Soot 的 Java 程序控制流分析及图形化输出①

Java Program Control Flow Analysis and Graphic Output of Soot

李远玲　陈　华　刘　丽　(北京系统工程研究所 北京 100101)

摘　要： Soot 是一个 Java 编译优化框架，可以利用它实现 Java 字节码程序的数据流分析和控制流分析。在深入分析 Soot 控制流生成机制的基础上，详细叙述了利用 Soot 分析 Java 类的控制流并生成其控制流图的方法和过程，同时提出了将 Soot 生成的抽象的控制流图进行图形化输出的方法。

关键词： Java Soot 控制流分析 DOT 语言 图形化输出

程序分析具有广泛的应用。为了实现程序优化，编译器必须能够刻画出程序的控制流及其数据操纵特征，以便优化执行流程、删除无用代码，从而提高运行效率。另一方面，为了对程序进行安全分析，寻找其薄弱环节，需要确定程序中与数据处理有关的全局信息，也就是对程序进行数据流分析。控制流分析是数据流分析的基础，通过生成程序的控制流图，可以全面了解程序中每一个过程内的控制流层次结构以及过程间的全局调用关系。

Soot 是 McGill 大学的 Sable 研究小组自 1996 年开始开发的 Java 字节码分析工具。它完全用 Java 实现，最新版本是 2.3.0 版。Soot 提供了多种字节码分析和变换功能，利用它可以很方便地进行程序的分析、编译优化和变换。通过 Soot 提供的过程内和过程间的分析与优化支持，用户可以很方便地获得每个方法的控制流图和各个折衷级别上的全局调用图。本文深入分析了 Soot 提供的过程内控制流图的生成机制，利用其获取被分析软件的控制流信息，并将该信息以图形化的形式显示出来。Soot 提供的工具包将控制流信息转换成 DOT 语言表示的形式。DOT 语言是一种通用的结构化图形描述语言，目前存在多种工具可以将这种语言描述的图形转换成不同格式的可视化图形。

1　Soot概述

Soot 提供了一个 Java 字节码优化框架[1,2]。它既可以作为一个独立的工具用于优化和检查类文件，也可以作为一个开发 Java 字节码优化和变换的框架。图 1 是 Soot 的使用模式。

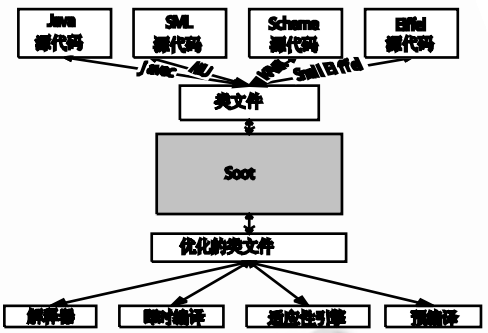


图 1　Soot 的使用模式

如图 1 所示，Soot 的输入是多源(如 javac 编译器)产生的字节码。采用 Soot 进行代码的变换和优化，产生新的已被优化的类文件。优化后的字节码程序可以在任何标准的 Java 虚拟机上执行。

Soot 框架提供了一组中间表示法，其分析和变换建立在这些中间表示法之上；另外还提供了一组直接用于优化 Java 字节码的 API。Soot 的扩展机制以 Pack 为中心，一个 Pack 包括若干个变换。用户可以自行设计新的变换，将其加入到 Soot 的调度执行过程中以实现特定的程序分析。本文将对该框架的这些特性进行系统的描述。

