ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.9, September 2008, pp.2218−2227
DOI: 10.3724/SP.J.1001.2008.02218

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于面向方面调用图的 AspectJ 动态通知编织优化*

曹 璟[1,2+], 徐宝文[1,2], 周晓宇[1,2], 钱 巨[1,2], 杨 彬[1,2]

[1](东南大学 计算机科学与工程学院,江苏 南京 210096)

[2](江苏软件质量研究所,江苏 南京 210096)

Optimizing AspectJ Dynamic Advices Weaving Based on Aspect-Oriented Call Graph

CAO Jing[1,2+], XU Bao-Wen[1,2], ZHOU Xiao-Yu[1,2], QIAN Ju[1,2], YANG Bin[1,2]

[1](School of Computer Science and Engineering, Southeast University, Nanjing 210096, China)

[2](Jiangsu Institute of Software Quality, Nanjing 210096, China)

+ Corresponding author: E-mail: jing_cao@seu.edu.cn

Cao J, Xu BW, Zhou XY, Qian J, Yang B. Optimizing AspectJ dynamic advices weaving based on aspect-oriented call graph. *Journal of Software*, 2008,19(9):2218−2227. http://www.jos.org.cn/1000-9825/19/2218.htm

Abstract: This paper firstly presents an aspect-oriented call graph (ACG for short), then introduces an AspectJ dynamic advices weaving optimizing method based on the ACG (aspect-oriented call graph) of AspectJ programs. Our method firstly solves a call stack through the ACG and deduces the types of the nodes in the stack, then match the call stack with pointcuts, and finally decides how to weave dynamic advices based on the result of matching. A case study shows that this method has great precision and can identify most of weaving points statically.
Key words: advice weaving; optimizing; call graph; AspectJ; aspect-oriented programming

摘 要: 在提出一种适合 ApsectJ 程序分析的面向方面调用图的基础上,给出了一种 AspectJ 动态通知编织优化方法.该方法利用程序调用图求解调用栈,并对栈中节点进行类型推导,再将调用栈与切点匹配,根据匹配结果决定通知织入方式.实例研究结果表明,该方法精确度高,能够静态确定程序中大部分动态通知的织入点.
关键词: 通知编织;编译优化;调用图;AspectJ;面向方面程序设计
中图法分类号: TP314 **文献标识码**: A

在软件开发过程中,系统存在一些无法使用传统技术封装的横切关注点[1,2],它们散布在程序的多个模块中,容易引起代码混乱.面向方面程序设计(aspect-oriented programming,简称 AOP)即是一种模块化横切关注点的技术.在 Java 基础上扩充起来的 AspectJ 作为一种典型的 AOP 语言,在 AOP 的研究与实现过程中发挥着非常重要的作用[1−3].

AspectJ 将横切关注点封装为单独的方面(aspect),编译程序通过分析其中通知(advice)关联的切点

* Supported by the National Natural Science Foundation of China under Grant No.60503033 (国家自然科学基金); the National Natural Science Foundation for Distinguished Young Scholar of China under Grant No.60425206 (国家杰出青年科学基金); the Natural Science Foundation of Jiangsu Province of China under Grant No.BK2006094 (江苏省自然科学基金); the High Technology Research Projects Foundation of Jiangsu Province of China under Grant No.BG2005032 (江苏省高技术研究项目)

Received 2006-10-08; Accepted 2007-07-17

(pointcut)寻找程序中对应的织入点,织入相应的通知调用.由于动态通知的织入点在编译时难以准确判定,因此,编译程序通过在所有可能的织入点上插入滞留程序留待运行时匹配来保证编织的准确性.程序中,此类织入点往往较多,这就增加了程序额外的运行开销.如何减少这部分开销是 AspectJ 程序编译优化的一个重要课题.

为此,人们在深入研究的基础上提出了一些解决方法,主要有利用部分求值技术的优化方法[4]、构造匹配自动机的优化方法[5]以及基于 *cflow* 分析的优化方法[6,7].其中,前两种方法只能适当提高动态匹配效率,均未尝试消除匹配过程;而后一种方法只考虑了 *cflow* 切点带来的动态性,没有涉及对象运行时类型检查带来的动态性.

本文提出了一种 AspectJ 程序动态通知编织的优化方法.该方法利用程序的调用图求解调用栈,并对调用栈中的节点进行类型推导,再将调用栈与切点匹配,根据匹配结果优化通知的织入.由于传统程序调用图不适合面向方面程序的分析,因此,本文首先给出了一种面向方面的调用图 ACG(aspect-oriented call graph),在此基础上进行动态通知的编织优化.实例研究结果表明,在 ACG 上求解调用栈并结合动态类型推导的方法能够显著地提高优化的精度,有效减少动态匹配过程,从而有助于提高程序的运行效率.

