·体系结构与软件技术·　　文章编号：1000-3428(2016)11-0083-06　　文献标志码：A　　中图分类号：TP391

基于字节码插桩的多线程调试工具

周一耒[a]，于海波[a]，钟　浩[b]

（上海交通大学 a. 软件学院；b. 电子信息与电气工程学院，上海 200240）

摘　要：在软件演化过程中，代码的调试是不可或缺的重要环节。对于多线程程序，线程的交错和调度是非确定的，不同的线程调度会产生不同的结果，程序员较难调试多线程程序。为此，对实际项目中多线程程序出现的错误进行分析，提出一种新的多线程调试工具，并给出顺序点概念。设计用于调试的线程调度语言，在字节码层面对测试程序进行插桩，使线程按照期望的顺序调度。在此基础上实现一个 Eclipse 上的调试插件。实验结果表明，与现有工具 IMunit 相比，该调试工具减少了程序员的工作量，并拓展了两类程序的调试场景，具有较好的实用性。

关键词：多线程；调试；字节码；插桩；领域特定语言；插件

中文引用格式：周一耒，于海波，钟　浩. 基于字节码插桩的多线程调试工具[J]. 计算机工程，2016，42(11)：83-88.

英文引用格式：Zhou Yilei，Yu Haibo，Zhong Hao. Multithreaded Debugging Tool Based on Bytecode Instrumentation[J]. Computer Engineering，2016，42(11)：83-88.

Multithreaded Debugging Tool Based on Bytecode Instrumentation

ZHOU Yilei[a]，YU Haibo[a]，ZHONG Hao[b]

（a. School of Software；b. School of Electronic Information and Electrical Engineering，
Shanghai Jiaotong University，Shanghai 200240，China）

【Abstract】During the evolution of software，the debugging of programs is an important and necessary process. For multithreaded programs，the interleaving and scheduling are non-deterministic. Different scheduling can give different results，so it is difficult for developers to debug a multithreaded program. Therefore，this paper analyzes the concurrency bugs in real projects，presents a new multithreaded debugging tool，and proposes the concept of sequence point. It designs the scheduling language for debugging，instruments the test programs in bytecode level，and makes threads schedule in a desired sequence. It implements a debugging plugin in Eclipse. Experimental results show that compared with existing tool IMunit，the tool reduces developers' workload and enhances two debugging scenarios. It has better availability.

【Key words】multithreading；debugging；bytecode；instrumentation；domain-specific language；plugin

DOI：10.3969/j.issn.1000-3428.2016.11.014

0　概述

代码的调试是软件演化过程中不可或缺的重要环节[1]。该环节主要包含 3 个步骤：1）程序员生成一些测试用例让程序执行；2）如果程序报错，则修改代码以尝试修复；3）程序员会再次运行测试用例，验证前面发现的错误是否被修复。若仍然报错，则重复进行这 3 个步骤。相对于单线程程序而言，多线程程序的调试更为困难，因为线程的调度和线程间的交错是非确定的，不同的调度可能得到不同的结果[2]，导致多线程程序错误的发生有一定的偶然性，也很难控制多线程的调度。

现有的多线程相关的研究或工具大部分集中在多线程程序的错误检测上，其中典型的错误包括死锁[3-4]和数据竞争[5-6]。然而，无论是工具检测出的错误还是人为发现的错误，都需要程序员的参与去进行错误修复，并验证其是否被修复[7]。当知道程序有错误后，程序员一般会根据自己的经验，猜测或推测可能导致错误发生的某种线程调度顺序，并通过不同的方式，控制程序执行时的线程调度。现在使用最广泛的线程控制方法是改写源代码，插入如 Thread.sleep() 或 Object.wait()/notify 这类控制代码，以期望得到预测的线程调度顺序，而这类方式有诸多弊端。本文通过对实际项目中发生的多线程错

