类型和 Class 的类型都加入到 DomT 类。

以上步骤中，当处理处理复杂的 Domain 类，如抽象线程集合 DomA 类是，需要在构建一些 Dom 类的基础上，查找符合要求的组合。如 DomA 类的构建中，需要遍历 DomI 对象，如果调用方法为 start 方法，将其对应的 DomH、DomC 和 DomM 对象组合存放在 DomA 中。在代码实现过程类中，使用 HashMap 和 List 的对象保存相关的对象，并提供查找和添加功能，形成的类图如图 4-6 所示。

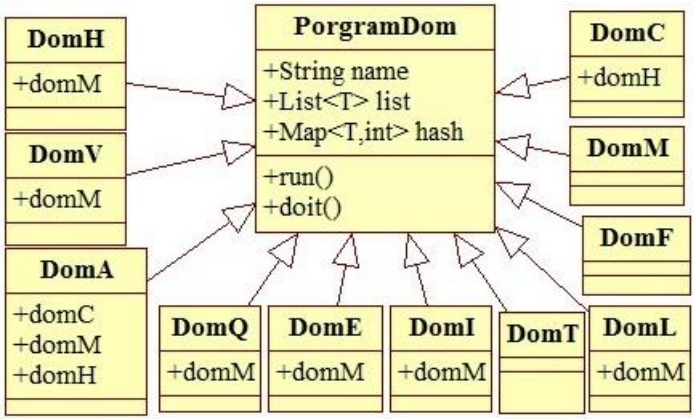

图 4-6 基本 Domain 类类图

在构建上下文相关的函数调用关系图中，需要在关系提取步骤的基础上进行。上下文相关的函数调用图表示为四元组 CICM (c:C,i:I,c':C,m:M)，该表达方式的含义是在上下文关系为 c 且函数调用语句在调用点 i 时，存在一条执行路径，调用上下文环境为 c'的方法 m。计算该关系之前，需要首先利用 3.2.2 节的方法生成函数调用图，并提前计算一些关系。其中，非静态方法的 CICM 关系计算过程如下。

（1）使用 3.2.2 小节方法计算函数调用图 call graph。

（2）计算关系 reachCI。关系 reachCI(c:C,i:I)代表在上下文为 c 时，包含有调用点 i。通过计算 MI(m,i)和 ReachCM(c,m)，求交集得到 reachCI。其中关系 MI 的含义是在方法 m 中包括调用点 i，ReachCM 的含义是在上下文为 c 时方法 m 是可达的。

（3）计算关系 CIH。关系 CIH(c:C,i:I,o:H)代表在上下文 c 时，调用关系中在调用点 i 的调用者对象可能为对象 o。Java 中类的非静态方法的调用形式为 d=d'.method()，其中 d'为调用者对象的引用变量。在得到方法的调用者的引用变量 d'之后，分析此时上下文环境 c，得到 v'指向的对象 o。

（4）计算关系 CtxtICM。关系 CtxtICM(i:I,o:H,m:M)代表在调用点 i，可能调用的是对象 o 的方法 m。

（5）综合以上四个步骤，可以得出，当上下文为 c 时，调用点 i 所调用的方法 m 是可达的，而且调用者变量 v'指向的对象为 o，并且在调用点 i 执行的方法是对象 o 的方法 m。因此将关系 reachCI、CIH 和 CtxtICM 综合考虑，得到 CICM 为(c,i,o,m)。使用 Datalog 语言描述的计算过程规则为：

reachCI(c,i) :- MI(m,i), ReachCM(c,m) (4-17)

CIH(c,i,o) :- IinvkArg0(i,v), CPointTo(c, v,o) (4-18)

CICM (c,i,o,m) :- reachCI(c,i), CIH(c,i,o), CtxtICM(i,o,m) (4-19)

其中，IinvkArg0 关系为得到调用点 i 的第 0 个参数的变量 v，即调用者对象的引用变量。CPointTo 关系则保证变量 v 在上下文为 c 时指向对象 o。通过以上的执行过程，可以在函数调用关系图 CG 上引入上下文信息，完成 CICM 关系的构建。