1 动态通知编织问题分析

AspectJ 通过连接点(join point)将基程序和方面程序联系起来.连接点是一组预定义的程序执行点(例如,方法调用、方法执行等),AspectJ 用这些执行点组成程序调用栈.在程序运行过程中,当调用栈出现特定模式时,相应通知中的代码就会被执行.通知与切点相关联,切点指定了该通知关注的调用栈模式.AspectJ 的切点可分为静态和动态两种.静态切点只考察调用栈的栈顶元素,同时忽略对象的运行时类型.例如,切点 *call*(*void m*())表示关注当前任意对象 *m* 方法的调用.

动态切点又可分为 3 类:第 1 类只考察对象的运行时类型(通过 *this*,*target* 或 *args* 切点指定),例如,附录程序中通知 *ad*1 的切点要求被调对象是 *B* 的实例(本文在论述过程中以附录的 AspectJ 程序为例);第 2 类只考察调用栈中的调用信息(通过 *cflow* 或 *cflowbelow* 切点指定),例如,通知 *ad*2 的切点要求栈中包含 *m*3 方法的调用;第 3 类则兼具前两类动态特性,既考察运行时类型,又考察调用栈,例如,通知 *ad*3 的切点.通常,方面程序中第 3 类动态切点居多.我们分别把包含静态切点和动态切点的通知称为静态通知和动态通知.

AspectJ 编译程序将方面静态织入到程序中[8].通过分析通知的切点,确定程序中的织入点,织入相应通知的调用.该织入方式使得方面的执行不需要修改 Java 虚拟机.AspectJ 编译程序能够确定静态通知的织入点,但难以确定动态通知的织入点,只能将其调用织入所有可能的织入点,并插入条件判断语句作为滞留程序,留待运行时作匹配.例如,编织通知 *ad*1 会在通知调用前插入 instanceof 判断语句(如图 1(a)所示),编织通知 *ad*2 会在通知调用前插入调用栈是否为空的判断语句(如图 1(b)所示,编译程序会在 *m*3 方法调用前、后分别插入入栈和出栈语句).

```
if (obj instanceof B)                if (!cflowstack.getThreadStack().isEmpty())
    call advice;                         call advice;
obj.m1();                            obj.m1();
```

(a) (b)

Fig.1 Residue programs

图 1 滞留程序

滞留程序无疑增加了程序运行时的开销.由于方面程序通常包含较多的动态通知,如何减少这部分开销成为提高 AspectJ 程序运行效率的关键.

1.1 cflow分析

解决上述问题的一种可行方法是利用包含了方法调用信息的扩展调用图作优化.对于第 2 类动态切点,可以通过该图求解程序执行到织入点时调用栈的所有可能模式,若这些模式被切点包含,则能够明确织入通知调用;若切点不包含任一种模式,则无须织入.程序中的织入点对应于调用图中一个(或多个)节点(方法执行对应图中的单个节点,方法调用可能对应图中的多个节点),因此,调用栈的所有可能模式可以表示为扩展调用图中从

起点到这些节点的所有路径的逆序列.例如在图 2 中,当程序执行到节点 10 时,调用栈为

(*mcall*,*m*1);(*mexec*,*m*3);(*mcall*,*m*3);(*mexec*,*main*).

当程序执行到节点 9 时,调用栈为

(*mcall*,*m*1);(*mexec*,*m*2);(*mcall*,*m*2);(*mexec*,*main*),

或

(*mcall*,*m*1);(*mexec*,*m*2);(*mcall*,*m*2);(*mexec*,*m*3);(*mcall*,*m*3);(*mexec*,*main*),

其中,*mcall* 表示方法调用, *mexec* 表示方法执行.因此,通知 *ad*2 可织入节点 10,但需条件织入节点 9.