1.1 中间表示法和变换

对于优化实现来讲，将 Java 字节码这种基于栈的代码作为中间表示法不是好的选择。原因有这样几个

① 收稿时间:2009-02-11

方面。由于 Java 字节码包括表达式指令(对操作数栈产生影响的指令,如装入操作数指令)和动作指令(产生副作用的指令，如修改域指令)。为了确定动作指令要操作的表达式，需要分析表达式指令和重新构造表达式树，因此 Java 字节码的表达式是不清晰的。另外，由于 Java 字节码表达式的长度是可变的，增加了构造表达式树的复杂性，使一些分析变得极为复杂。如常见的子表达式删除这样的分析，只有使用简单的 3 地址代码表达式才能有效地进行。再者，栈代码的碎片形式同样会使分析和变换变得很复杂。原因在于表达式被分割成碎片形式并与动作指令分离，这些指令与其它指令交织在一起越过了控制流的边界，代码分析和变换要跟踪和维护这些隔开的碎片很困难。为了解决基于栈的 Java 字节码在优化和变换中存在的问题，Soot 提供了四种中间表示法(Intermediate Representation，简称 IR)来解决这些难题。每一种 IR 都有不同的抽象级别，在分析过程中具有不同的优势。它们是 Baf、Grimp、Jimple 和 Shimple。Baf 是一种简洁的基于栈的 Java 字节码中间表示法，它的无常数池、完全类型化等特性使其处理起来比较简单。使用这种基于栈的中间表示法为的是简化一定要在栈代码上执行的分析和变换的开发。Jimple 是一个紧凑、无栈、类型化的 3 地址代码中间表示法，相对 baf 来讲它的结构更清晰，每条指令都有显式的操作数，更适用于程序分析和代码优化。Grimp 是 Jimple 聚合版本，与 Jimple 的平面表达式不同的是它可以为表达式构造树形结构，更接近 Java 源代码，适用于反汇编并

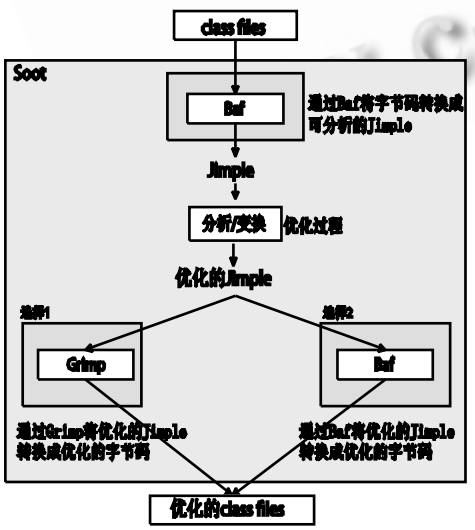


图 2 Soot 框架中代码的变换和优化流程

便于阅读。Shimple 是 Jimple 的 SSA(静态单赋值，一种较新的中间代码表示，能有效地将程序中的运算值与它们的存储位置分开，提供一种更有效的优化形式)变种。Jimple 是 Soot 的核心 IR。图 2 是 Soot 框架中代码的变换和优化过程。

由图 2 可见，Soot 提供了上述不同中间表示法之间的变换。

1.2 基本的 Soot 对象

Soot 有一个复杂的类层次结构，这里介绍基本的 Soot 应用程序接口所涉及的一些最重要的类，它们是 Scene、SootClass、SootField、SootMethod、Body、Local、Trap、Unit。

Scene 类表示整个分析发生的环境。通过 Secne 可以设置提供给 Soot 分析的所有的应用类、包含 main 方法的类和有关过程间分析的访问信息。SootClass 表示装入到 Soot 中或由 Soot 创建的单个类。在 Scene 中的各个类都是由 SootClass 类的实例表示的。一个 SootClass 包含一个 Java 类所有相关的信息，如这个类的类名、修饰符、超类、SootField 链、SootMethod 链等。SootMethod 表示一个类中的单个方法。SootField 表示类中的一个域。

方法的实现通过 Body 接口的实现来表示。在 Soot 中，一个 Body 隶属于一个 SootMethod，即 Soot 用一个 Body 为一个方法存储代码。分析应用可以使用 Body 访问各种信息，如一组声明的局部变量、组成方法体的语句和方法体中处理的所有异常等。Soot 中有四种类型的 Body：BafBody、JimpleBody、ShimpleBody 和 GrimpBody，对应四种不同的中间表示法。一个 Body 中有三个主链(Chain)，分别是 Units 链、Locals 链和 Traps 链。Locals 指的是 Body 中的局部变量。Units 指的是方法体中的语句，也就是说，一个语句由一个 Unit 接口表示。Traps 表示支持 Java 异常处理。

1.3 Soot 的执行过程及其扩展机制

Soot 的执行分为多个阶段。JimpleBody 就是由被称为 jb 的阶段构造的，jb 阶段本身又由几个子阶段组成。

Soot 命令行中的阶段选项提供了一种从 Soot 的命令行改变一个阶段行为的方式。阶段选项的格式是：-p phase.name option:value。在命令行中没有明确指定的话，每个阶段选项都有一个缺省值。如果要