4.4.3 其他模块设计

在数据竞争检测过程中，经过字节码解析模块和程序流程建模模块，已经建立程序抽象模型，并且将控制流程图和函数调用关系图计算完成，将基本信息封装为 Domain 类。剩下的模块如可达性计算模块、别名分析模块以及锁集合计算模块，都是在这两个模块的基础上，通过计算用到的相应的关系，执行 Datalog 描述的算法，完成各自的任务。其中，ReachPairsTask 类、AliasPairsTask 类和 UnlockedPairsTask 类分别为相应各自模块的主类，这些类都继承自 Task 接口，都被 DataRaceAnalysis 类引用，这三个模块的运行步骤与 DataRaceAnalysis 类运行步骤相似。通过以上的模块分析，得知完成数据竞争检测还需要进行以下几个步骤。

（1）与数据竞争检测相关的关系计算。数据竞争检测过程需要计算其他的一些关系，如 MI、EG 等关系。例如表示在方法 m 中包含有调用点 i 的关系 MI 的计算过程如表 4-2 所示。

表 4-2 计算关系 MI 算法

算法 4-6 计算关系 MI 算法
输入：DomM 集合，DomI 集合
输出：(mid, iid)有序对集合
1. for each iIdx in DomI 2. inst = DomI.getInst(iIdx) 3. method = inst.getMethod() 4. mid = DomM.getId(method) 5. add(mid, iIdx)

从 MI 的计算过程中可以看出，简单的二元关系的计算过程通常是在相应的组成的 Domain 元素中遍历。由于在相应的 Domain 类中，使用 Map 的数据结构，该结构不仅保存了对应集合的对象，还保存了对应对象的标识符 id。因此，对于二元关系 MI 而言，需要保存的是(mid, iIdx)的集合，mid 是指 DomM 中保存的包含有某调用点的 Method 对象的标识符，而 iIdx 是指调

用函数点集合 DomI 中保存的 InvkInst 对象的标志符。

（2）对于复杂的关系计算过程中，需要一些简单关系的计算结果。例如，在可达性计算过程 ReachPairs 的计算过程中，需要的 EG、ME 等简单关系的计算结果。

（3）调用 Executor 类最终执行 Datalog 算法。在将所有需要提取计算的关系完成后，就可以调用接口，完成整个数据竞争检测的算法。图 4-7 展示了整个数据竞争检测执行过程中需要的部分类的关系图。