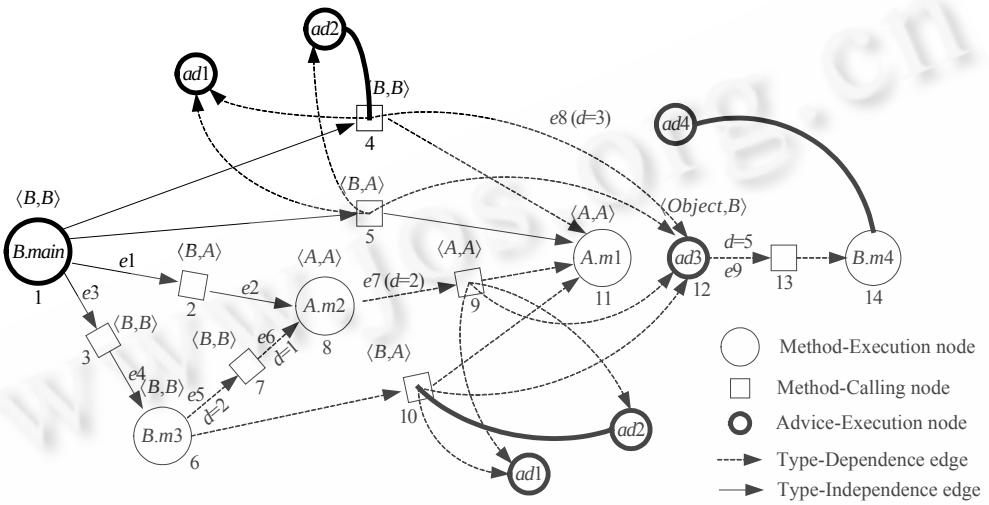

Fig.2 Aspect-Oriented call graph of the case

图 2 实例的面向方面调用图

1.2 类型分析

对于第 1 类动态切点,可以通过在调用图节点上附加类型信息,并利用节点间的调用关系进行类型推导,求出执行到该节点时对象的所有可能类型.若这些类型被切点包含,则明确织入;若切点不包含任一种类型,则无须织入.例如在图 2 中,节点 5 处被调对象的类型是 *A*,节点 9 处被调对象的类型是 *A* 或 *B*,节点 10 处被调对象的类型是 *B*,因此,通知 *ad*1 织入节点 10,不织入节点 5,条件织入节点 9.

对于第 3 类动态切点,则需综合 *cflow* 分析和类型分析的方法进行优化.由此,本文提出了一种通用的动态切点编织的优化方法:首先构造程序的面向方面调用图,然后在其上求出可能织入点的路径表达式,并推导式中所含节点的类型,最后将该表达式与切点相匹配,若切点包含表达式中的所有模式,则将通知调用无条件织入该节点;若切点不包含其中任意一种模式,则通知调用不织入;至于其他情况,通知调用仍需条件织入.

为了突出重点,我们对方面使用的切点加以约束,采用 AspectJ 切点语言的子集,但这不妨碍将该方法推广到一般情况下:

切点::=*call*(方法名)|*execution*(方法名)|*adviceexecution*()

|*target*(类名)|*this*(类名)

|*cflow*(切点)|*cflowbelow*(切点)

|(切点&&切点)|(切点||切点)

2 面向方面调用图

本节提出一种适合面向方面程序分析的调用图,后继的优化工作基于该调用图.

传统调用图是一种表示方法间调用关系的有向图.传统调用图不适合面向方面程序分析:

- 首先,传统调用图只表示方法执行这一种程序执行点,无法表达面向方面语言引入的其他程序执行点(例如,方法调用、通知执行、属性访问等);
- 其次,传统调用图无法直接描述与节点相关的对象类型(例如,与执行节点关联的执行对象的类型).

为此,我们提出面向方面调用图 ACG,它在传统调用图的基础上作了如下扩展:

首先,我们扩充了节点的种类,使之能够表示面向方面语言预定义的所有程序执行点.为此,节点包含了表示自身种类的字段 k.例如,要优化只含上述切点子集的动态通知只需 3 类节点,分别是方法执行节点($k=mexec$)、方法调用节点($k=mcall$)和通知执行节点($k=aexec$).如需引入新的节点,则只要扩充 k 的取值即可.

其次,节点包含字段 C_{this},C_{target},分别表示当前执行对象和目标对象的类型.执行对象和目标对象的含义与节点的种类有关.对于方法执行节点和方法调用节点,执行对象是程序运行到该节点时伪变量 *this* 指向的对象,目标对象是操作的作用对象.对于通知执行节点,执行对象是其配合 *this* 切点暴露给通知体的对象,目标对象是其配合 *target* 切点暴露给通知体的对象.我们将二元组$\langle C_{this},C_{target}\rangle$定义为节点的类型.

节点类型分为静态和动态两种.静态节点类型是构造 ACG 时赋予节点的类型.静态方法执行节点类型的 C_{this} 和 C_{target} 值都是方法所属的类,例如,图 2 中节点 8 的静态类型为$\langle A,A\rangle$.静态方法调用节点类型的 C_{this} 值是调用该方法的类,C_{target} 值是构造 ACG 时分析获得的调用对象的类,例如,图 2 中节点 5 的静态类型为$\langle B,A\rangle$.静态通知执行节点类型的 C_{this} 值是其声明配合 *this* 切点静态暴露出的类型,C_{target} 值是其声明配合 *target* 切点静态暴露出的类型,若没有暴露,则为 *Object*.例如,*ad*3 通过声明 *after*(*B b*)和 *target*(*b*)暴露出类型 *B*,因而节点 12 的静态类型为$\langle Object,B\rangle$.

