第41卷
2005年10月

南京大学学报(自然科学)
JOURNAL OF NANJING UNIVERSITY
(NATURAL SCIENCES)

Vol.41
Oct., 2005

Java并发程序动态分析技术研究*

王 越,金茂忠,刘 超
(北京航空航天大学软件工程研究所,北京,100083)

摘 要: 并发程序应用越来越普遍,但其固有的运行不确定性,加大了设计、编码、测试和纠错的难度.介绍一种基于插装的Java并发程序动态执行轨迹跟踪技术,通过该技术可视化地向用户展示并发程序执行的内部逻辑.详细讨论Java并发同步原理、动态测试的模型、插装库设计与插装策略等,该项技术在实际工作中已得到应用.

关键词: 软件测试,动态测试,插装技术,并发程序
中图分类号: TP 311.5

Profiling Java Concurrent Programs by Dynamic Analysis

Wang Yue, Jin Mao-Zhong, Liu Chao
(Software Engineering Institute, Beihang University, Beijing, 100083, china)

Abstract: Concurrent applications are widely accepted by the industry. But their inherient runtime uncertainty always leads to troubles related to designing, coding, testing, and debugging. This paper propose an instrumentation - based profiling technology to reveal the runtime logic to end user visually. The concurrent theory of Java applications, dynamic testing models, instrumentation base and stategy will be introduced in detail. The technology has been improved to be feasible in real work around.

Key words: software testing, dynamic testing, instrumentation, concurrent programming

近年来由于硬件成本大幅度下降,为了高效利用大量信息资源,多台机器,多个CPU(多个主机协同计算)并发完成一个计算任务已经是极普通的事.采用并发语言进行并发程序设计已成为应用软件开发的主流技术.为支持并发程序设计的多线程机制,Java语言为多线程编程提供了语言级的支持[1,2],通过Java语言可以很方便地让一个程序同时处理多个任务,这在一定程度上已经满足了我们应用的要求.由于并发程序固有的运行不确定性(non-determine)[2,3],往往很难对其进行设计、编码、测试和纠错.并发程序里常常有多个线程同时运行,它们共享一些变量,所以,即使有相同的输入,各线程间的执行序列也是不可预见的,这也就导致其可能有不同的输出,进一步增加了问题的重现难度.出现这种情况时,编程人员往往很难检测到错误,更不用说去纠错了.

对并发程序的分析通常有两种方法:一种是静态分析(Static Analysis),即不运行程序,分析各个线程对共享变量的使用序列,借此发现

* 基金项目:国家863高技术项目"基于Web Services的系统软件核心技术及运行平台研究"(2004AA112030)

死锁等情况.另一种方法是动态分析(Dynamic Analysis),通过运行程序,来获得程序在某次执行过程中的真实情况,如[4].静态分析后的并发程序在特定的环境(既包括软件环境,也包括硬件环境)下,仍然可能会出现各种不可预料的问题,此时,由于程序开发人员并不能知道程序执行的真实过程,也就拿它束手无策.本文通过分析 Java 并发程序开发中的关键技术,归纳出运行过程中会引起程序状态改变和各线程进行交互的关键点;通过插装技术,提取出程序在真实环境下执行过程中,各线程发生的状态变化和各线程之间的交互情况,分析出程序动态执行轨迹,并以可视化的形式展示给用户[5],辅助程序开发人员分析程序中潜在的问题,进而完美地解决问题.

本文主要介绍了一种基于插装的 Java 并发程序动态执行轨迹跟踪技术和对跟踪数据的可视化表示.

1　Java 同步技术概述

1.1　Java 同步方法及其实现级别　同一进程的多个线程共享同一片存储空间,在带来方便的同时,也引发访问冲突的问题,Java 语言通过锁机制解决此类冲突.锁机制限制同一时间,两个线程不能够同时访问被锁住的对象,借此来实现对对象的互斥访问.在 Java 中,通过 synchronized 关键字对对象进行加锁.如果一个线程已经对一个对象进行了加锁,其它线程就不能访问该对象,直到该对象被解锁为止.在 Java 语言中,有两种方式来对一个对象进行加锁:synchronized 块和 synchronized 方法[6,7]

synchronized 块:　通过 synchronized 关键字来声明 synchronized 块:

```
synchronized (syncObject) {
  //允许访问控制的代码
}
```

synchronized 块是这样一个代码块,其中的代码必须获得对象 syncObject(可以是类实例或类)的锁方能执行,具体机制同前所述.由于可以将任意代码块说明为同步块,且可任意指定上锁的对象,故灵活性较高.