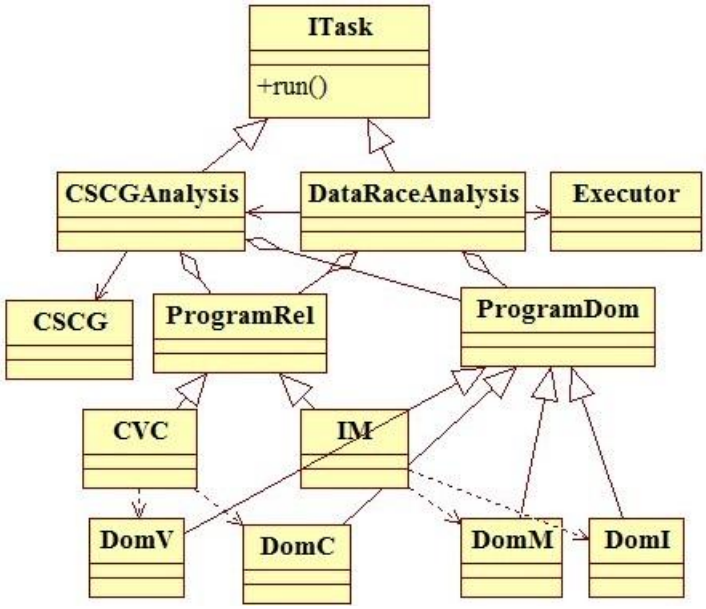

图 4-7 数据竞争检测过程中部分类图

从图 4-7 中可以看出，在完成 Domain 类的抽象的基础上，首先进行简单的二元关系的计算，如图中的 CVC、MI 关系等。接着，对于复杂的关系使用 ITask 类的子类完成，如图中的 CSCGAnalysis 类，该类完成计算上下文相关的函数调用关系图 CSCG 图。CSCGAnalysis 类中需要执行 Datalog 语言描述的脚本，因此需要使用一些简单的关系的计算结果，最终生成 CSCG 对象。最后，对于 DataRaceAnalysis 类而言，需要将整个计算过程中需要全部关系计算完成之后，再开始自己的检测算法。在实现上，DataRaceAnalysis 类将任务委托给 Executor 类去执行，调用过程中传递给 Executor 类自己所使用的算法实现的文件的名称。Executor 类完成之后，将结果保存在 xml 文件中。最后，由检测报告生成模块根据结果生成检测报告文件。

数据竞争检测报告生成模块则是通过读取数据竞争检测的结果文件 race.xml，将此 XML 文件作为输入数据，将描述数据报告格式的 XSL 文件

作为标准，通过利用 saxon 第三方软件包提供的 API 接口，生成数据竞争检测报告的 html 文件。

4.5 数据竞争检测实验

4.5.1 实验内容

实验过程中的系统环境是 Windows7 64 位操作系统，JDK 版本为 1.7，开发环境为 Eclipse4.2，测试用例使用 ant 工具运行，物理机内存 8G。

为检测数据竞争问题，需要找出一些包含有软件缺陷的测试用例。在实验过程中，使用公共测试程序集合 pjbench[47]，该测试集中包括可能发生数据竞争的程序代码。本实验用到的测试用例如表 4-3 所示。

表 4-3 测试用例信息列表

测试用例	文件数目	代码行数	描述信息
test1	1	34	读写全局静态变量
test2	3	87	读写在实例变量
elevator	2	537	开源实现的电梯调度算法
tsp	4	776	解决旅行商问题的开源算法实现
weblech	13	1879	Java 实现的网站下载工具
cache4j	22	3897	Java 对象的缓存框架

测试用例 test1 和 test2 是单独编写的用来测试分别在静态全局变量和对象中的实例变量的访问的数据竞争问题。elevator 和 tsp 是开源算法的实现，weblech 和 cache4j 是小型的 Java 第三方工具框架。其中，测试用例 test2 中两个线程同时对父类对象中的共享变量进行先读后写的操作，可能发生的数据竞争应该有 3 对，即两个子线程直接的两个写操作，以及一个线程执行读操作而另外一个线程执行写操作，检测数据竞争的检测报告如图 4-8 所示。

```
Thread    Memory Access
Dataraces on test.Worker.i
1.1  test.Worker$1.run()
     test.Worker.access$112(int) (Rd)
1.2  test.Worker$1.run()
     test.Worker.access$112(int) (Wr)
1.3  test.Worker$1.run()
     test.Worker.access$112(int) (Wr)
```

```
Thread    Memory Access
Dataraces on test.Worker.i
1.1  test.Worker$2.run()
     test.Worker.access$120(int) (Wr)
1.2  test.Worker$2.run()
     test.Worker.access$120(int) (Rd)
1.3  test.Worker$2.run()
     test.Worker.access$120(int) (Wr)
```

图 4-8 数据竞争检测报告示意图

在上图中，详细罗列出了每个线程的执行访问的操作以及读写操作类型，而且标明了发生数据竞争的访问变量和每个线程进行的读写操作所在的函数。从上图可以清楚地看出，总共有 3 个数据竞争发生，符合预期结果。

4.5.2 实验结果分析

由于本系统采用基于 Java 字节码的分析方法，为了跟基于源代码的数据竞争检测方法作比较，将测试用例过程所花费的时间与开源的数据竞争检测工具 chord[46]作比较。chord 工具的分析过程中使用 Soot 框架来实现对程序流程的建模，并且使用的程序控制流程图是基于源代码语言构建的。所有测试用例的使用时间如图 4-9 所示，纵轴时间单位为秒。