基金项目：国家自然科学基金(61572313)。

作者简介：周一耒(1991—)，男，硕士研究生，主研方向为程序分析；于海波，助理教授、博士；钟　浩，副教授、博士。

收稿日期：2015-11-13　　**修回日期：**2015-12-15　**E-mail：**yl.zhou@sjtu.edu.cn

误进行分析,发现现有的技术已不能满足程序员调试的需求,因此设计了一种新的多线程程序调试方法。首先定义了顺序点(sequence point)的概念,并设计了针对多线程调试场景的领域特定语言(domain-specific language)。然后在 Eclipse IDE 上实现了一个调试插件 SP Debugger,以帮助程序员进行多线程程序的调试。

1 相关研究

在调试多线程程序的过程中,最重要的是控制线程的调度[8],本节将介绍 3 种主要的线程控制方法。

1.1 基于睡眠的控制方法

程序员为调试多线程程序,会在源代码中插入时间相关的控制代码,这是目前使用最广泛的方法之一,本文称之为基于睡眠的控制方法。例如在 Java 中,程序员会插入 Thread.sleep(*n*),其中,*n* 是一个与现实时间相关的参数,表示该线程的执行会暂停约 *n* ms。假设有 2 个线程,在理想状态下,程序员希望线程 1 运行至某一处时,睡眠 *n* ms,让线程 2 先执行一段关键代码后,刚好切换回线程 1 继续执行。然而,在实际操作过程中,由于时间的不确定性,调度器未必会按照程序员设想的执行。假设一段多线程程序未按照设想的调度顺序执行,并且未能通过测试用例,这就有可能引发错误的误报。同样,假设该程序有多线程的并发错误,然而由于未按照设想的顺序执行线程调度,并通过了测试用例,那就会引发错误的漏报。此外,线程的睡眠时间一般是由程序员根据自身的经验,加上多次的尝试而估计的。可这一时间与程序的运行环境如硬件、软件等密切相关,一旦发生小小的变更,都有可能使得程序员得不到期望的调度顺序。即使是在运行环境完全一样的电脑上,时间的估计也是不精确的,因为程序员往往会故意高估,这也增加了资源的开销。因此,基于睡眠的控制多线程方式是不可靠的,而且是低效的。

1.2 基于通信锁的控制方法

在 Java 中,除了 Thread.sleep(),程序员也会往源代码里插入 Object.wait()/notify()/notifyAll() 这类代码,本文称为基于通信锁的控制方法,如框架 ConAn[9]。然而,这类方法极易往源程序中引入新的并发错误[10]。假设有 2 个线程,线程 2 先调用 Object.notify()/notifyAll(),线程 1 再调用 Object.wait(),则线程 1 有可能一直处于线程等待,永远不会有机会被唤醒。另一可能的场景是,假设有 3 个或多个线程,线程 1 和线程 2 都已经调用 Object.wait() 并进入了等待状态,某时刻线程 3 执行了 Object.notify(),则线程 1 和线程 2 中只有一个能被唤醒,且被唤醒的线程是由 JVM 决定的,其他线程将一直处于线程等待。因此,程序员在使用基于通信锁的方式进行调试时需尤为小心,这种调试方式也不是非常高效和实用。

1.3 基于事件的控制方法

近年来,有研究者提出了基于事件的控制方法,以 IMUnit[11] 为例,它为程序员提供了一种测试框架,通过在源代码中定义“事件”(event)以及“注释”(annotation)的方式控制多线程程序的调度,如代码 1中的程序片段,程序员定义了 4 个事件,并通过“注释”表示程序员期望程序执行时能满足如下调度条件:事件 finishedAdd1 发生在事件 startingTake1 之前,事件 startingTake2 发生在事件 startingAdd2 之前。