synchronized 方法:　通过在方法声明中加入 synchronized 关键字来声明 synchronized 方法:

```
public synchronized void syncMethod() {
  //允许访问控制的代码
}
```

synchronized 方法控制对类成员变量的访问:每个类实例对应一把锁,每个 synchronized 方法都必须获得调用该方法的类实例的锁方能执行,否则所属线程阻塞.方法一旦被执行,就独占该锁,直到从该方法返回时才将锁释放,此后被阻塞的线程方能获得该锁,重新进入可执行状态.这种机制确保了同一时刻对于每一个类实例,其所有声明为 synchronized 的成员函数中至多只有一个处于可执行状态(因为至多只有一个能够获得该类实例对应的锁),从而有效避免了类成员变量的访问冲突(只要所有可能访问类成员变量的方法均被声明为 synchronized).

在 Java 中,每一个类也对应一把锁,这样也可将类的静态成员函数声明为 synchronized,以控制其对类的静态成员变量的访问.

当然,一个方法也有可能锁住几个对象.如:

```
synchronized (syncObject1) {
  // 锁住第一个对象 syncObject1
  synchronized (syncObject2) {
    // 同时锁住这两个对象 syncObject1 和
    // syncObject2
  }
}
```

1.2　等待和通知方法

wait 是等待方法,它将释放对当前对象所加的锁,从而允许其它线程有机会对该对象进行操作.notify 是通知方法,负责通知等待队列中的线程其等待条件已经发生了变化,现在可以运行了.

wait 和 notify 常常配套使用,wait 使得线

程进入阻塞状态,在对应的 notify 被调用或者超出指定时间时该线程再重新进入可执行状态.

比如,生产者和消费者问题中,有两个方法 put 和 get,它们共享缓冲区(buffer). put 方法负责向缓冲区中添加数据,get 方法则负责从缓冲区中取数据.它的 put 方法常常如下所示.

进入该方法的线程首先检测 buffer 中是否有数据,如果有,就通过 wait 方法等待数据被消费之后再添加数据;如果没有,向其中添加数据,并通过 notify 方法唤醒可能被阻塞的消费者线程.

```
public synchronized void put (Item item) {
  while(buffer.size() > 0) {
    wait();
  }
  buffer.add(item);
  notify();
}
```

从上文可以看出,同步方法、同步块以及引起线程之间通信信息的等待和通知等语法现象都是我们进行线程测试需要关注的侧重点,也是进行 Java 并发程序动态执行轨迹跟踪的关键.

2 线程插装设计

插装技术(instrumentation),是 20 世纪 70 年代由 J. C. Huang 教授首先提出的,一种对程序进行动态测试的新技术[8,9].通过有目的地向被测程序中插入一些预先设计的语句,来捕获程序在动态执行过程中的关键信息,进而帮助编程人员了解程序的实际执行情况,计算出程序的语句、分支覆盖率等信息.本节就结合第一节介绍的 Java 同步技术,介绍如何剖析 Java 并发程序动态执行轨迹.

采用插装技术,对 Java 并发程序进行动态分析的原理如图 1 所示,主要通过 4 个步骤实现:第一步,对用户要分析的源文件进行插装,使程序运行时能捕获有关同步块、等待和通知的动作(2.2 节将进行详细描述).第二步是准备插装库,实现对线程的封装.这两个步骤没有依赖关系,可以并行开发.第三步在 Java 虚拟机中运行程序,在运行过程中,植入的插装语句将能根据插装库的提示正确地输出跟踪到的数据.

最后,结果显示部分根据在程序运行过程中跟踪到的数据,采用 UML 顺序图的方式,向用户友好地显示出各线程之间的交互信息,以及它们进入同步块的情况,以方便用户的进一步分析.