动态节点类型是程序沿 ACG 上某条路径执行到该节点时,执行对象和目标对象的实际类型,需要沿路径推导求得.若节点 B 类型依赖于节点 A,则 B 的类型推导自 A;否则,等于自身的静态类型.

节点间的类型依赖关系与节点的种类有关,以方法执行、方法调用及通知执行 3 类节点为例,共有以下 5 种依赖:

(1) 被调方法执行节点的 C_{this} 和 C_{target} 等于被调方法调用节点的 C_{target}.这种依赖发生在:当通过非 *this* 伪变量调用子类继承的方法时,例如,图 2 中节点 11 和节点 4 间的依赖;通过 *this* 伪变量调用方法时,例如,节点 11 和节点 9 间的依赖;通知体内通过 *this* 切点(或 *target* 切点)中的参数调用时,例如,节点 14 和节点 13 间的依赖.

(2) 被调方法调用节点的 C_{this} 和 C_{target} 等于施调方法执行节点的 C_{target}.这种依赖发生在通过 *this* 伪变量调用方法时,例如,节点 9 和节点 8 之间的依赖.

(3) 通知执行节点的 C_{this} 和 C_{target} 分别等于前驱节点的 C_{this} 和 C_{target}.这种依赖发生在调用通知体时,例如,节点 12 和节点 4 之间的依赖.

(4) 被调方法调用节点的 C_{target} 等于通知执行节点的 C_{this}.这种依赖发生在通知体内通过 *this* 切点中的参数调用方法时.

(5) 被调方法调用节点的 C_{target} 等于通知执行节点的 C_{target}.这种依赖发生在通知体内通过 *target* 切点中的参数调用方法时,例如,节点 13 和节点 12 之间的依赖.

最后,每条有向边包含字段 d,用以表示边的种类.我们用 d=0 表示该边连接的节点间没有依赖,用 d=1~5 分别表示上述的 5 种依赖(例如,图 2 中部分边上标注了种类).我们称 d=0 的边为类型不依赖边,d>0 的边为类型依赖边.

我们给出面向方面调用图的具体定义.

定义 1. 面向方面调用图是一个有向图 $ACG=\langle N,E,r\rangle$,其中:

(1) N 是节点集,$\forall n\in N,n=\langle k,ID,C_{this},C_{target}\rangle$,$k$ 表示节点的种类,ID 表示节点的标识,C_{this} 表示执行对象的类型,C_{target} 表示目标对象的类型,节点的类型为二元组$\langle C_{this},C_{target}\rangle$.

(2) E 是边集,$\forall e\in E,e=\langle d,n_{head},n_{tail}\rangle$,$d$ 表示边的种类,n_{head} 表示头节点,n_{tail} 表示尾节点.

(3) r 是入口节点,通常为 *main* 方法执行节点.

3 动态通知编织优化

我们提出的优化方法适用于 AspectJ 切点全集,只需为每类切点定义所需的具体操作.为了便于论述,本文就不针对每类切点一一列举了,而是首先给出方法的一般步骤,然后以第 2 节的切点集为例介绍具体的优化过程.其他切点的相关操作可类似定义.优化方法的一般步骤如下:

(1) 将方面中的所有切点转换成正规表达式的形式.

(2) 通过对整个程序进行快速类型分析,确定每个方法参数、方法体内 *this* 指针、方法返回值以及每个通知参数的类型范围.

(3) 结合上下文不敏感、方法体内流敏感的指向分析,构建基程序的 ACG.然后,根据切点正规表达式确定每个通知在基程序 ACG 上的待分析织入点,将通知关联到这些点上,构造出整个程序的 ACG.

(4) 采用高斯消去法在 ACG 上求解从起点到达每个待分析织入点的路径表达式,并根据边的类型推导函数推导表达式中节点的类型.

(5) 将路径表达式转换成调用栈表达式,并与切点表达式匹配,根据匹配结果确定通知调用的织入方式.

其中的几个关键步骤是切点的转换、ACG 图的构造、调用栈表达式的求解以及调用栈表达式与切点表达式的匹配.下面以第 2 节的切点集为例,分别详述这些步骤.

3.1 切点转换

为便于与调用栈表达式匹配,我们将切点转换成如下正规表达式的形式:

切点表达式::=基本项{;基本项}*

基本项::=(k,ID,C_{this},C_{target})|true

其中,k,ID,C_{this} 和 C_{target} 的含义与第 2 节中的定义相同.true 表示与任意的基本项匹配.例如,切点

call(*m1()) && *target*(*B*) && *cflow*(*call*(*m3()) && *target*(*B*))

转换成的切点表达式为

(*mcall*,*m*1,_,*B*);true*;(*mcall*,*m*3,_,*B*);true*.

