基于 Soot 的 JAVA 字节码优化及性能分析

沈卫强　朱金平

(暨南大学　计算机科学系　广州　510632)

摘要: Java 语言至今已成为最受欢迎的编程语言之一,由于其平台无关性、执行的安全性以及垃圾收集等特性而得到广泛应用,受到众多的 IT 企业及开发者的支持。然而,与诸如 C/C++ 这类语言比起来,Java 语言的运行性能在很多情况下还有待提高,优化 Java 应用性能的课题就成为当前业界迫切需要解决的问题和研究的热点。本文从优化 Java 字节码的角度切入,介绍一款在其上进行优化和变换的框架,并展示相关应用的例子。

关键词: Java,字节码,优化,Soot

JAVA By Tecode Optimization and Performance Analysis Based on Soot

SHEN Weiqiang,ZHU Jinping

(Jinan university computer science,guangzhou,510632,China)

Abstract: Java has become one of the most popular programming languages and well supported by numerous IT enterprises and developers, due to its many features like platform - independence, execution security and garbage collection. However, as to C and C++, the execution speed of Java needed much further improvement and the issue of optimizing Java applications has been evolving a hotspot in the industry. The paper discusses Java bytecode optimization technique along with an optimization and transformation framework called Soot, and presents some examples used on optimization issues.

Keyword: Java,Bytecode,Optimization,Soot

1　引言

Java[1] 具有简单、解释执行、动态、安全、健壮、独立于平台、可移植等特点,它通过丰富的类库支持包括分布式计算、多线程、网络等各种应用的开发,并支持通过第三方类库进行进一步的扩展,因而被各种应用领域广泛采用。

Java 之所以能具有这些特点尤其是平台无关性跟它的实现机制有关。Java 编译程序首先将源程序编译成与平台无关的字节码,并将结果保存在 .class 文件中。随后,Java 虚拟机的解释程序逐条解释执行字节码[2]。由上述可知,在执行 Java 程序前混合有编译过程和解释过程。由于 Java 程序不是直接在物理机而是在虚拟机上运行,加之解释类型语言一般比纯编译型语言效率要低,所以用其开发的应用的运行性能较之以 C/C++ 或其他编译型语言开发的应用的性能存在不同程度的差距。

考虑到 Java 在诸多方面的优势,它在性能上的缺陷显得有些美中不足,这也促使业界迫切地想通过优化 Java 运用来提升其性能。笔者通过研究当前 Java 性能优化的已有的工作成果,探讨了许多关键技术。Java 的性能主要取决于几点[3]:①应用本身的设计和编码;②虚拟机执行字节码的速度;③运行库的运行速度;④操作系统和硬件的速度。

其中①和③取决于良好的架构设计和高质量的代码,当前业内已总结有许多关于这方面的经验和技巧

本文于 2012-02-01 收到。

(4) 显然已超出一般优化技术的讨论范围。本文将着重讨论②中的字节码优化,以及一种字节码优化框架 – Soot。Soot 是由 McGill 大学的 Sable 研究组开发的一个工具,为字节码的分析、优化、反编译、可视化和注解提供一个可扩展的框架。

2 Soot 框等概述

Soot 从两方面来优化 Java 字节码的。

(1) 直接优化字节码: 有些字节码指令会比其他一些所用的代价更高。例如,将一个本地变量载入到堆栈所需的消耗不大,但虚方法调用、接口调用、对象分配以及捕捉异常等操作的消耗很大。像复制传播这种传统的类 C 语言优化技术效果不佳,因为它们优化的对象不是消耗大的字节码。为了在这一层次达到更好的优化效果,需考虑更为有效的优化技术,如方法内联及静态虚方法调用解析等,因为这类方法会直接减少使用这些高消耗的字节码指令。

(2) 注释字节码: Java 的运行安全机制保证所有的潜在的非法内存访问均要在运行前作安全检查。在某些情况下,在编译时就可确定某些检查是不必要的。例如,许多数组边界检查能够完全确定是不必要的[4]。但尽管确定了一些数组访问的安全性之后,仍不能直接从字节码中消除边界检查,因为它们就隐含在字节码指令当中。但若能够利用某些注释机制就安全问题与虚拟机进行沟通的话,虚拟机就能因为省去这些冗余操作而提高运行速度了。

从以上层面着手优化的好处是显见的: ①类文件是许多编译器的目标文件,其对于所有的 JVM 和预编译器(Ahead – of – time compiler)来说都是可移植的。因而仅需通过优化字节码就相当于同时为众多源语言/目标虚拟机组合作了优化。②类文件优化能够静态地进行而且仅需一次。进行静态的类文件优化能够减轻即时编译器的负担,从而允许其在运行时做更多别的优化。