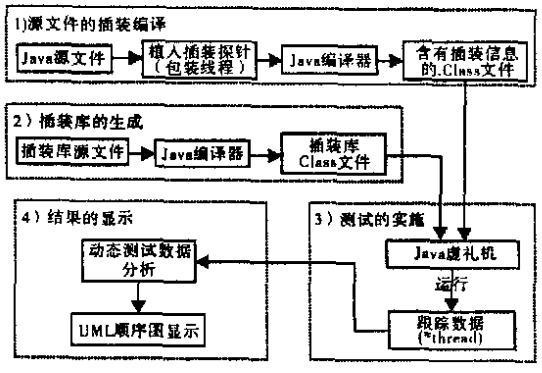

图 1 基于插装的动态测试实现原理图

2.1 用户线程的封装 在动态分析过程中,既需要区分各个不同的线程,还需要为每个线程保留其状态转移情况等.如何得到和处理这些信息是一个关键问题.

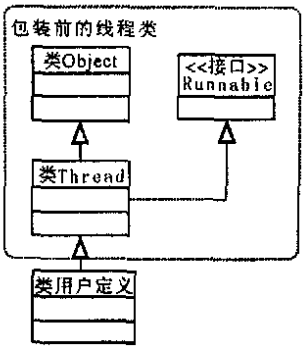

图 2 封装前的线程类

Java 中,用户的每一个线程都是通过创建 java.lang.Thread 类或其子类的实例生成,如图 2 所示.本文对 JDK 提供的 Thread 类进行封装,即在插装库中定义一个直接继承自 Thread

的类 JThreadInst,在对用户程序进行插装的过程中将用户对 Thread 类的继承(或实例化),改变为对 JThreadInst 类的继承(或实例化).这样,就实现了对 Thread 类的封装,如图 3 所示:

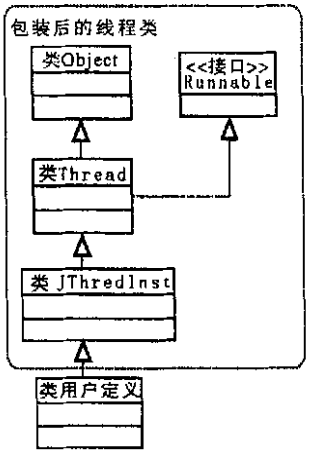

图 3　被封装后的线程类

通过线程封装,在 JThreadInst 类中重载 Thread 类的构造函数,为每一个新创建的用户线程动态分配唯一的标识号,以便在动态跟踪信息中识别线程.同时,重载 Thread 类中用于管理线程的方法,如 start、sleep、yield 等方法,实现对用户线程状态转移的跟踪.如对 start 方法的重载方式是:

```
public void start(){
    //插入探针语句
    threadmethodstub-(2,
JThreadInst.currentThread(), instancedId);
    super.start();
}
```

在插入的语句中,JThreadInst.currentThread 方法记录执行 start 方法的线程号,而 instanceId 记录将要运行的那个线程的动态标识号,这样,就捕获到由 JThreadInst.currentThread 线程启动的 instanceId 线程运行的程序动态运行信息.

通过把插装线程用的探针放在中间层中,减少对用户程序的修改,以实现对用户的透明性.

这样,就在封装层中简洁的实现动态分配标识号及部分的线程插装.

2.2　等待、通知和同步块的插装技术　控制同步块访问的 wait,notify 和 notifyAll 三个方法都在 java.lang.Object 类中,如果采用上述的封装技术对这个类进行重载,势必影响到太多的类,并且有时是办不到的.

这种情况下,可以在用户程序中 wait 语句的前后分别加入一条探针语句,记录等待动作的开始和结束;在 notify 和 notifyAll 方法前加入一个探针语句,记录通知动作的发生.

同步块(包括同步方法)是并发程序的核心,需要记录每个线程进入和退出同步块的情况.为此,就要在同步块的入口和出口处分别插入一条语句,用来识别进入或退出的线程 ID.在同步块的入口处插入语句比较简单,但程序的出口往往比较难以发现,想要在该处放入插装语句也就没有那么简单了.

首先,程序可能会通过 return 语句从同步块的中间退出;其次,即使块中间没有 return 语句,还有可能有 if{…}else{…}或 switch(.){…}等各种控制流语句,它们的出口都是很难判断的.稍不留意插入插装语句以后的程序可能就不能正常编译.在这种情况下,可以考虑使用 Java 对异常处理的方法,Java 的异常处理里有 try{…}finally{…}形式,它能确保不管程序从 try 块的何处退出,都肯定会执行 finally 块中的语句.为此,在遇到同步块时,可以将整个同步块放入 try 块中,将需要在同步块出口处插装的语句放到 finally 块中,这样就很好地解决了在同步块的入口、出口插装问题.