下面首先给出一个辅助定义及基本转换函数,然后给出切点转换算法.

定义 2. 设 R_1,R_2 是切点表达式,定义切点表达式的∩运算如下:

(1) 如果 R_1 和 R_2 都是基本项,则 R_1∩R_2 生成新的基本项,项中各字段的值取自 R_1 或 R_2 中的相应字段.

(2) 如果 R_1 是基本项, R_2 形如"true*;R_3;切点表达式"且 R_3≠true,若 R_1=R_3,则 R_1∩R_2=true*; R_3;切点表达式;若 R_1≠R_3,则 R_1∩R_2=R_1;true*;R_3;切点表达式.

(3) 如果 R_1 是基本项, R_2 形如"true+;R_3;切点表达式"且 R_3≠true,则 R_1∩R_2=R_1;true+;R_3;切点表达式.

算法 1 给出了基本转换函数.

算法 1. 基本转换函数 ***t***.

输入:切点;

输出:切点表达式.

begin

(1) if (切点==切点 1&&切点 2) then return ***t***(切点 1)∩***t***(切点 2) end if

(2) if (切点==*cflow*(切点 1)) then return "(true)*;"***t***(切点 1) end if

(3) if (切点==*cflowbelow*(切点 1)) then return "(true)+;"***t***(切点 1) end if

(4) if (切点==*call*(方法名)) then return "(*mcall*,方法名,_,_)" end if

(5) if (切点==*execution*(方法名)) then return "(*mexec*,方法名,_,_)" end if

(6) if (切点==*adviceexecution*()) then return "(*aexec*,_,_,_)" end if

(7) if (切点==*target*(类名)) then return "(_,_,_,类名)" end if

(8) if (切点==*this*(类名)) then return “(_,_,类名,_)” end if

end

切点转换算法 *Trans* 如算法 2 所示.

算法 2. 转换算法 *Trans*.

输入:通知的切点;

输出:切点表达式集.

begin

(1) 运用分配率提取出切点中的||运算符,将原切点转换成“切点 1||…||切点 *n*”的形式,收集切点 1 至切点 *n*,组成待转换切点集;

(2) 针对待转换切点集中的每个切点调用基本转换函数 ***t***(切点)转换成正规表达式;

end

3.2 ACG构造

AspectJ 程序 ACG 的精度对优化的结果有较大的影响,粗糙的 ACG 会包含较多的虚假路径,影响调用栈与切点的匹配结果.但不论 ACG 的精度如何,我们的优化方法都是安全的,即优化后不会产生错误的通知调用.

我们采用方法间快速类型分析[9]与方法内流敏感的指向分析[10]相结合的方法构造 ACG(构造不考虑 Java 库函数).由于方面是被织入的,在程序中没有显式的通知调用点,所以,AspectJ 程序 ACG 的构造不同于 Java 程序调用图的构造,需分为两步:先构造基程序的 ACG,再关联通知.

在构造基程序 ACG 时,首先采用快速类型分析方法确定每种方法的参数、局部变量、返回值及 *this* 指针的类型范围,以及每个通知的参数、局部变量、返回值(around 通知具有返回值)的类型范围.然后,由基程序的入口方法开始,在方法体内作流敏感的指向分析,并实时地构造基程序 ACG.

所谓关联通知,即寻找通知的所有可能织入点,由织入点引出调用边到通知上,并构造通知体的 *ACG*.

通知的所有可能织入点通过察看切点表达式集中每个表达式的第 1 项(即第 1 个“;”前的项)寻找:

(1) 如果该项形如“(*mcall*,方法名,_,类名)”,表示可能织入点是类型为“类名”的方法调用点,这对应于 ACG 上两类节点:一类是静态节点类型 C_{target} 字段值为该类或其子类,且类型不依赖于前驱节点的方法调用节点;另一类是静态节点类型 C_{target} 字段值为该类或其父类,且类型依赖于前驱节点的方法调用节点.例如,图 2 中 *ad*1 在节点 10 上的关联就属于后者.

(2) 如果该项形如“(*mexec*,方法名,_,类名)”或“(*mexec*,方法名,类名,_)”,表示可能织入点是类型为“类名”的方法执行点,这对应于 ACG 上两类可能节点:一类是静态节点类型 C_{this} 字段值为该类或其子类,且类型不依赖于前驱节点的方法执行节点;另一类是静态节点类型 C_{this} 字段值为该类或其父类,且类型依赖于前驱节点的方法执行节点.例如,图 2 中 *ad*4 在节点 14 上的关联也属于后者.

