第35卷 第15期 计 算 机 工 程 2009年8月
Vol.35 No.15 Computer Engineering August 2009
·软件技术与数据库· 文章编号：1000—3428(2009)15—0088—03 文献标识码：A 中图分类号：TP301.6

在静态编译器中实现Java异常机制的算法

曹志伟，杨克峤，王 伟，周 寻，杨 珉
(复旦大学计算机科学与技术学院，上海 201203)

摘 要：将Java程序静态编译成可执行程序是使用Java虚拟机动态编译/解释执行Java程序的另一种运行Java程序的方式。针对Java异常机制的特点和静态编译的需求，在介绍Java异常处理逻辑的基础上，提出一种在静态编译器中实现Java异常机制的算法，结合Open64开源编译器，给出该算法的具体步骤以及实现方式，以SPECjvm98为测试集，验证该算法的有效性。

关键词：Java语言；异常；静态编译；Open64编译器

Algorithm for Implementing Java Exception Mechanism in Static Compiler

CAO Zhi-wei, YANG Ke-qiao, WANG Wei, ZHOU Xun, YANG Min
(School of Computer Science and Technology, Fudan University, Shanghai 201203)

【Abstract】Compiling a Java program into native executable file statically is an alternative way of using Java VM(Virtual Machine) to run a Java program dynamically. This paper introduces an algorithm to implement Java exception mechanism in a Java static compiler, and presents the implementation of this algorithm based on the Java Open64 open source compiler. The effectiveness of this algorithm is proved by the test with SPECjvm98 testsuite.

【Key words】Java; exception; static compilation; Open64 compiler

1 Java异常的处理逻辑

Java程序的编译运行有2种方式：在Java虚拟机(Java Virtual Machine, JVM)上动态编译执行其class文件；静态编译成可执行程序。Java异常机制中的finally机制是Java异常的最大特点，并大大增加了Java异常处理的复杂性。根据Java语言规范[1]的要求，Java异常的处理逻辑如图1[2]所示，其中，

(1)try block raises no exception; no finally block
(2)try block raises no exception; finally block specified
(3)try block raises exception; catch block does not handle exception; no finally block
(4)try block raises exception; catch block does not handle exception; finally block specified
(5)try block raises exception; catch block handles exception
(6)catch block handles exception; finally block specified
(7)catch block handles exception; no finally block
(8)catch block handles exception, raises another exception; finally block specified
(9)catch block handles exception; raises another exception; no finally block
(10)finally block raises no exception
(11)finally block raises exception
(12)finally block propagates previous exception, or raises another exception
(13)nested block propagates exception; catch block handles exception
(14)nested block propagates exception; catch block does not handle exception; finally block specified
(15)nested block propagates exception; catch block does not handle exception; no finally block

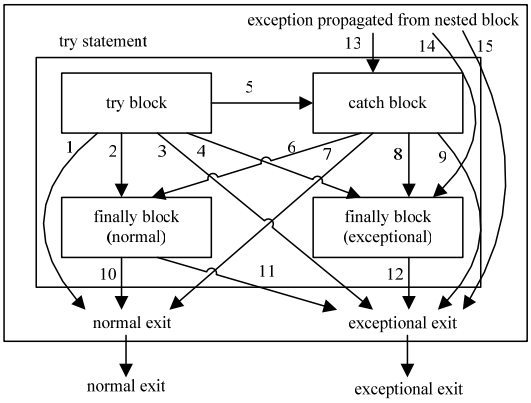

图1 Java异常处理的流程

2 静态编译器中实现Java异常的数据结构

由图1可见，正确实现Java异常机制中所规定的逻辑关系的关键就是要明确try与try之间的从属关系。为此设计了一个类，用来分析和记录try的属性信息。

```
class Try_Monitor{
std::vector<Try_Info> try_vector;
std::stack<Try_State> state_stack;  …};
```

该类中的try_vector用于记录一个函数中所有try的属性信息，其中每个Try_Info记录一个try的属性信息。在Try_Info中，本文主要关注其中的parent_try域。state_stack记录了当前分析和转换抽象语法树的状态。Try_State的定义为

作者简介：曹志伟(1983－)，男，硕士研究生，主研方向：先进编译技术；杨克峤，博士研究生；王 伟、周 寻，硕士研究生；杨 珉，助教、博士

收稿日期：2008-11-30 **E-mail：**062021097@fudan.edu.cn

```
struct Try_State{
int index;
Try_Branch branch;};
```