综上所述,线程封装可以跟踪用户创建的每一个线程及其状态转移情况;对同步块和等待、通知方法的插装,可以跟踪每一个线程进入并发程序同步块以后的信息交互情况.至此,便可以得到 Java 并发程序动态执行情况的数据,为了给用户比较直观的印象,并便于了解各线程之间的交互信息,本文采用 UML 顺序图向用户展示所得到的结果.对会引起线程状态改变和引起线程交互的 Java 并发程序关键语句和对它们进行的插装实现方式总结如表 1 所示:

表 1 插装方式

	方法类型	所在类	类 型	处理方法
1	new	java. lang. Thread	Thread()	封装线程，重载 java. lang. Thread 中的构造函数和对应的方法，在重载的函数中，插入插装语句.
2	start 方法		start()	
3	destroy 方法		destroy()	
4	Dsleep 方法开始处			
5	sleep 方法结束处		sleep()	
6	wait 方法开始处	java. lang. Object	wait()	用户程序中进行插装.
7	wait 方法结束处			
8	notify 方法		notify()	
9	notifyAll 方法		notifyAll()	
10	run 方法	户定义的类中	run()	在 run 方法出口处插装
11	Synchronized 块/方法开始处			在同步块的入口和出口处进行插装
12	Synchronized 块/方法结束处			

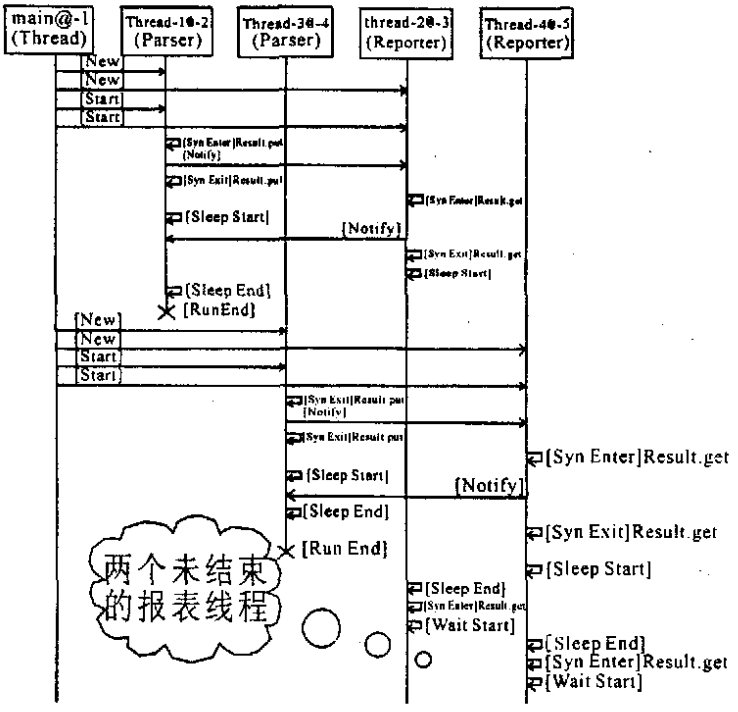

图 4 线程交互信息图

3 应用实例

QESat/Java(Java 软件分析与测试工具)是北京航空航天大学软件工程研究所承担的 863 课题——"基于 Web Services 的系统软件核心技术及运行平台研究"下的子课题，Java 并发程序动态执行轨迹跟踪技术是其中一项关键技术，目前已经实现，并已经在软件分析与软件测试中得到广泛应用，下面就介绍其在 QESat/Java 工具本身报表子系统开发中的应用情况.