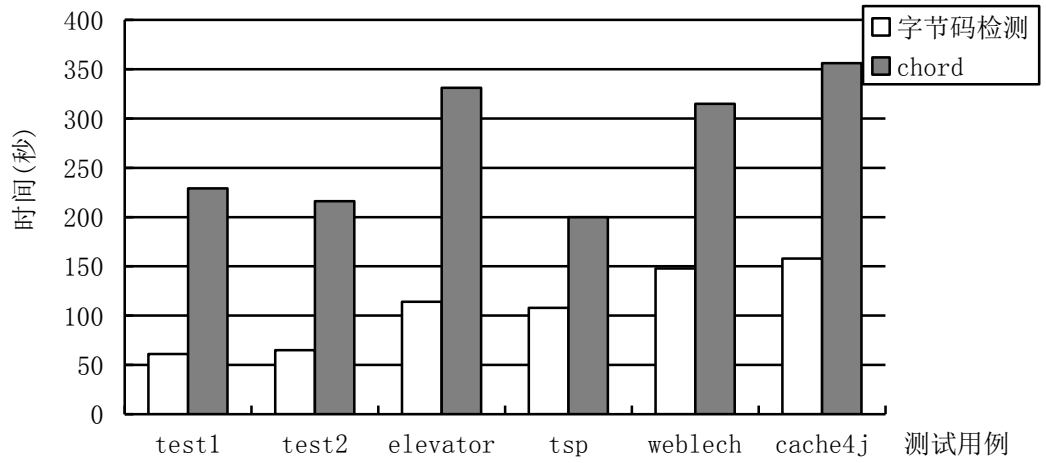

图 4-9 测试用例使用时间数据图

从上图的使用时间可以看出，使用 Java 字节码解析比使用源代码分析的平均使用时间缩短一半以上，从而有效地证明了使用字节码解析技术具有更高的效率。字节码分析方法花费较少时间的主要原因是：在程序分析期间，多线程程序的关键信息可以直接从字节码文件中获取，节省了对 Java 程序的语法分析和语义分析过程，从而避免了复杂的解析步骤，减少了检测时间。此外，基于 Java 源代码的程序分析中，需要对 Java 程序的语言细节进行处理，包括对类的成员变量的定义、父类接口的提取等，需要分析语言的关键字，特殊的方法调用和程序语句的类型。然而字节码文件的信息格式固定，字节码指令操作简单明确，不需要考虑语言细节的实现，节省了较多的程序分析时间。

本系统中使用的是上下文敏感的数据竞争检测方法，主要是通过构建控制流程图和函数调用关系图，使用别名分析等静态检测技术基于字节码指令

分析数据竞争问题。通过使用实现检测方法的系统，检测表 4-3 中的六个测试用例程序，得到的实验结果如表 4-4 所示。

表 4-4 数据竞争检测实验结果表

测试用例	Real	Total	Right	Chord	RightC
test1	1	1	1	1	1
test2	3	3	3	3	3
elevator	0	0	0	0	0
tsp	0	2	0	0	0
weblech	4	4	4	4	4
cache4j	2	7	2	8	2

表 4-4 中的 Real 表示某个测试用例中原本包含的数据竞争的数目，Total 列表示使用字节码分析工具得到的数据竞争数量，Right 列表示正确检测数据竞争的数目，Chord 列和 RightC 列分别表示工具 chord 可以检测到的数据竞争总数和正确的数目。从以上实验结果可以看出，当程序规模较小时，能够保持有较高的检测正确率，错误预报的情况比较少。此外该工具和 chord 工具都能检测出全部的数据竞争位置，也证明了静态检测方法能够检测程序执行的全部路径，不存在漏报的特点。在对开源软件的测试过程中可以发现，当程序逻辑比较复杂且规模比较大时，静态检测会有错误预报出现。

基于字节码的数据竞争检测工具在检测过程中出现的错误预报的主要原因有两个：一是对锁同步之外的其他同步方式缺乏分析；二是对数组元素的分析不准确。由于本系统中主要考虑关键字 synchronized 的锁同步方式，因而当测试用例 tsp 中使用了 join 同步方式时，会产生错误预报，如图 4-10 所示。而对于数组元素的处理需要运行时的索引信息，因此静态检测方法不可能根本消除数组元素的错误预报。