代码 1 IMUnit 对多线程程序的调度控制片段

```
1. @Test
2. @Schedule("finishedAdd1->startingTake1,
3.   [startingTake2]->startingAdd2")
4. public void testTakeWithAdd(){
5.   ArrayBlockingQueue<Integer> q;
6.   q = new ArrayBlockingQueue<Integer>(1);
7.   new Thread(
8.     new CheckedRunnable(){
9.       public void realRun(){
10.        q.add(1);
11.        @Event("finishedAdd1")
12.        @Event("startingAdd2")
13.        q.add(2);
14.    }
15.  },"addThread").start();
16.  @Event("startingTake1")
17.  Integer taken = q.take();
18.  assertTrue(taken == 1 && q.isEmpty());
19.  @Event("startingTake2")
20.  taken = q.take();
21.  assertTrue(taken == 2 && q.isEmpty());
22.  addThread.join();
23.}
```

这样的调试框架虽然在一定程度上帮助了程序员进行多线程的控制,但仍有一定的缺陷:首先,程序员要理解“事件”(event)和“注释”(annotation)等由框架开发者定义的语法规则,有些调试框架设计的语法比较晦涩难懂,程序员需要花费大量的学习时间和精力,编写时容易出错;其次,与前文提到的 2 种方式一样,程序员必须修改程序源代码,这在一定程度上限制了调试场景。

2 调试多线程程序的需求分析

本文研究对实际项目中多线程程序出现的错误

进行了分析,该节将举例说明程序员对多线程程序进行调试时的需求。

代码 2 展示了在 JDK1.6.0 中 Logger.java 的一段代码,这段代码有可能导致程序抛出空指针的异常。

代码 2　Logger.java 中的代码片段

```
1. public void log(LogRecord record){
2.    if(record.getLevel().intValue() < levelValue
3.       || levelValue == offValue){
4.          return;
5.    }
6.    synchronized(this){
7.       if(filter != null
8.          && ! filter.isLoggable(record)){
9.             return;
10.   }
11.   }
12.}
13.
14. public void setFilter(Filter newFilter)
15.       throws SecurityException {
16.    checkAccess();
17.    filter = newFilter;
18.}
```

为简化该异常发生的情景,假设有 2 个线程对 Logger 进行访问。在某一时刻,线程 1 执行了条件语句的前半段,即第 7 行,判断出变量 filter 不为空。此时发生了线程切换,调度器选择了线程 2 作为当前执行线程。线程 2 将执行函数 setFilter,并且参数 newFilter 恰好为 null。线程 2 执行完第 17 行,将 filter 重新赋值为 null 之后,调度器又切换回了线程 1。线程 1 会继续执行第 8 行,而此时 filter 的值为 null,故而程序会抛出空指针的异常。整个线程切换过程如图 1 所示。究其原因,是由于函数 setFilter 中未加关键词 synchronized,导致线程 2 依然能访问并修改 filter 的值。

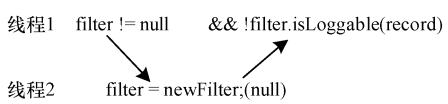

图 1　线程切换示意图

当程序员看到错误报告,显示程序有可能抛出 filter 为空的异常后,他们会期望通过尝试不同的线程调度,以重现这一错误发生的情景,因此,程序员需要一个调试工具,按照指定的顺序,控制多线程的调度。

对于代码 2 所示的程序片段,若程序员期望得到图 1 所示的调度顺序,按照现有的技术或框架,需要对语句进行拆分,并添加控制代码,对源代码的改动很大。因此,程序员希望工具能减少调试的工作量。

此外,Logger 是 Java 自带的日志类,程序员即使能找到 Logger 的源代码,也无法轻易地对其进行修改。因此,针对包含 Java 自带类或第三方类库的这些无法修改代码的程序,若需要控制多线程的调度,程序员需要调试工具能处理这类情况,避免对源代码进行修改。

因此,在分析了程序员对多线程程序调试工具的需求的基础上,本文提出了基于顺序点的调试技术,有如下特性:

1)可控性:对多线程程序进行调试,最重要的是能控制多线程的调度。本文提出的方法不会将多线程程序变成一个单线程程序,而是对关键代码的线程交错进行监控,帮助程序员得到期望的线程调度。