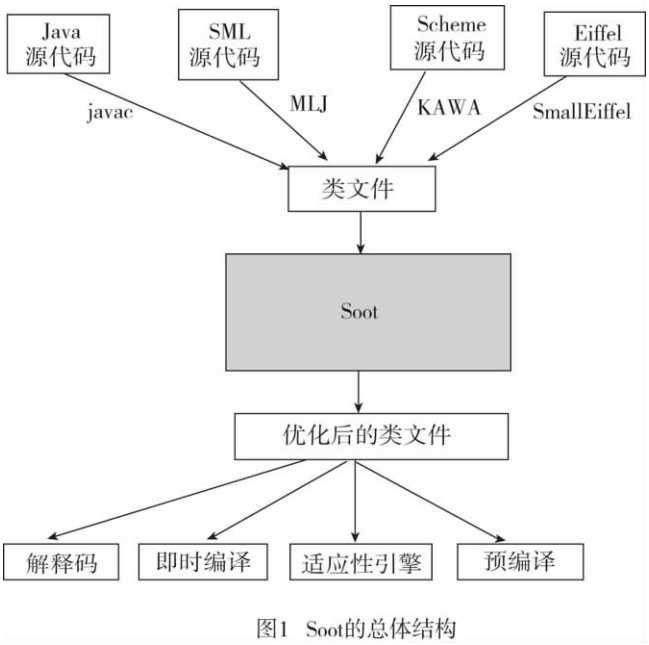

图1 Soot的总体结构

Soot 的总体结构如图 1 所示。

图 1 中经过 Soot 框架处理的类文件的来源多样,如 javac 编译器。Soot 处理这些类文件后输出的是优化过的类文件。Soot 框架提供一套中间表示(IR)以及 Java API,其分析和变换建立在内部的中间表示之上,它们可作局部(仅对某个方法体)或全局(对整个程序)优化。这些 IR 分别是[5]: ①Baf 是基于栈的字节码简洁表示; ②Jimple 是一种类型化的三地址无栈表示; ③Grimple 是 Jimple 的聚合版本,适于反编译和代码审查; ④Shimple 是 Jimple 的 SSA(Static Single Assignment)变种; ⑤Dava 是适用于反编译的结构化表示。其中 Jimple 是 Soot 的核心 IR,许多优化和转换都是基于它的。这些 IR 之间均可相互转换。

3 Soot 的工作机制

3.1 Soot 的主要数据结构

Soot 构建的主要数据结构有: 类 Scene、SootMethod、SootField 和 Body。类 Scene 表示整个分析变换的环境; SootClass 表示载入到 Soot 或由其创建的单个类; SootMethod 表示类中的单个方法; SootField 表示类中的单个域。类 Body 是一个抽象类,表示一个方法体,它的类结构如图 2 所示。

Body 类包含域 localChain，trapChain 和 unitChain，各代表局部变量、Trap 和 Unit 链。Trap 是描述异常捕获的接口; Unit 是描述代码片段的接口，子接口 Inst 和 Stmt 分别表示 Baf IR 的上的指令和其它 IR 上语句。许多类由 Unit 的抽象子类 AbstractUnit 派生并实现 Inst 或 Stmt 接口，表示具体的指令或语句类。

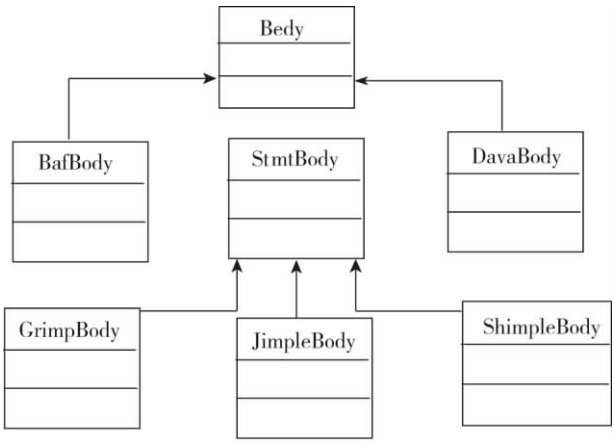

图2　Body抽象类的类结构图

3.2　Soot 的阶段

Soot 的运行分为几个阶段，每个阶段叫作 Pack。为便于扩展，Soot 采用以 Pack 为中心的优化框架。Pack 是一套优化变换的包装类，其 opts 域收集各变换对象(为 Transform 实例) ，方法 apply 调用 internalApply 执行所包装的各种变换。Pack 有 BodyPack 和 Scene-Pack 两个子类，继承前者的类包装方法体内的变换，继承后者的类则包装全局变换。各 Pack 实现类需重写 internalApply 方法。