Tsp.java : main()

```
for (....) {
   thread[i] = new TspSolver();
   thread[i].start();
}
for(....) {
   thread[i].join();
}
....
System.out.println(TspSolver.MinTourLen);
```

TspSolver.java : run()

```
public void run() {
   .....
   TspSolver.MinTourLen += 1;
   .....
}
```

图 4-10 测试用例中使用的 join 同步代码示意图

图 4-10 中线程类 TspSolver 中的执行方法 run 中存在对全局的静态共享变量 MinLonLen 的写操作，而在主线程 main 方法中存在对该变量的读操作。由于只考虑 synchronized 的锁同步方法，对于 main 方法中出现的 join 同步不做处理，因此将原本会发生线程等待的 join 方法当做普通系统方法处理，产生了错误的数据竞争预报。

此外，该工具除了可以检测数据竞争外，也可以方便地对其他并发程序的缺陷进行检测，如死锁。工具设计时考虑了程序的扩展性，检测死锁的任务类 DeadlockAnalysis 需要从 ITask 继承，并且检测过程采用与检测数据竞争任务类 DataRaceAnalysis 相似的实现：首先通过字节码解析、程序流程建模建立程序的抽象模型，计算简单的函数和关系，然后建立死锁检测方法，并使用 Datalog 描述死锁检测方法；最后通过运行 DeadlockAnalysis 的方法，调用 Executor 类完成死锁检测的任务。死锁检测结果如表 4-5 所示。

表 4-5 死锁检测实验结果表

测试用例	死锁数目	检测数目	正确数目
weblech	22	22	22
cache4j	0	0	0

测试用例 weblech 和 cache4j 中包括了较多的锁操作，因此选择它们作为死锁检测的测试用例。从表 4-5 中可以看出，该工具能够检测出全部的死锁发生位置，具有较高的正确率。

4.6 本章小结

本章在基于字节码数据竞争检测方法的基础上，首先完成了整个系统的模块功能分析和流程设计，并对检测方法中的可达性计算、别名分析和锁集合计算方法使用 Datalog 语言进行了描述，完成了对算法的设计；接着在已有设计的基础上，进一步对模块的功能和实现进行了分析，并完成了字节码解析模块、程序流程模块以及 DataRaceAnalysis 类的具体流程分析；最终完成了整个系统的实现，并且针对测试用例进行了实验。实验结果表明，采用基于 Java 字节码的数据竞争检测方法花费的检测时间较少，而且能够检测出较多的数据竞争问题。

结　论

并发程序设计是目前软件工程中常用的开发技术，而多线程程序是实现并发程序最常用的技术。但是由于多线程程序开发难度比较大，而且在运行期间程序执行的不确定性，使得多线程程序容易发生问题。本文主要是针对 Java 多线程程序的数据竞争问题，提出一种基于字节码的数据竞争检测方法，并通过采用别名分析、数据流分析等程序分析方法完成了实现检测方法的系统。本文所提出的数据竞争检测方法，充分考虑了数据竞争产生的条件，通过字节码解析、程序流程建模等步骤，可以有效地检测数据竞争问题。本文的主要工作和创新之处如下：

（1）针对 Java 字节码指令，提出了构建函数内的控制流程图和函数间调用关系图的分析方法。通过对 Java 字节码指令的分析，建立程序抽象模型，并提取关键的程序信息，为数据竞争检测提供了基础。

（2）结合别名分析和 Datalog 程序分析方法，提出了一种数据竞争检测方法。该方法通在控制流程图和函数调用关系图的基础上，通过可达性计算、别名分析和锁集合计算，有效地检测出数据竞争问题。

（3）对提出的数据竞争检测方法进行了系统模块设计和流程分析，并进行了系统实现。在采用多个测试用例的对比实验发现，基于 Java 字节码的数据竞争检测方法具有较高的执行效率，同时能够检测出较多的数据竞争问题。