2)可靠性:上文提到基于睡眠的控制方式依赖现实时间,本文提出的方法将按照程序员的意图进行多线程的调度控制,不再依赖现实时间,从而避免错误的误报和漏报。

3)细粒度:从真实的错误报告中可以发现,在由逻辑运算符连接的长语句之间,程序可能发生线程间的切换,有可能引发多线程的错误。本文提出方法能控制语句中间的线程切换,有助于此类错误的调试。

4)广泛性:有时多线程程序的错误涉及第三方类库的使用,本文提出的技术不需要对源代码进行修改,故而适用更广泛的调试场景。

5)易操作性:本文所实现的调试工具提供了操作界面,不再像已有的测试框架需要用户手动添加代码,能简化操作,提升用户体验。

3　基于顺序点的调试技术

本文提出了一种基于顺序点的调试技术,用户需要将线程与对应顺序点进行绑定,并指定调度约束。为得到用户期望的线程调度,该调试技术会对程序进行插桩。

3.1　顺序点

顺序点这一想法是非常直观的,因为程序员想要指定线程的调度顺序时,必须清晰地表述哪一线程在哪一位置进行线程切换[12]。因此,在线程需要切换的位置,本文用连续的自然数表示顺序点,即顺序点 1 所在线程,必须等顺序点 0 执行之后才能继续执行。

本文对顺序点的定义如下:

定义 1　顺序点 $sp = < t,s,n >$,其中:

1)t:线程实例 ID,表示该顺序点属于线程 t,每个线程 ID 唯一。

2)s:源代码中的位置,包括顺序点的行(1)、

列(c)信息,如 l3c0,表示在源代码的第 3 行第 0 列处期望发生线程交错。

3)n:从 0 开始的连续的自然数,且 n 唯一,表示该顺序点的值,顺序点 n 必须先于顺序点 $n+1$ 执行。

以代码 2 中的程序为例,假设程序员想要重现图 1 所示的线程交错,那么需要绑定 4 个顺序点:{<线程 1,l8c0,0>,<线程 2,l17c0,1>,<线程 2,l18c0,2>,<线程 1,l8c0,3>},则程序执行到顺序点的序列是 0→1→2→3。当该段代码在执行时,如果线程 2 先到达顺序点 1,即线程 2 执行完 16 行,则线程 2 会被暂停,直到线程 1 到达了顺序点 0 后再唤醒线程 2。同样,线程 1 会被阻塞在第 8 行,直到线程 2 执行完第 17 行代码中对 newFilter 的赋值。

3.2 插桩

程序插桩是在程序中植入代码片段,并通过运行程序来获得该程序动态信息的技术。通过插桩可以收集程序在执行过程中某一时刻的系统状态(快照)信息,从而控制程序的执行[13-14]。

基于睡眠、基于信号锁的调试方式以及现有的调试框架,都是采用在源代码层面进行插桩的方式。如第 1 节所讨论,不仅增加了程序员的工作量,也可能会引入新的错误,更限制了错误的调试场景。

因此,本文研究采用字节码层面的插桩方式,不仅能对程序进行更细粒度的控制,也能适用于第三方库或 Java 类库这类无法修改源代码的调试场景,同时,在源代码层面,保证了代码的纯净性,保证插桩之后可得到的线程调度顺序,在未插桩的情况下一样可能发生。

3.3 调度约束

在代码 1 中,本文展示了一类最简单的约束,即数值约束,仅根据顺序点数值的大小进行线程交错的依据,比如顺序点 1 必须在顺序点 2 之前执行。而在实际项目中,有一些更复杂的场景,如代码 3 所示。

代码 3 循环声明多线程代码

```
1. public void testMultiThread(){
2.   for(int i = 0;i < 10;i++){
3.     new Thread(new Runnable(){
4.       public void run(){
5.         …
6.       }
7.     }).start();
8.   }
9. }
```