我们在开发 QESat/Java 工具的报表子系统过程中，为提高后端显示效率引入了 Java 并

发开发技术.静态分析器线程分析被测试项目,将结果放到临界区;报表显示线程监测该临界区,一旦有数据就进行一次报表内容的更新,然后清空该临界区.在开发过程中,发现系统中的报表显示线程在进行报表更新以后并没有结束,于是,在系统进行多次静态分析以后,累积下来的线程就越来越多.通过采用 QESat/Java 对这一部分并发程序动态执行轨迹进行跟踪,得到某一次进行两次静态分析以后,各线程的创建及执行情况如图 4 所示:

从分析出的 UML 顺序图可以很明显地看出:两个静态分析器(parser)线程分别进入一次 Result.put 方法放置静态分析结果,然后整个线程运行结束;而两个报表子系统(reporter)线程都是在成功更新一次报表内容以后,还进入一次同步块 Report.get 方法,而这一次并没有取到可供更新的静态信息,从而一直在等待,使得程序一直都结束不了.显然,报表线程的控制逻辑有问题,它应该在进行一次数据更新以后就结束运行;通过改变报表线程的处理逻辑,解决该问题.

实践证明:该 Java 并发程序动态执行轨迹跟踪技术能精确检测到程序真实执行情况,通过采用 UML 顺序图输出,友好、直观地向用户展示各线程的执行轨迹.从而方便软件测试人员和程序开发人员发现问题、定位问题进而高效地解决问题.

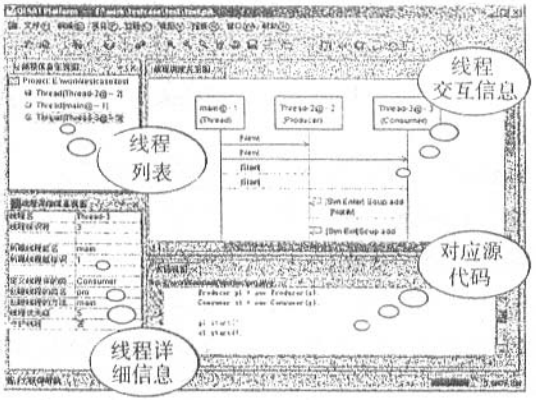

图 5　工具完整界面

图 5 为工具的完整界面,其中既以树状图的方式显示了线程组、线程之间的依赖关系,还显示了各个选中线程的相关信息和对应源程序.

4　总　结

Java 在并发程序中的应用越来越广泛.其执行过程的不确定性也使得对其开发、测试和维护极为困难.本文通过对线程进行封装,跟踪各线程进入同步块的情况,并以 UML 顺序图的方式显示给用户,大大方便了用户的测试,使并发程序的内部执行轨迹都确定、友好地呈现给用户.通过采用 Java 并发程序动态分析技术,不仅减少了并发程序开发引入错误的可能,还为开发者发现错误、定位错误提供了方便.在研究过程中发现,这样的工作还可以再深入下去:监测到程序的执行情况以后,可以基于一定的 bug 模式,自动检测程序是否出现了问题,如果检测到问题,就结束该线程并主动向用户报告,以进一步提高工具的自动化程度.

References

[1] Bruce E. Thinking in Java. 3rd Edition. New York: Prentice Hall, 2004.

[2] James G, Bill J, Guy S, *et al*. The Java Language Specification. 2rd Edition. California: Addison Wesley, 2000.

[3] Tim L, Frank Y. The Java Virtual Machine Specification. 2rd Edition. California: Addison Wesley, 1999.

[4] Bechini A, Tai K C. Design of a toolset for dynamic analysis of concurrent Java programs. Program Comprehension. IWPC ' 98. Proceedings. 6th International Workshop, 1998, 190~197.

[5] Mehner K, Wagner A. Visualizing the synchron-ization of Java-threads with UML. Visual Languages, 2000. Proceedings of 2000 IEEE International Symposium, 2000, 199~206.

[6] Scott O, Henry W. Java Threads. Beijing: China Electrical Power Press. 2003. (Scott O, Henry W, Java 线程. 黄若波,程峰.北京:中国电力出版社, 2003).

[7] Paul H. Java Thread Programming. Beijing: Posts and Telecom Press, 2003. (Paul Hyde. Java 线程编程. 周良忠.北京:人民邮电出版社, 2003).

[8] Huang J C. Program instrumentation and software testing. IEEE Computer, 1978, 11(4):25～32.

[9] Huang J C. Detection of data flow analysis through program instrumentation. IEEE Trans Software Eng, 1979, SE-5(3):226～236.

[10] Long B, Strooper P. A classification of concurrency failures in Java components. Parallel and Distributed Processing Symposium, Proceedings. International, 2003:22～26.