执行某一个阶段或子阶段的话，应该设置该阶段的选项为“enable:true”。

在 Soot 里，每个阶段都由对应的 Pack 实现。Pack 是一组变换器(transformer)，一个变换器对应着相应的子阶段。当 Pack 被调用时，它按照顺序执行每一个变换器。提供扩展机制的是那些允许用户自定义变换(transformation)的 Pack：jtp (Jimple transformation pack) 和 stp (Shimple transformation pack)。在不改变 Soot 自身的情况下，用户可以编写满足自身需求的类(变换器)，然后将其注入到这些 pack 中，之后调用 soot.Main，使其进入到 Soot 的调度执行过程中。

Soot 变换器通常是继承了 BodyTransformer 或 SceneTransformer 的类的实例。BodyTransformer 针对单个方法体(也就是过程内)进行变换，SceneTransformer 针对整个应用(也就是过程间)进行变换。在这两种情况下，变换器类都必须重构 internalTransform 方法，对要被分析的代码执行某种变换。

1.4 Soot 包的构成

为了构造 Soot，Soot API 被分成多个程序包。包层次结构的根节点是 soot，其中包含供不同 IR 共享的基础 Soot 类。其下包含的子包有：不同 IR 独立的程序包，包含其公共类、内部实现专用类及变换和优化代码等；coffi 包含来自 Coffi 工具的包；options 提供命令行选项分析程序；toolkits 提供产生和处理各种控制流图的工具包、流分析框架等；util 存放有用的应用类，如实现 DOT 图的类。控制流图可视化工具 CFGViewer 存储在 soot.tools 包中，其执行过程会调用其它程序包中的功能，从而实现控制流图的生成和输出。

2 Soot的控制流分析

2.1 控制流图

正如前面所提到的，执行数据流分析，需要某种描述数据流如何通过一个程序的结构，如控制流图。

一个过程的控制流是用含有结点集合和边集合表示的有向图[3]。图中的每一个结点代表一个基本块(基本块指一段只能从它的开始处进入，并从它的结束处离开的线性代码序列)，另外还有一个入口结点和出口结点，以及一个边的集合。一个包含了结点集合和边集合的流图可以表示为：。在图中一个结点 b 可以有多个后继结点，这样的结点被称为分支结点。假设表示结点的后继集合，形式上可以表示为：

$$Succ(b)=\{n\in N\mid \exists e\in E \quad \text{其中} \quad e=b\to n\}$$

$b\to n$ 表示 b 结点到结点 n 的一条边。

在图中一个结点也可以有多个前驱结点，这样的结点称之为汇合结点。若用 $\Pr ed(b)$ 表示结点 $b\in N$ 的前驱集合，则其形式上可表示为：

$$Pred(b)=\{n\in N\mid \exists e\in E \quad \text{其中} \ e=n\to b\}$$

2.2 Soot 的控制流表达方法

Soot 中有两种类型的控制流图，UnitGraph 和 BlockGraph。BlockGraph 是基本控制流图，结点由基本块组成；UnitGraph 的结点是 Units。两种类型图的形式化表示是相通的。UnitGraph 的优点之一是简化了控制流图，没有了基本块的概念，使用它可以简化传统的和非传统的固定点数据流分析(fixed point data flow analyses)的实现。它的主要缺点是由于没有基本块，增加了分析的时间开销。

UnitGraph 和 BlockGraph 都需要实现 DirectedGraph 接口。该接口定义的方法可获取下列信息：图的入口结点和出口结点；一个给定结点的前驱结点和后继结点；一个以某种未明确规定的顺序和结点数在图上进行迭代的迭代器。

在此基础上可以开发对任意 DirectedGraph (有向图)进行处理的方法。事实上 UnitGraph 和 BlockGraph 都是抽象类，Soot 提供了若干可将这些抽象类实例化的子类，如 CompleteUnitGraph，BriefUnitGraph，CompleteBlockGraph 和 BriefBlockGraph 等。Soot 在包 soot.toolkits.graph 中提供了这些不同类型控制流图的实现。