确定织入点后,对通知体采用与方法体相同的分析方法,构造其 ACG.

关联通知操作需要反复进行,直至没有新的织入点为止.至此,程序的 ACG 构造完毕,图中被通知关联的节点成为待分析织入点,留待下一步分析.

3.3 调用栈表达式求解

假设构造好的 ACG 为 G_1,并且确定了待分析织入点,下一步是求解这些节点的调用栈表达式.调用栈表达式描述了程序执行到某节点时调用栈的所有模式,在 ACG 上看是从起点到该节点路径的逆序列.我们将调用栈表达式定义成四元组$\langle k,ID,C_{this},C_{target}\rangle$上的正规表达式.例如,图 2 中节点 9 的调用栈表达式为

$$(mcall,m1,\{A,B\},\{A,B\});(mexec,m2,\{A,B\},\{A,B\});$$

$$\{(mcall,m2,B,A)+(mcall,m2,B,B);(mexec,m3,B,B);(mcall,m3,B,B);\}(mexec,main,B,B).$$

我们分 3 步求解 ACG 上调用栈表达式:首先求出图上包含由起点至该节点所有路径的路径表达式,然后推导表达式中节点的动态类型,并且在推导的过程中实时地修正路径表达式,最后再将路径表达式转换成调用栈

表达式.

我们采用高斯消去法[11]求解路径表达式,所得结果是由边集组成的正规表达式.例如,图 2 中节点 9 的路径表达式为$(e_1e_2+e_3e_4e_5e_6)e_7$.

在给出节点动态类型推导算法之前,我们首先给出边 e 的类型推导函数的相关定义如下:

定义 3. 设 RT 是程序中引用类型的集合,$\forall rt_1,rt_2\in RT$,定义 rt_1 和 rt_2 之间的偏序关系为 $rt_1\leq_{rt}rt_2$ 当且仅当 rt_1 是 rt_2 的父类.

定义 4. 设 $NT\in RT\times RT$ 是节点类型的集合,$e.n_{tail}.\langle C_{this},C_{target}\rangle$表示边 e 尾节点的静态类型,令 $t\in NT$,则边 e 上的类型推导函数 $f_e(t)\in NT\to NT$ 为:

a) 若 $e.d$==0,则 $f_e(t)=\langle e.n_{tail}.C_{this},e.n_{tail}.C_{target}\rangle$.

b) 若 $e.d$==1,则 $f_e(t)=\langle t.C_{target},t.C_{target}\rangle$.

c) 若 $e.d$==2 且 $t.C_{target}\leq_{rt}e.n_{tail}.C_{target}$,则 $f_e(t)=\langle\_,\_\rangle$;否则,$f_e(t)=\langle t.C_{target},t.C_{target}\rangle$.

d) 若 $e.d$==3,且 $t.C_{this}\leq_{rt}e.n_{tail}.C_{this}$,或 $t.C_{target}\leq_{rt}e.n_{tail}.C_{target}$,则 $f_e(t)=\langle\_,\_\rangle$;否则,$f_e(t)=\langle t.C_{this},t.C_{target}\rangle$.

e) 若 $e.d$==4,则 $f_e(t)=\langle e.n_{tail}.C_{this},t.C_{this}\rangle$.

f) 若 $e.d$==5,则 $f_e(t)=\langle e.n_{tail}.C_{this},t.C_{target}\rangle$.

节点动态类型推导算法 Deduce 如算法 3 所示,其中,V_p 是路径表达式 P 的节点集,$PRED(v)$是 V_p 中节点 v 的前驱节点集,$SUCC(v)$是 V_p 中节点 v 的后驱节点集,$StaType(v)$和 $DynType(v)$分别表示节点 v 的静态类型和动态类型.

算法 3. 类型推导算法 Deduce.

输入:路径表达式 P;

输出:含有节点动态类型信息的路径表达式 P.

```
begin
(1)  foreach v∈V_P
(2)      DynType(v)=⟨_,_⟩;
(3)  end for
(4)  worklist={头节点 v_0};
(5)  while worklist≠{}
(6)      take v from worklist;
(7)      foreach q∈PRED(v)
(8)          Type_{q→v}=f_e(DynType(q));   /*e 是 q→v 的边*/
(9)          if (Type_{q→v}=⟨_,_⟩) then
(10)             删除包含 e 作为必经路径的子表达式,并调整 V_P
(11)         end if
(12)     end for
(13)     if (PRED(v)为空) then DynType(v)=StaType(v);
(14)     else DynType(v)=∪_{q∈PRED(v)}Type_{q→v};   /*要求 Type_{q→v}≠⟨_,_⟩*/
(15)     end if
(16)     if (DynType(v) changed) then
(17)         worklist=worklist∪SUCC(v);
(18)     end if
(19) end while
end
```

类型推导完毕后,将路径表达式中的形如 e^*的项约简为 e,再把它转换成由节点表示的正规表达式,该表达

式是一个调用序列表达式,最后将它转置得到调用栈表达式.