类 Transform 维护二元组 <阶段名，变换器 > ，以 Transformer 为基类的变换器是实现从字节码到 IR 以及 IR 间的转化，或基于 IR 的分析、优化、反编译、标注等的类。和 Pack 类似，它有 BodyTransformer 和 SceneTransformer 两个子类。变换由类中的 transform 方法调用 internalTransform 完成，各 Transformer 实现类需重写 internalTransform 方法。不论是全局还是局部变换，一般都是对方法体中的语句逐条分析变换的，故操作的核心是 Unit 实例。

各种 Pack 由类 PackManger 来管理，其 init 方法来负责创建各 Pack 的实例对象，并为之添加变换器。表格 1 列举了 Soot 中部分 Pack [5] 。

4　Soot 的运用

4.1　指向分析 (Points – to Analysis)

指向分析的目的很简单，即分析出程序中任何两个指针或引用变量是否可能指向同一个主存的位置。该分析在优化编译器中是非常有用的。在某两个变量可以确定无关的情况下，编译器常常可以进行更加激进的优化。同时，在自动向量化，自动并行化的方法中，也需要指向分析的信息 [6] 。

表格 1　Soot 中的部分 Pack

Pack 名	所属 Pack 类	说明
jb	JimpleBodyPack (BodyPack 的子类)	创建 Jimple 体
jj	JavaToJimpleBodyPack (BodyPack 的子类)	实现 Java 到 Jimple 的转换
cg	CallGraphPack (派生自 ScenePack)	调用图生成，指针分析、类层次分析(CHA)
wstp	ScenePack	全局 Shimple 变换包
wsop	ScenePack	全局 Shimple 优化包
wjtp	ScenePack	全局 Jimple 变换包
wjop	ScenePack	全局 Jimple 优化包
wjap	ScenePack	全局 Jimple 注释包
jtp	BodyPack	Jimple 变换包
jop	BodyPack	Jimple 优化包
jap	BodyPack	Jimple 注释包
tag	BodyPack	代码属性 tag 聚集包

Soot 提供一套模块化的指向分析的工具集 – SPARK(Soot Points – to Analysis Research Kit) ,其功能有[7]: ①测试影响程序准确度和效率的各种指针分析的参数; ②实现各种类型的指针分析; ③进行各种各样的客户端分析。Spark 通过构建 PAG(Pointer Assignment Graph) 图,运用传播算法计算图中每个结点的 P2Set(Points – to Set) ,分析程序中的指针在运行时所指向的对象。

Soot 项目团队还开发了 Eclipse 的 Soot 插件,以便于用户能够更高效地使用 Soot。以下是一个指向分析的例子,指向分析例子的主要代码如下:

```
StaticVoid foo( ) {
  a1: P = new o( ) ;
   q = p;
  a2: r = new o( ) ;
   p • f = r;
   t = bar( q) ;
  }
static o bar( o s) {
return s • f;
} |
```

设定基于子集等参数后,运行 Spark,产生的 PAG 示意图如图 3 所示。

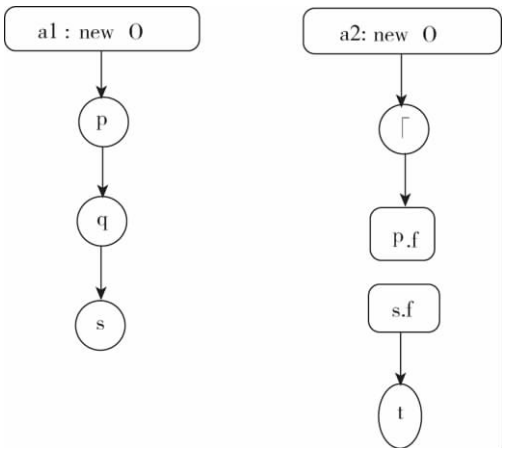

图 3 运行 Spark 后产生 PAG

图 3 中有向边指向的节点表明了赋值和指向关系。图中左部显示 p、q 和 s 三个变量均指向新产生的类 O 的一个实例,而 r 和 p 的域 f 指向类 O 的另一个实例。

4.2 字节码优化(Bytecode Optimization)