数据竞争的检测问题是程序分析领域一个热点问题，现阶段以及有很多的检测方法和工具。但是，任何一种检测方法都很难有效地快速准确地进行检测。本文提出的基于 Java 字节码的数据竞争检测方法具有一定的实际意义，但是还是存在错误预报的问题，因而还有很多问题需要进一步的研究。例如，对于其他同步机制的支持，如何进一步地降低错误预报等。

参考文献

[1] Grogono P, Shearing B. Concurrent Software Engineering: Preparing for Paradigm Shift[C]//Proceedings of the 2008 C3S2E conference. New York:ACM Press, 2008: 99-108.

[2] Zeng W, Zhao J, Liu M. Several Public Commercial Clouds and Open Source Cloud Computing Software[C]//Computer Science & Education (ICCSE), 2012 7th International Conference on. New York: IEEE, 2012: 1130-1133.

[3] Bykov S, Geller A, Kliot G, et al. Orleans: Cloud Computing for Everyone[C]// Proceedings of the 2nd ACM Symposium on Cloud Computing. New York: ACM Press, 2011: 111-126.

[4] Abdelqawy D, Kamel A, Omara F. A Survey on Testing Concurrent and Multithreaded Applications Tools and Methodologies[C]//The International Conference on Informatics and Applications (ICIA2012). Washington:The Society of Digital Information and Wireless Communication, 2012: 458-470.

[5] Short T L. Friend This: Why Those Damaged During the Facebook IPO Will Recover Nothing from NASDAQ[J]. Washington and Lee Law Review, 2014(71): 1519-1569.

[6] 眭俊华,刘慧娜,王建鑫,秦庆旺. 多核多线程技术综述[J]. 计算机应用, 2013 (33): 239-242.

[7] Lee E A. The Problem with Threads[J]. Computer, 2006, 39(5): 33-42.

[8] Erickson J, Musuvathi M, Burckhardt S, et al. Effective Data Race Detection for the Kernel[C]//USENIX Symposium on Operating Systems Design and Implementation(OSDI). New York: ACM Press, 2010: 1-16.

[9] 霍玮,于洪涛,冯晓兵,张兆庆. 静态检测中断驱动程序的数据竞争[J]. 计算机研究与发展, 2011(12)：2290-2299.

[10] Engler D, Ashcraft K. RacerX: Effective, Static Detection of Race Conditions and Deadlocks[C]//ACM SIGOPS Operating Systems Review. New York:ACM Press, 2003: 237-252.

[11] Flanagan C, Freund S N. Type-based Race Detection for Java[C]//ACM SIGPLAN Notices. New York: ACM Press, 2000: 219-232.

[12] Elmas T, Qadeer S, Tasiran S. Precise Race Detection and Efficient Model Checking Using Locksets[R]. Washington: Microsoft Tech Report, 2006.

[13] Kahlon V, Yang Y, Sankaranarayanan S, et al. Fast and Accurate Static Data Race Detection for Concurrent Programs[C]//Computer Aided Verification. Berlin: Springer Berlin Heidelberg, 2007: 226-239.

[14] Ferrara P. A Generic Static Analyzer for Multithreaded Java Programs[J]. Software: Practice and Experience, 2013, 43(6): 663-684.

[15] Ferrara P. Checkmate: A Generic Static Analyzer of Java Multithreaded Programs[C]//Software Engineering and Formal Methods, 2009 Seventh IEEE International Conference on. New York: IEEE, 2009: 169-178.

[16] Naik M, Aiken A. Conditional Must Not Aliasing for Static Race Detection[J]. ACM SIGPLAN Notices, 2007, 42(1): 327-338.

[17] Flanagan C, Freund S N. FastTrack: Efficient and Precise Dynamic Race Detection[C]//ACM Sigplan Notices. New York: ACM, 2009, 44(6): 121-133.

[18] Savage S, Burrows M, Nelson G, et al. Eraser: A Dynamic Data Race Detector for Multithreaded Programs[J]. ACM Transactions on Computer Systems (TOCS), 1997, 15(4): 391-411.