其中，index 是 try_vector 的索引，用于指定一个 try。Try_Branch 的定义如下:

```
enum Try_Branch{
try_branch,
catch_branch,
finally_branch};
```

其中的每一个枚举值用于指示所处的 try 的分支。

3 静态编译器中实现Java异常的算法

在编译器中实现异常机制需要解决以下 4 个问题：(1)识别可能抛出异常的语句。(2)分析 try 的从属关系。(3)确定可能抛出异常的语句的异常着陆点(异常发生时程序跳转到的异常处理代码的起始位置)。(4)构造异常处理代码。通常，编译器在分析和转换抽象语法树的过程中解决以上 4 个问题，并记录这些信息，为之后代码产生阶段做准备。在 Open64 中，分析和转换抽象语法树的任务由 wgen 子程序完成。该程序以 gcc 产生的抽象语法树 AST[3]为输入，以 Open64 的中间表示语言 WHIRL(Winning Hierarchy Intermediate Representation Language)[4]为输出。

(1)识别可能抛出异常的语句

当 wgen 处理到可能抛出异常的语句时，它创建一个 REGION，将该语句添加到其中，并在 REGION 的附加信息中记录对应的异常着陆点信息。笔者修改了 wgen，使其在处理 Java 程序时，对除了 CALL 语句之外的其他可能抛出异常的语句，如 ILOAD(间接载入，可能抛出 NullPointerException 异常)、DIV 和 REM(除法运算和求模运算，可能抛出 ArithmeticException 异常)执行同样的操作。

(2)分析 try 的从属关系

当 wgen 遇到一个 try 的 AST 时的算法(算法 1)如下，wgen 可以在转换该 try 的同时，分析和设定它的 parent_try 信息：

```
begin build_try_block
  BuildingTry = the current try
  EnclosingTry = current_state.index
  EnclosingTryState = current_state.state
  add BuildingTry into try_vector
  if EnclosingTry = -1
set BuildingTry have no parent
  else   if EnclosingTryState = try_branch
       set the parent of BuildingTry EnclosingTry
    else if EnclosingTryState = (catch_branch or finally_branch)
       set the parent of BuildingTry the parent of EnclosingTry
end if
  end if
  push Try_State(BuildingTry, try_branch)
  for each statement in BuildingTry block
    if it is a may-throw-exception statement
      record_landpad(this statement)
    end if
  end for
  pop Try_State
  generate the after_EH_label of BuildingTry
  if there is a finally block following BuildingTry
    push Try_State(BuildingTry, finally_branch)
    generate the finally block
    pop Try_State
  end if
end
```

(3)确定可能抛出异常的语句的异常着陆点

为确定各个可能发生异常的语句的异常着陆点，设计了算法 2：

```
begin record_landpad
  if current_state.state is try_branch
     set the landpad the EH_block_lable of current_state.try
  end if
  if current_state.state is catch_branch
    if there is a finally block following current_state.try
      set the landpad the finally_block_label of the finally block
    else if the parent_try of current_state.try exists
      set the landpad the EH_block_lable of the parent_try of
      current_state.try
    end if
  end if
  if try state is finally_branch
    if the parent_try of current_state.try exists
      set the landpad the EH_block_lable of the parent_try of
current_state.try
    end if
  end if
end
```

(4)构造异常处理代码

wgen 在转换完一个函数的函数体之后，构造该函数中各个 try block 的异常处理代码。根据 Java 语言规范的要求，对 wgen 进行了相应的修改，使其产生能够正确处理 Java 异常的代码。算法 3 的代码如下：

```
begin build_EH_block
  for each try in try_vector
    generate the EH_block_label of this try
    generate the exception type judgment and jump statements
    if there is a finally block being part of this try
      push Try_State(this try, finally_branch)
      generate the finally block
      pop Try_State
    end if
    generate Unwind_Resume function callsite
    for each catch block of this try
      generate the catch_block_label of this catch block
      push Try_State(this try, catch_branch)
      generate this catch block
      pop Try_State
      generate jump statement to the after_EH_label of this try
    end for
    if there is a finally block being part of this try
      generate the finally_block_label of this finally block
      push Try_State(this try, finally_branch)
      generate the finally block
      pop Try_State
      if the parent_try of this try exists
        generate jump statement to the EH_block_lable of the
parent_try of this try
      else        generate Unwind_Resume function callsite
      end if
    end if
```

```
    end for
end
```