2.3 Soot 的控制流图的实现

Soot 提供了一个基础结构用于审查和修改以控制流图形式出现的 Unit 链。这里主要描述结点为 Soot Units 的控制流图的实现。UnitGraph 是这类控制流图的基类，为构建控制流图提供了基本功能部件。通过它可以实现三种不同类型的控制流图[2]：BriefUnitGraph，ExceptionalUnitGraph 和 TrapUnitGraph。BriefUnitGraph 相对较为简单，这类控制流图上没有 unit 到异常处理程序的边。ExceptionalUnitGraph 包含从 throw 子句到处理程序(即 catch 模块)的边，即 Soot 中的 Trap。此外，这类图

还涉及可能由虚拟机隐含抛出的异常，比如数组指针越界异常。对于每一个可能抛出隐含异常的 unit，从每一个前驱结点到各自处理程序的第一个 unit 都有一条边。控制流分析一般使用这类控制流图。TrapUnitGraph 类似于 ExceptionalUnitGraph，涉及可能抛出的异常。但是它们之间有三点主要的不同：从每一个要捕获的 unit(例如，在 try 模块中的 unit)到 trap 处理程序加入一条边；从可能抛出一个隐含异常的 unit 的前驱结点到 trap 处理程序没有边；从可能抛出一个隐含异常的 unit 到 trap 处理程序总是有一条边。

使用 Soot 构造一个方法的控制流图，只需将该方法的 body 传递给某类控制流图的构造函数即可。例如，构造一个 ExceptionalUnitGraph 可以调用下面的语句：

UnitGraph g=new ExceptionalUnitGraph (body);

3 控制流的图形化输出

Soot 控制流图可视化工具 soot.tools.CFGViewer 利用 Soot 框架提供的功能，分析类中每个方法的控制流并生成以 DOT 语言描述的控制流图。这是一种抽象的图形描述语言，要想控制流图以几何图形的方式显示出来，需要将其转换成可视化图形格式，如 gif、svg、ps 等。

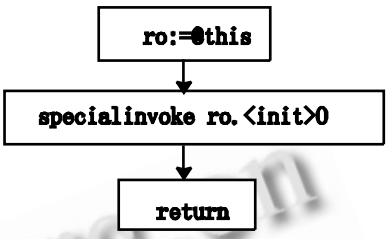
3.1 DOT 语言

DOT 语言是描述结构化图形的语言[4]，它支持三种对象：graphs(图)、nodes(结点)和 edges(边)。主图(最外层)是有向图(digraph)或非有向图(graph)，其中还可以定义子图。下面是用 DOT 语言描述的一个图形：

```
//创建有向图，图的名称为"void <init>()"
1: digraph "void <init>()" {
//设置图的字体为宋体，字号为10
2: graph [fontname="sursong",fontsize=10]
//图的标识名称是"方法void <init>()控制流图"
3: label="方法void <init>()控制流图";
4: node [shape=box];//设置图中结点的形状为方框
//创建结点0，结点名为"r0:=@this",结点填充为灰色
5: "0" [style=filled,fillcolor=gray,label="r0 := @this"];
//创建结点1，结点名为"specialinvoke r0.<init>()"
6: "1" [label="specialinvoke r0.<init>()"];
7: "0"->"1";//创建从结点0到结点1的一条有向边
//创建结点2，结点名为"return",颜色填充为浅灰
8: "2" [style=filled,fillcolor=lightgray,label="return"];
9: "1"->"2";//创建从结点1到结点2的一条有向边
10:}
```

图 3 DOT 语言编写的图形文件

图的构造方法是，当一个结点的名字第一次出现在文件中时，这个结点就被创建了；当结点由边操作符“->”连接时，这条边就被创建了。下面是该文件所绘制的图形。



方法void<init>()控制流图

图 4 根据图形文件绘制的图形

3.2 Soot 控制流图的产生和显示

如前所述，CFGViewer 是 Soot 控制流图可视化工具。它可以为类文件中所有方法生成以 DOT 语言描述的控制流图。如果方法体中的语句使用的是 Jimple 中间表示法，那么被变换的对象就被称为 JimpleBody。要想生成方法的控制流图，首先需创建一个 BodyTransformer 对象。图 5 是 CFGViewer 程序的代码片断。