3.4 与切点表达式匹配

假设某织入点的调用栈表达式为 E_1,切点表达式为 E_2,优化的最后一步是检查 E_2 是否包含 E_1.设 E_1 表示的正规语言为 L_1,E_2 表示的正规语言为 L_2,原问题是 $L_1 \subseteq L_2$ 的判定问题.由于相应的判断算法复杂度较高[12],因此,我们考虑其他解决方法.

考察 E_1 的形式不难发现,E_1 不含*操作,只含+和•操作,而•对+满足分配率,因此,可以把+提取到最外面,将 E_1 转换成$(w_1+w_2+\ldots+w_n)$的形式,其中每个 w_i 都是具体的串.这样,原问题转换成 $w_i \in E_2 (i=1,\ldots,n)$的判定问题,可以使用高效的自动机判定算法[12].

最后根据判定结果决定通知调用的织入方式.令 $D(w)$是表示 $w \in E_2$ 的布尔函数,那么:

1) 若$\wedge_{i=1,\ldots,n} D(w_i)$=true,则通知调用织入此节点;
2) 若$\vee_{i=1,\ldots,n} D(w_i)$=false,则通知调用不织入此节点;
3) 其他情况下通知调用仍需条件织入.

4 相关工作及总结

AspectJ 程序的静态织入方式具有不需要修改虚拟机的优点,而如何进一步减少动态切点的匹配开销是静态织入优化的一个重要课题.Masuhara 使用部分求值技术,利用编译时获得的静态信息确定织入点.该方法对静态切点有效,但对动态切点的优化效果不好.AspectJ 编译程序针对 *cflow* 的匹配作了优化,使得匹配时只需察看栈顶元素.上述两种方法都没有尝试确定动态切点的织入点.Avgustinov 等人提出的方法只考虑了 *cflow* 的优化,忽略了对象运行时类型的分析,因此,该方法不适用于优化具有此类动态性的切点,而在所有的动态切点中又以这种切点居多.

我们曾在指向分析以及 AOP 语言程序分析方面进行了深入的研究[13,14],本文在此基础上进一步研究 AspectJ 程序的编译优化,提出了通用的动态通知优化方法,其具有以下优点:

- 面向方面调用图 ACG 具有良好的可扩充性,能够表示 AspectJ 的各种程序执行点,并且节点具有类型信息,能够作适当的类型分析;
- 能够优化具有两种动态特性的通知;
- 通过扩展 ACG 节点的种类,并且给出新的推导函数,可以优化包含 AspectJ 切点全集的动态通知
- 可以比较容易地推广到其他面向对象的 AOP 语言中,例如 AspectC++等.

我们将在以下 3 个方面进行更深入的研究:首先是 ACG 的构造算法,进一步提高构造效率和精确度;其次是 ACG 上调用栈表达式的增量式求解方法;再次是更加高效的切点和调用栈表达式的匹配算法.

致谢 在此,我们向对本文的工作给予支持和建议的同行表示感谢.同时,对审稿人提出的有益建议表示感谢.

References:

[1] Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier J, Irwin J. Aspect-Oriented programming. In: Aksit M, Matsuoka S, eds. Proc. of the ECOOP'97. LNCS 1241, Jyvaskyla: Springer-Verlag, 1997. 220−242.

[2] Laddad R. AspectJ in Action: Practical Aspect-Oriented Programming. Greenwich: Manning Publications, 2003.

[3] Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. An overview of AspectJ. In: Knudsen JL, ed. Proc. of the ECOOP 2001. LNCS 2072, Heidelberg: Springer-Verlag, 2001. 327−353.

[4] Masuhara H, Kiczales G, Dutchyn C. Compilation semantics of aspect-oriented programs. Technical Report, TR #02-06, Iowa State University, 2002.

[5] Wand M, Kiczales G, Dutchyn C. A semantics for advice and dynamic join points in aspect-oriented programming. ACM Trans. on Programming Languages and Systems, 2004,26(5):890−910.

[6] Sereni D, de Moor O. Static analysis of aspects. In: Aksit M, ed. Proc. of the Aspect-Oriented Software Development (AOSD 2003). New York: ACM Press, 2003. 30−39.