[19] Yu Y, Rodeheffer T, Chen W. Racetrack: Efficient Detection of Data Race Conditions Via Adaptive Tracking[C]//ACM SIGOPS Operating Systems Review. New York: ACM, 2005, 39(5): 221-234.

[20] Xie X, Xue J, Zhang J. Acculock: Accurate and Efficient Detection of Data Races[J]. Software: Practice and Experience, 2013, 43(5): 543-576.

[21] Schimmel J, Molitorisz K, Tichy W F. An Evaluation of Data Race Detectors Using Bug Repositories[C]//Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on. New York: IEEE, 2013: 1361-1364.

[22] 吴萍,陈意云,张健. 多线程程序数据竞争的静态检测[J]. 计算机研究与发展, 2006 (2): 329-335.

[23] 简道红. 多线程程序数据竞争静态检测方法研究[D]. 大连：大连理工大学, 2013.

[24] 富浩,蔡铭,董金祥,金星,龚宜. 基于锁集合算法的增强型数据竞争检测方法[J]. 浙江大学学报(工学版), 2009(02): 328-333.

[25] 章隆兵,张福新,吴少刚,陈意云. 基于锁集合的动态数据竞争检测方法[J]. 计算机学报, 2003(10): 1217-1223.

[26] 宋东海,贲可荣,张志祥. 一种基于类的Java多线程程序数据竞争静态检测算法[J]. 计算机工程与科学, 2014(2): 233-237.

[27] Banerjee U, Bliss B, Ma Z, et al. A Theory of Data Race Detection[C]//Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging. New York: ACM, 2006: 69-78.

[28] 周志明. 深入理解Java虚拟机[M]. 北京:机械工业出版社, 2013: 360-384.
[29] Rajkumar Buyya. Object Oriented Programming with Java: Essentials and Applications[M]. New Delhi: Tata McGraw Hill Press, 2009: 364-386.
[30] 黄俊飞. 一种基于代码静态分析的数据竞争检测方法及系统[P]. 中国: 20101010622730. 2013-07-03.
[31] 丁志义,宋国新,邵志清. 类型系统与程序正确性问题[J]. 计算机科学,2006 (1): 141-143.
[32] 李梦君,李舟军,陈火旺. 基于抽象解释理论的程序验证技术[J]. 软件学报, 2008(1): 17-26.
[33] Ben-Ari M. A Primer on Model Checking[J]. ACM Inroads, 2010, 1(1): 40-47.
[34] Lamport L. Time, Clocks, and the Ordering of Events In A Distributed System[J]. Communications of the ACM, 1978, 21(7): 558-565.
[35] Pugh W, Lindholm T. JSR-133: Java Memory Model and Thread Specification, final release[J]. 2004(1):1-4.
[36] Kahlon V, Sankaranarayanan S, Gupta A. Static Analysis for Concurrent Programs with Applications to Data Race Detection[J]. International Journal on Software Tools for Technology Transfer, 2013, 15(4): 321-336.
[37] Artho C, Biere A. Combined Static and Dynamic Analysis[J]. Electronic Notes in Theoretical Computer Science, 2005(131): 3-14.
[38] Kasikci B, Zamfir C, Candea G. RaceMob: Crowdsourced Data Race Detection[C]//Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. New York: ACM, 2013: 406-422.
[39] Kasikci B, Zamfir C, Candea G. CoRD: A Collaborative Framework for Distributed Data Race Detection[C]//Proceedings of the Eighth USENIX conference on Hot Topics in System Dependability. Berkeley:USENIX Association, 2012: 4-10.
[40] 董渊,王生原,张丽伟,朱允敏,杨萍. 一种用于字节码程序模块化验证的逻辑系统[J]. 软件学报, 2010(12): 3056-3067.
[41] 倪程,李志蜀. 基于数据流的Java字节码分析[J]. 微计算机信息, 2009(12): 231-232.
[42] 丁斌,张志祥. C语言的别名分析方法研究[J]. 计算机与数字工程, 2011(02): 60-63.
[43] 曾亮. 知识库语言—Prolog与Datalog[J]. 软件导刊, 2009(12): 58-60.
[44] Alpuente M, Feliu M A, Joubert C, et al. Using Datalog and Boolean Equation Systems for Program Analysis[C]//Formal Methods for Industrial Critical Systems. Berlin: Springer Berlin Heidelberg, 2009: 215-231.