CFGViewer 程序中的 internalTransform 方法执行具体的变换工作。在此方法中首先初始化控制流

```
/*BodyTransformer是Soot提供的抽象类，完成对方法的变换。其子类提
供具体的变换实现。子类CFGViewer实现方法体控制流图的可视化。*/
1: public class CFGViewer extends BodyTransformer {...
/* 实现具体变换的方法，生成用DOT语言描述的控制流图。*/
2: protected void internalTransform(Body b,
        String phaseName, Map options) {
//初始化图的属性
3: Initialize(options);...
//构造被分析方法体的控制流图，将其以DOT语言的形式输出
4: print_cfg(body);...
5: }
6: public static void main(String[] args) {
7: CFGViewer viewer = new CFGViewer();//创建变换对象
/*Transform类型构造函数的参数。实现这个变换的阶段是
jtp.printcfg，变换的对象是viewer。*/
8: Transform printTransform =
     new Transform(phaseFullName,viewer);
//*可以实现这个阶段必须设置的属性。如控制流图的类型、被分析类的路径等
9: printTransform.setDeclaredOptions("enabled"+...);
//设置这些属性的缺省值
10: printTransform.setDefaultOptions(...+
       "graph-type:BriefUnitGraph"+...);
//将构造的变换加入到jtp中，然后提交给Pack管理器
11: PackManager.v().getPack("jtp").add(printTransform);
//分析用户选择的来自命令行选项的控制流图属性，将其提交给Soot
12: args = viewer.parse_options(args);...
13: sootMain.main(args); //启动Soot的运行流程
14: }...
15:}
```

图 5 CFGViewer 程序片段

图的可视化属性，之后将根据图的属性和被分析的方法体的结构产生一个 dot 图形文件。

最终要确保这个变换在适当的时机被触发。Soot 在它执行的不同阶段运行许多 Pack，这些 Pack 在 PackManager 类的构造器中构造。在这里用户自定义的变换 printTransform 被加入到 jtp 中，然后提供给 Pack 管理器 PackManager 进行管理。为了实现扩展，Soot 提供了一个 Main 类。这个类可以调度 Soot 的运行流程，并调用包含在 PackManager 中的各个 Pack 执行。

下面举例说明控制流图生成过程。假设要分析的类文件是 ExceptionTest.class，存储在 D:\test 目录下，变换对象设定为 JimpleBody，要生成类型为 ExceptionUnitGraph 的控制流图，执行下面的命令：

```
$java soot. tools. CFGViewer - f J--soot-class-path D:\test--graph=ExceptionUnit Graph ExceptionTest
```

该命令为类的每一个方法生成一个控制流图形文件，图形文件的扩展名是.dot。

DOT 语言表示的图形文件产生后，还需要一个绘图程序将其绘制出来。dot 是一个图形绘制程序，它通过一定的算法产生图的布局，其输入为 DOT 语言描述的图形文件，通过命令行参数可以选择生成不同种格式的图形文件。如要生成 PostScript 格式的图形文件，可以执行下面的命令：

```
$dot - Tps graph.dot - o graph.ps
```

-Tps 指出输出文件的格式是 PostScript；graph.dot 是 DOT 语言描述的图形文件，它是输入文件；graph.ps 是输出文件，可通过 PostScript 阅读器显示。其它格式的输出文件都有相应的阅读器。比如，SVG 格式的文件就可以通过 IE 浏览器浏览。

4　结语

Soot 编译优化框架是一个开源软件，不仅能够利用它所提供的功能实现程序优化和分析的目标，而且还可以在其基础上进行二次开发。在我们的研究过程中，对 Soot 的控制流可视化程序进行了分析和开发，在控制流图中添加了更为丰富的图形表现元素，如中文标注和突出显示异常等。通过分析开发体会到 DOT 语言灵活而丰富，通过它可以获得控制流分析结果的形象表达，有助于对分析结果的理解。

参考文献

1 Vallee-Rai R. Soot:A Java Bytecode Optimization Framework.October,2000.http://www.sable.mcgill.ca/publications/thesis/Reference/

2 Einarsson A, BRICS JDN. A Survivor's Guide to Java Program Analysis with Soot. 07/17/2008. http://www.brics.dk/SootGuide/sootsurvivorsguide.pdf.

3 Muchnick SS,赵克佳,沈志宇,译.高级编译器设计与实现.北京:机械工业出版社, 2005:123 - 156.

4 Gansner E, Koutsofios E, North S. Drawing graphs with dot. 2006-01. http://www.grap hviz.org/ Documentation/dotguide.pdf