为了更直观地理解 Soot,下面我们举一个 Java 字节码优化的实际例子。

我们利用 Soot 优化一个解决八皇后问题的 Java 类,假定该类名为 NQueen. java。该类理论上能解决 N 皇后问题,N 为自然数且大于 0。其源代码[8] 如下:

```
//n 皇后问题
import java. io. * ;
/*
在 n 行 n 列的国际象棋棋盘上,最多可布 n 个皇后。
若两个皇后位于同一行、同一列、同一对角线上,
则称为它们为互相攻击。
n 皇后问题是指找到这 n 个皇后的互不攻击的布局。
n 行 n 列的棋盘上,主次对角线各有 2n – 1 条。
利用行号 i 和列号 j 计算
主对角线编号 k 的方法是 k = n + i – j – 1;
计算次对角线编号 k 的方法是 k = i + j
* /
//"n 个皇后问题" 之类定义
public class NQueen {
int n; // 皇后问题的大小
int col [ ]; // 数组,各列上有无皇后( 0,1)
int md [ ]; // 数组,各主对角线有无皇后( 0,1)
int sd [ ]; // 数组,各次对角线有无皇后( 0,1)
int q [ ]; // 数组,第 i 行上皇后在第几列( 0,n – 1)
int Q; // 已布皇后数,计数
int r; // n 皇后问题的解的组数
// 构造函数 n 皇后问题的初始化
public NQueen( int m) {
n = m;
Q = 0;
r = 0;
col = new int [n];
md = new int [2 * n – 1]; // 初始化 0
sd = new int [2 * n – 1];
q = new int [n];
}
// 函数: 打印棋盘
public void showBoard( ) {
// int i, j;
// System. out. println( " – – – – 第" + String. valueOf( r + 1)
+ "组解 – – – –");
// for ( i = 0; i < n; i ++) {
// for ( j = 0; j < n; j ++)
// if ( q [i] = = j)
// System. out. print( "1 ");
// else
// System. out. print( "0 ");
// System. out. println( ) ;
// }
r ++; // 解的组数
}
// 求解 n 皇后问题
/*
```

```
* 此函数试图在 n* n 的棋盘的第 i 行上放一个皇后，若找到可以放的位置，就递归调用自身试图在 i + 1 行 放另一个皇后，若第 i 行是最后一行，则打印棋盘。
* /
public void resolve( int i) {
int j;
// 在第 i 行给定后检查棋盘上的每一列
for ( j = 0; j < n; j ++) {
// 如果在第 i 行的第 j 列可以布放皇后
if ( col [j] = = 0 && md [n + i - j - 1] = = 0 && sd [i + j] = = 0) {
Q ++;
q [i] = j; // 布放皇后，第 i 行皇后在第几列
// 标记新布皇后的攻击范围
col [j] = md [n + i - j - 1] = sd [i + j] = 1;
// 如果已经布了 n 个皇后( 得到了一组解) ，
// 把棋盘( 解) 打印出来。
if ( Q = = n)
showBoard( ) ;
// 否则，递归。在第 i 行第 j 列布放皇后的前提下，
// 试探下一行( i + 1 行) 在哪一列布皇后?
else if ( i < n - 1)
resolve( i + 1) ;
else
resolve( 0) ; //约定起始行可以任选
// 移除在第 i 行的第 j 列新布的皇后，
// 并消除所标记的攻击范围，为回溯作准备。
Q - -;
q [i] = 0;
col [j] = md [n + i - j - 1] = sd [i + j] = 0;
// 试探在第 i 行的第 j + 1 列新布皇后的方案
}
} // 下一列，j 循环
// 对于给定的行，列扫描完毕后，从这里回溯。
}
// 输出解的个数
public void HowMany( ) {
System. out. println( " n" + n + " 皇后问题共有解" + r + "组。") ;
}
// 主方法 main( )
public static void main( String [] args) {
//程序执行开始时间
long start = System. currentTimeMillis( ) ;
// 定义一个 N 皇后问题
int N = Integer. parseInt( args [0]) ;
NQueen Q1 = new NQueen( N) ; // N 表示解 N 皇后问题
//可以在任意一行布放，参数在 0 到 n - 1 之间任选
Q1. resolve( 0) ;
Q1. HowMany( ) ;  // 输出解的个数
//程序执行结束时间
long end = System. currentTimeMillis( ) ;
System. out. println( " 解" + N + " 皇后问题，共耗时: " + ( end - start) + " 毫秒。nn") ;
}
} // 类 Queen 定义结束
```