[7] Avgustinov P, Christensen AS, Hendren L, Kuzins S, Lhotak J, Lhotak O, de Moor O, Sereni D, Sittampalam G, Tibble J. Optimising AspectJ. In: Sarkar V, Hall M, eds. Proc. of the Conf. on Programming Language Design and Implementation (PLDI 2005). New York: ACM Press, 2005. 117−128.

[8] Hilsdale E, Hugunin J. Advice weaving in AspectJ. In: Lieberherr K, ed. Proc. of the Aspect-Oriented Software Development (AOSD 2004). New York: ACM Press, 2004. 26−35.

[9] Bacon DF, Sweeney PF. Fast static analysis of C++ virtual function calls. In: Berman AM, ed. Proc. of the Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA'96). New York: ACM Press, 1996. 324−341.

[10] Hirzel M, Dincklage DV, Diwan A, Hind M. Fast online pointer analysis. ACM Trans. on Programming Languages and Systems, 2007,29(2).

[11] Tarjan RE. Fast algorithms for solving path problems. Journal of the ACM, 1981,28(3):594−614.

[12] Hopcroft JE, Motwani R, Ullman JD. Introduction to Automata Theory, Languages, and Computation. 2nd ed., Addison-Wesley, 2001.

[13] Zhou XY, Xu BW, Shi L, Chen L. Express calculation decomposition with extended aspect-oriented programming language. Journal of Electronics & Computer Science, 2005,7(1):89−99.

[14] Qian J, Xu BW, Ming HB. Interstatement must aliases for data dependence analysis of heap locations. In: Proc. of the Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007). New York: ACM Press, 2007. 17−24.

附录 实例分析

下面是一个 AspectJ 程序,包含了 4 种常见的动态通知.图 2 是程序的 ACG,其中部分节点上标注了静态类型,部分依赖边上标注了边的种类.

AspectJ 程序实例

```
class A{
    public void m1() {…}
    public void m2() {m1();}}
class B extends A{
    public void m3() {m1();m2();}
    public void m4() {…}
    public static void main(String[]args){
            A a=new A(); B b=new B();
            a.m1(); a.m2(); b.m1(); a=b; a.m3();
        }}
    public aspect Aspect1 {
    ad1 before(): call(*m1()) && target(B){..}
    ad2 before(): call(*m1()) &&
        cflow(call(*m3()){..}
    ad3 after(B b): call(*m1()) && target(b)
        && cflow(call(*m3()) && target(B))
        {b.m4();}
    ad4 void around(): execution(*m4()) &&
        this(B) && cflow(adviceexecution()){.}}
```

分析结果见表 1.其中,打"√ "表示确定织入,打"×"表示确定不织入,打"?"表示条件织入.节点 9 上通知的织

入方式确定过程为:首先,节点 9 的路径表达式为$(e_1e_2+e_3e_4e_5e_6)e_7$,其次,式中节点动态类型推导过程如下:

节点 1: $DynType(1)=StaType(1)=\langle B,B\rangle$; 节点 2: $DynType(2)=StaType(2)=\langle B,A\rangle$;

节点 3: $DynType(3)=StaType(3)=\langle B,B\rangle$; 节点 6: $DynType(6)=StaType(6)=\langle B,B\rangle$;

节点 7: $DynType(7)=f_{e5}(DynType(6))=\langle B,B\rangle$;

节点 8: $DynType(8)=f_{e6}(DynType(7))\cup StaType(8)=\langle B,B\rangle\cup\langle A,A\rangle=\langle\{A,B\},\{A,B\}\rangle$;

节点 9: $DynType(9)=f_{e6}(DynType(8))=\langle\{A,B\},\{A,B\}\rangle$;

由此可知节点 9 处的调用栈表达式为

$$(mcall,m1,\{A,B\},\{A,B\});(mexec,m2,\{A,B\},\{A,B\});$$

$$\{(mcall,m2,B,A)+(mcall,m2,B,B);(mexec,m3,B,B);(mcall,m3,B,B);\}(mexec,main,B,B).$$

由此可知 $ad1,ad2,ad3$ 在节点 9 上需条件织入.

Table 1 The result of optimization

表 1 实例优化结果

Advice \ Weaving point	5	9	10	4	14
*ad*1	×	?	√	√	
*ad*2	×	?	√	×	
*ad*3	×	?	√	×	
*ad*4					√

曹璟(1980－),男,江苏通州人,博士生,主要研究领域为程序设计语言,程序分析.

钱巨(1981－),男,博士生,主要研究领域为程序分析与测试.

徐宝文(1961－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为程序设计语言,软件工程,并行与网络软件.

杨彬(1980－),男,博士生,主要研究领域为软件工程.

周晓宇(1972－),男,副教授,主要研究领域为程序设计语言,软件分析与测试.