下面以一个有嵌套 try block 的 Java 函数为例，对它应用算法 1 和算法 3 后，其对应的 WHIRL 结构如图 2 所示，其中，各个 block 都可能包含产生异常的语句，而这些异常所对应的异常着陆点却并不相同。

```
F(){
 try     //try outside
 {              code block a
        try     //try inside
        {…}
        catch(ExceptionType A)  //catch A
        {…}
catch(ExceptionType B)    //catch B
        {…}
        finally                 //finally inside
        {…}
        code block b }
 catch(ExceptionType C)  //catch C
 {…}
 finally                 //finally outside
 {…}}
```

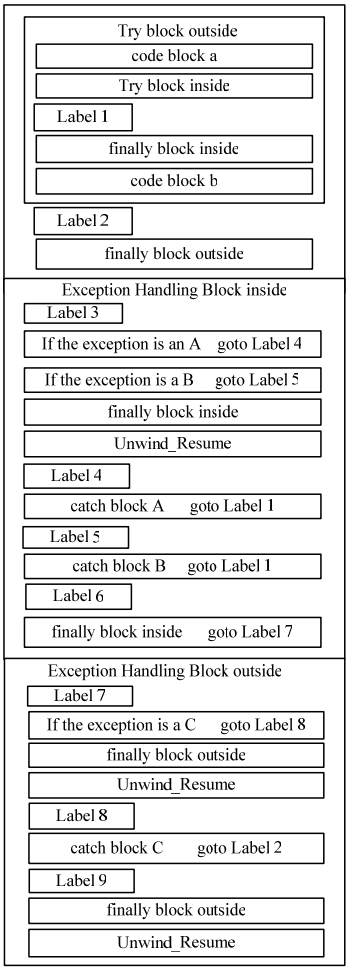

图 2　示例函数 *F* 的转换结果

通过算法 2，可以使 try block inside 的异常着陆点设置为 Label 3，catch block A 和 catch block B 的异常着陆点设置为 Label 6，finally block inside, code block a 和 code block b 的异常着陆点设置为 Label7，catch block C 的异常着陆点设置为 Label 9，对于其他地方产生的异常不做设置。这样，就可以使该函数的异常处理逻辑符合 Java 语言规范的要求。

4　测试及结果

SPECjvm98[5]是当前被普遍采用的一种衡量 Java 虚拟机有效性和性能的测试集，其中包括 7 个测试程序。这 7 个测试程序中包含的异常数量如表 1 所示。在扩展 Open64 的过程中采用了本文的算法支持静态编译下的 Java 异常机制。SPECjvm98 的测试结果表明，该异常处理算法是有效的。

表 1　SPECjvm98 中各个测试程序中的异常数量

测试程序名称	异常数量
_227_mtrt	226
_202_jess	1 324
_201_compress	100
_209_db	19
_222_mpegaudio	451
_228_jack	242 318
_213_javac	23 849

5　结束语

笔者参与了开源编译器 open64 的扩展工作，使其能够支持对 Java 程序的静态编译。如何支持 Java 异常机制成为开发过程中的一个关键问题。本文讨论了一种在静态编译器中实现 Java 异常机制的算法。通过该算法，Java 的静态编译器可以正确实现 Java 异常的复杂逻辑，保证包含 Java 异常的静态程序能够正确执行。

下一步将通过适当的修改使该算法可用于实现 C++的异常机制，成为一种语言无关的编译器异常实现算法。另外，将尝试设计一种能够处理异常机制的控制流分析算法，并使用其分析结果对异常处理代码进行适当的优化。

参考文献

[1] James G, Bill J, Guy S, et al. The Java Language Specification[M]. 3rd ed. [S. l.]: Addison Wesley, 2006.

[2] Saurabh S, Jean M H. Analysis and Testing of Programs with Exception-handling Constructs[J]. IEEE Transactions on Software Engineering, 2000, 26(9): 849-871.

[3] Richard M. Stallman and the GCC Developer Community[Z]. GNU Compiler Collection Internals, 2008.

[4] WHIRL Intermediate Language Specification[EB/OL]. (2000-05-10). http://open64.sourceforge.net.whirl.pdf.

[5] The Standard Performance Evaluation Corporation. JVM Client98 (SPECjvm98)[EB/OL]. [2008-03-11]. http://www.spec.org/osg/jvm98/.

编辑　张　帆