将 NQueen. java 类放置于目录 1: F: workspacesootsrc，通过 javac 命令编译在目录 2: F: workspacesootin 生成 Java 字节码 NQueen. class。接下来我们将利用 Soot 对 NQueen. class 进行优化，并对比优化前后程序的执行时效。

以命令行模式进入 NQueen. class 的所在目录 2，运行命令 java soot. Main - W NQueen，此命令执行完毕后将会在目录 3: F: workspacesootinsootOutput 生成 Soot 优化后的 Java 字节码 NQueen. class。

接下来，我们通过 Java 命令分别运行位于目录 2 的优化前的 NQueen. class 和位于目录 3 的优化后的 NQueen. class 的运行时效。因目前微机的运行速度较快，解 8 皇后问题用时基本都在几毫秒内，为了能更清晰地对比优化前后程序的执行效率，我们让程序解 15 皇后问题。优化前后，程序运行时效对比情况如下:

运行次数	优化前(毫秒)	优化后(毫秒)	效率提高
第 1 次	23343	22157	5. 08%
第 2 次	23346	22175	5. 02%
第 3 次	23345	22170	5. 03%

从上面的运行结果数据我们可以看出，NQueen. class 在经过 Soot 优化后，性能提升了 5% 左右。

4. 3　注释代码

Soot 的注释框是用来支持运用 Java 类文件属性进行的 Java 程序的优化。它给相关代码段打上标签，虚拟机就能根据这些标签进行一些优化，如不必要的边界检查等。该框架有以下四个主要概念:

(1) Hosts 任何能容纳和管理标签的对象，SootClass、SootField、SootMethod、Body、Unit 等都实现了这个接口。

(2) Tags 任何能给 Hosts 作标签的对象。它将 name - value 对绑定到 hosts。

(3) Attributes 顾名思义,就是所有能作为属性的对象,它是 Tag 的扩展概念。

(4) TagAggregators BodyTransformer 的一种,将某些类型的 Tag 收集起来后生成新属性输出到类文件中。

有了一套标签机制后,就可应需求定制注释转换器了。软件行业中有一种较为流行的说法即: 90% 的程序运行时间消耗在 10% 的代码上。优化不可能面面俱到,一定是抓重点,这样才能以最小的投入产生最大的收益。有时候在代码量较大的情况下要甄别这些繁忙的代码本身也不是件简单的任务。利用 Soot 的注释框架可以较高效地找出那些繁忙的表达式 (Very Busy Expression) 并将之通过某种方式注释标明,这样我们就可以更容易地确定哪些代码最值得去优化了。

目前有许多基于 Soot 的项目,如加州伯克利分校的 Ptolemy、斯坦福大学的 chord、IBM 的 Canvas 等; 还有一些基于 Soot 的大学研究生课程,如 McGill 大学的"优化编译器"和华盛顿大学的"程序设计语言的实现"等。

5 结束语

利用 Soot 框架,笔者对一些 Java 应用的部分关键代码作了一些粗略优化后,它们获得了 5% -13% 不等的性能提升。Java 应用优化通过本文开篇所述的各种优化进行综合优化,大多数都能取得理想的性能提升,满足实际需求。接下来笔者会更加深入和细致地对这些应用进行分析和优化,以期达到更好的优化效果。

Soot 主要作为一个字节码优化框架,提供了许多分析、优化和注解技术,但逃逸分析以及现在非常重要的程序并发性等功能还有待实现。

参 考 文 献

[1] Gosling James, Joy Bill, Steele Guy, et al. The Java Language Specification Third Edition [M]. 2005.

[2] Lindholm Tim, Bracha Gilad, Buckley Alex, et al. The Java Virtual Machine Specification Third Edition [M]. 2009

[3] J Shirazi. Java Performance Tuning [M]. O'Reilly. 2002.

[4] Gupta Rajiv. Optimizing array bound checks using flow analysis [J]. ACM, 1993, 2(1-4): 135-150

[5] Einarsson Arni and Nielsen Janus Dam. A Survivor's Guide to Java Program Analysis with Soot [M]. University of Aarhus, Denmark. 2008.

[6] Rayside Derek. Points-To Analysis [EB/OL]. http://groups.csail.mit.edu/pag/6.883/lectures/points-to.pdf 2005.

[7] Lhoták Ond ej. Spark: A flexible points-to analysis framework for Java [D]. Montreal: School of Computer Science, McGill University, 2003.

[8] lp503609. 八皇后问题. 2010-5-9. http://zhidao.baidu.com/question/151874290.html.

作者简介

沈卫强,男,(1984-),(汉族),广东韶关人,工程硕士研究生,主要研究方向: 计算机网络、数据库系统的应用与开发。