这段简化的代码展示了多线程程序中一类常见的代码模式,即通过循环声明多个匿名线程实例[15]。假如程序员期望在第 5 个创建的线程中,第 5 行的代码先于第 3 个创建的线程中的第 5 行代码执行,则需要给出更具体的约束。因此,除了数值的约束外,本文所设计的调度机制包含约束模块,允许用户添加其他约束,如代码 3 中“第 5 个创建的线程”对应的“i = =5”。

3.4 调度语言

为了更好地表述本文所提出的调度机制,本文研究定义了针对多线程调度的领域特定语言(Domain-specific Language)[16]。

1)事件(Events)

satisfyConstraints <thread;contraints>

notSatisfyConstraints <thread;contraints>

reachedSp <sp>

2)命令(Commands)

checkConstraints <constraints>

autoIncreaseSp <sp>

suspendThread <thread>

resumeThread <thread>

3)状态::就绪(State::Ready)

action checkConstraints

satisfyConstraints => Running

notSatisfyConstraints => Suspended

4)状态::运行(State::Running)

action resumeThread

action autoIncreaseSp

reachedSp => Ready

5)状态::等待(State::Suspended)

action suspendThread

reachedSp => Ready

在该领域特定语言中,含有 3 类事件:事件 satisfyConstraints 表示线程当前满足了程序员事先定义的所有约束;相反,事件 notSatisfyConstraints 表示线程当前不能满足所有约束;事件 reachedSp 表示有线程到达了某一个顺序点。

该语言包含 4 种命令:checkConstraints 命令所有线程检查是否满足约束条件;autoIncreaseSp 会自动将当前顺序点的值增加 1;suspendThread 和 resumeThread 会分别挂起或唤醒某一线程。

该语言定义了线程的 3 种状态:首先是就绪状态,当某线程遇到插入的顺序点,则触发了事件 reachedSp,该线程以及所有等待状态的线程都会加入到就绪状态的集合里。随后执行 checkConstraints 命令,只要该集合里的线程满足约束条件,即触发事件 satisfyConstraints,那线程状态就会由就绪变为运行,执行 resumeThread,同时执行 autoIncreaseSp 命令,改变当前顺序点的值以匹配下一个顺序点。最后若发现线程不满足所有的约束条件,即触发事件

notSatisfyConstraints，则线程会由就绪状态切换为等待状态，执行 suspendThread 命令。若运行状态的线程再次遇到顺序点，则该线程会进入就绪状态。而对于等待状态的线程而言，只要事件 reachedSp 被触发，都会进入就绪状态。图 2 展示了该调度语言对应的状态。

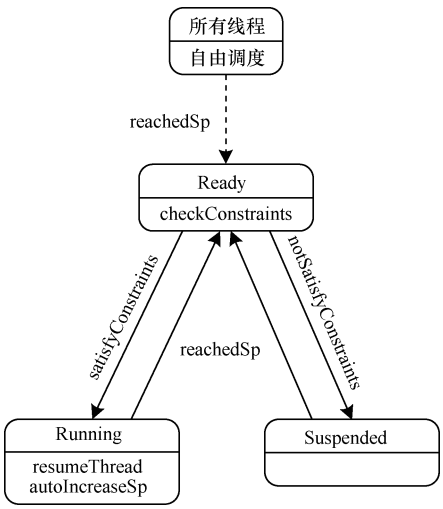

图 2　调度语言状态

本文研究所设计的调度语言，不仅能直观地展现本文设计的多线程调度机制，也有助于程序员理解和设计多线程程序调试时期望的调度顺序。

4　调试工具的实现

4.1　工具框架

为了验证本文所设计的调试方法的可行性，本文针对 Java 语言，基于上文提到的调度机制，在 Eclipse 上实现了一个调试插件 SP Debugger。图 3 即为该调试工具的框架。