[45] Allen F E, Cocke J. A Program Data Flow Analysis Procedure[J]. Communications of the ACM, 1976(19): 137-147.

[46] Alowibdi J S, Stenneth L. An Empirical Study of Data Race Detector Tools[C]//Control and Decision Conference (CCDC), 2013 25th Chinese. New York: IEEE Press, 2013: 3951-3955.

[47] Yu M, Yoo S K, Bae D W. SimpleLock: Fast and Accurate Hybrid Data Race Detector[C]//2013 Internatinal Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT). New York: IEEE Press, 2013: 50-56.

攻读硕士学位期间发表的论文及其他成果

[1] 黄荷姣, 顾崇林, 史鹏宙, 石帅. 基于树回归的虚拟化云平台能耗测量方法及系统[P]. 专利申请号: 201410592305.6, 已受理.

哈尔滨工业大学学位论文原创性声明和使用权限

学位论文原创性声明

本人郑重声明：此处所提交的学位论文《基于 Java 字节码的多线程数据竞争检测方法研究及工具实现》，是本人在导师指导下，在哈尔滨工业大学攻读学位期间独立进行研究工作所取得的成果，且学位论文中除已标注引用文献的部分外不包含他人完成或已发表的研究成果。对本学位论文的研究工作做出重要贡献的个人和集体，均已在文中以明确方式注明。

作者签名：史鹏宝　　日期：2014 年 12 月 30 日

学位论文使用权限

学位论文是研究生在哈尔滨工业大学攻读学位期间完成的成果，知识产权归属哈尔滨工业大学。学位论文的使用权限如下：

（1)学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文，并向国家图书馆报送学位论文；（2）学校可以将学位论文部分或全部内容编入有关数据库进行检索和提供相应阅览服务；（3）研究生毕业后发表与此学位论文研究成果相关的学术论文和其他成果时，应征得导师同意，且第一署名单位为哈尔滨工业大学。

保密论文在保密期内遵守有关保密规定，解密后适用于此使用权限规定。

本人知悉学位论文的使用权限，并将遵守有关规定。

作者签名：史鹏宝　　日期：2014 年 12 月 30 日

导师签名：黄荷叶　　日期：2014 年 12 月 30 日

致　谢

两年多的硕士生活即将结束，在此谨向给予我无私帮助的老师和同学表示衷心的感谢。

研究生期间的研究工作的顺利完成，首先应该感谢我的导师黄荷姣老师。在整个硕士学习期间，黄老师给予我非常多的帮助，从研究论文的选题、研究内容的选择、研究方案的确定直到论文的书写和审阅，都给与我充分的指导和帮助；黄老师以渊博的专业知识、严谨的治学态度使我有充分的信心和勇气去面对研究课题过程中的各种困难。在两年多的学习过程中，我时刻感受到黄老师认真负责的态度。黄老师教会了我科学研究的方法，这将对我今后的工作和学习带来极大的影响。在此，谨向黄老师表示崇高的敬意和感谢。

感谢贾小华老师和杜宏伟老师，在每周的例会上贾老师和堵老师耐心地答疑解惑，给出宝贵的建议，帮助我克服了研究过程中的问题。

感谢云计算与移动互联网实验室的所有师兄师姐，尤其是刘春颜和顾崇林师兄，他们热心的帮助和耐心的指导，使我的研究工作可以顺利完成，感谢他们一直的陪伴。

感谢所有实验室同学们，与你们在一起的日子是最快乐的日子，感谢石帅、王会会、芦浩、谷浩、程楹楹、彭屾屾、罗海铭、张婧和陈理仕等，是你们的帮助和鼓励深深影响了我。

感谢我的父母和家人，你们永远是我前进的动力。