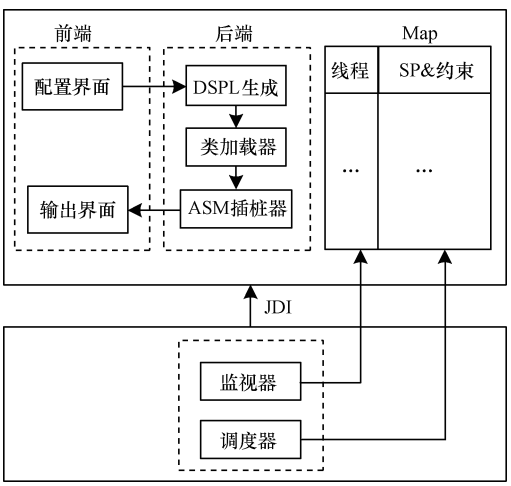

图 3　SP Debugger 框架

用户在配置界面对需要调试的线程进行顺序点的绑定，并设定线程交错的约束条件，在后端，这些配置将映射为前文定义的调度语言。SP Debugger 通过自定义的类加载器，筛选出用户需要进行监控的线程类，并通过开源插桩项目 ASM，在字节码层面对代码进行插桩，同时，在内存中，该插件会维护一个线程与顺序点和约束的映射。

为控制线程的调度，该工具的实现使用了 JDI（Java Debug Interface），它提供了一系列 Java API 以用于程序的调试。事实上，待调试的程序和调试工具将运行在 2 个 Java 虚拟机上，通过 JDI，调试器能得到待测程序运行时的状态，如类、数组、对象等信息，并能根据用户的规约，控制指定线程的挂起或唤醒。采用 JDI 作为线程控制的手段，一方面能避免基于时间的调度技术带来的不可靠性，另一方面易于在将来将这样的调试技术扩展为远程调试。

4.2　工具评估

为验证 SP Debugger 的有效性，本文研究选取了以下 3 种类型的多线程程序对工具进行评估：

1）通过基于睡眠的控制方式或基于信号锁的控制方式，能得到指定调度顺序的程序。

2）程序源代码中含有由“||”或“&&”这类逻辑运算符连接的长语句，且程序员期望在逻辑运算符的位置控制线程的交错，如代码 2 中第 7 行、第 8 行代码所示。

3）程序源代码中用循环声明多个线程实例，程序员期望指定其中若干线程的调度顺序，例如代码 3 中展示的程序片段。

本文研究设计了第 1 类程序共 8 个，第 2 类程序共 4 个，第 3 类程序共 4 个，共计 16 个测试用例用于 SP Debugger 与现有框架 IMUnit 的对比测试。对于每一个待测程序，指定一种潜在的线程调度顺序，然后借助工具，使程序在运行过程中按照预期的顺序进行线程调度。表 1 展示了不同工具对程序控制的成功率。

表 1　3 类程序调度控制成功率分析比较　%

类别	SP Debugger	IMUnit
第 1 类	100.00	75.0
第 2 类	100.00	—
第 3 类	75.00	—
总计	93.75	37.5

经实验可以看出，对于第 1 类程序，IMUnit 无法对 2 个涉及第三方类库的测试用例进行线程控制，而 SP Debugger 都能按照指定顺序进行线程调度。

对于第 2 类、第 3 类程序，在不修改原有代码的前提下，IMUnit 无法得到预期的线程调度，除非对原有语句进行拆分，或循环体内插入判断语句等，故认为 IMUnit 不适用这 2 类程序的调试。

对于第 2 类程序，SP Debugger 通过了所有 4 个

用例的测试。

而对于第 3 类程序中的一个测试用例,SP Debugger 未能成功控制线程调度,经分析后发现,该程序有 *A*. *a*(*B*. *b*())这样的函数调用链,即调用函数 *B*. *b*()所得结果是函数 *A*. *a*()的参数,程序员期望调用 *B*. *b*()之后立刻发生线程切换,而 SP Debugger 目前尚未支持该调试场景。

经过对比实验发现,本文实现的 SP Debugger 具有以下优势:

1)不依赖运行时间和运行环境,程序员绑定的顺序点和指定的调度约束是控制调度的仅有依据,故所得的调度顺序真实可靠,完全按照程序员的意图进行控制。

2)不需要修改源代码,减少了程序员调试的工作量。

3)能控制由"||"或"&&"组成的长语句间的线程交错,从而对多线程程序进行更细粒度的调度控制。

4)能控制涉及循环、第三方类库代码的线程交错,与现有工具相比,拓展了调试场景,有更好的实用性。

5 结束语

本文从实际的多线程程序错误出发,分析了程序员对多线程程序调试工具的需求。为帮助程序员调试多线程程序,本文针对 Java 提出一种基于顺序点的多线程调试技术,将需要监控的线程与顺序点进行绑定,与现有技术不同,本文选择在字节码层面对程序进行插桩,并通过 JDI 控制线程的调度,以得到程序员期望的线程调度顺序。最后实现了一个 Eclipse 上的多线程调试插件,实验结果表明,该插件可适用于更广泛和更复杂的调试场景。下一步将完善该插件,控制函数调用链之间的线程切换,并提供类似单线程调试时打断点(breakpoint)的这种简单操作,以及远程调试等更多功能。

参考文献

[1] Pressman R. 软件工程:实践者的研究方法[M]. 郑人杰,马素霞,译. 7 版. 北京:机械工业出版社,2001.

[2] 周广川. 多线程应用程序调试技术[J]. 现代计算机(专业版),2011(3):28-30,47.

[3] Yan Cai, Chan W K. Magic Fuzzer: Scalable Deadlock Detection for Large-scale Applications [C]// Proceedings of the 34th International Conference on Software Engineering. Zurich, Switzerland: IEEE Press, 2012:606-616.

[4] Eslamimehr M, Palsberg J. Sherlock: Scalable Deadlock Detection for Concurrent Programs[C]//Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. Hong Kong, China:[s. n.],2014:353-365.

[5] Sen K. Race Directed Random Testing of Concurrent Programs[C]//Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. Tucson, USA: ACM Press, 2008:11-21.

[6] Kasikci B, Zamfir C, Candea G. RaceMob: Crowdsourced Data Race Detection[C]//Proceedings of the 24th ACM Symposium on Operating Systems Principles. Farminton, USA: ACM Press, 2013:406-422.

[7] Layman L, Diep M, Nagappan M, et al. Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers[C]//Proceedings of 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. Baltimore, USA: IEEE Computer Society, 2013: 383-392.

[8] Carver R H, Tai K C. Modern Multithreading: Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs [M]. Hoboken, USA: John Wiley & Sons Inc.,2005.

[9] Long B, Hoffman D, Strooper P A. A Concurrency Test Tool for Java Monitors[C]//Proceedings of the 16th IEEE International Conference on Automated Software Engineering. San Diego, USA: IEEE Computer Society, 2001:421-425.

[10] Welch P H, Martin J M R. A CSP Model for Java Multithreading[C]//Proceedings of International Symposium on Software Engineering for Parallel and Distributed Systems. Limerick, Ireland: IEEE Computer Society, 2000:114-122.

[11] Jagannath V, Gligoric M, Jin Dongyun, et al. Improved Multithreaded Unit Testing [C]//Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. Szeged, Hungary: ACM Press, 2011:223-233.

[12] 姜 波. 线程调度方法与测试工具的研究与实现[D]. 长沙:国防科学技术大学,2010.

[13] 郑晓梅. 一个基于 Eclipse 的通用 Java 程序插桩工具[J]. 计算机科学,2011,38(7):139-143,169.

[14] Huang Jeff, Liu Peng, Zhang Charles. LEAP: Lightweight Deterministic Multi-processor Replay of Concurrent Java Programs[C]//Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Santa Fe, USA: ACM Press, 2010:207-216.

[15] Travis G, Wooldridge M, Horstmann C S, et al. Building a Java Chat Server[EB/OL]. (2001-01-30). http://www.ibm.com/developerworks/edu/j-dw-javachat-i.html.

[16] Fowler M. 领域特定语言[M]. Thought Works 中国,译. 1 版. 北京:机械工业出版社,2013.

编辑 索书志