# 基于对象引用关系的Java程序内存行为分析方法

李文杰[1],姜淑娟[1],钱俊彦[2,3],王兴亚[1],鞠小林[1,4]
(1. 中国矿业大学计算机科学与技术学院,江苏徐州 221116;2. 桂林电子科技大学计算机科学与工程学院,广西桂林 541004;
3. 广西可信软件重点实验室,广西桂林 541004;4. 南通大学计算机科学与技术学院,江苏南通 226019)

**摘 要:** 本文提出一种基于对象引用关系的Java程序内存行为分析方法.与传统的通过内存消耗的大小来确定程序中数据结构的重要性并分析相关内存行为的方法不同,本文方法同时考虑内存消耗和内存支配两个因素来确定一个数据结构在程序内存行为中的重要性,通过研究数据结构之间在内存使用上的支配关系和对数据结构进行引用分析,得到程序中重要的内存行为.实验结果表明该方法能有效地分析程序的内存行为,且对比其它方法能提供更加准确的内存行为分析结果.

**关键词:** 程序理解; 内存行为; 引用分析; 度量策略
**中图分类号:** TP311 **文献标识码:** A **文章编号:** 0372-2112 (2015)07-1336-08
**电子学报URL:** http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2015.07.013

# Memory Behavior Profiling of Java Program Based on Object Reference Relations

LI Wen-jie[1], JIANG Shu-juan[1], QIAN Jun-yan[2,3], WANG Xing-ya[1], JU Xiao-lin[1,4]
(1. *School of Computer Science and Technology, China University of Mining and Technology, Xuzhou, Jiangsu* 221116, *China*;
2. *School of Computer Science and Engineering, Guilin University of Electronic Technology, Guilin, Guangxi* 541004, *China*;
3. *Guangxi Key Laboratory of Trusted Software, Guilin, Guangxi* 541004, *China*;
4. *School of Computer Science and Technology, Nantong University, Nantong, Jiangsu* 226019, *China*)

**Abstract:** An approach was proposed to analyze the memory behavior of Java programs based on object reference relations. Unlike traditional methods that just consider the factor of memory consumption to find out program's important data structures and analyze their memory behavior, our approach considers both memory consumption and memory domination to decide whether a data structure is important, and get the important memory behaviors via analyzes the memory dominance relationship between data structures and references analysis. Experimental results show that the proposed approach is effective, and can get more precise analysis results than other methods.

**Key words:** program comprehension; memory behavior; reference analysis; measurement strategy

## 1 引言

随着信息技术的发展,如何保证和提高软件质量成为软件界最为关心的问题之一.现代软件系统常常因为大量内存被低效地使用而出现内存泄漏、内存膨胀等问题,对系统的性能和可扩展性构成严重影响,甚至导致系统崩溃[1,2].有效的内存行为分析有助于程序员深入了解程序的内存使用情况,进而辅助发现潜在的内存问题.

目前已有方法主要是对程序的堆内存进行统计分析和引用分析来研究程序的内存使用情况[3~7].Rayside等人[3]将堆内存中的对象以对象引用图的方式进行表示,通过所有权关系分析程序的内存使用情况.Kelley等人[4]通过对程序的堆转储文件进行引用分析,将堆内存以有向图的方式进行全局性的表示.Mitchell等人[5]依据对象所有权关系分析堆内存中消耗内存较多的数

收稿日期:2014-09-23;修回日期:2014-11-13;责任编辑:孙瑶
基金项目:国家自然科学基金(No.61340037);江苏省333高层次人才培养工程;江苏省高校自然科学研究(No.12KJB520014);南通市应用研究计划(No.BK2014055);广西高等学校高水平创新团队及卓越学者计划

据结构,而后进行引用分析.Reiss 等人[7]通过构建程序的堆模型,将程序的内存使用情况进行可视化显示.但是,这些方法存在两个问题:①Java 程序涉及大量对象的创建和使用,直接依据对象引用关系进行引用分析需要大量的资源开销;②已有方法从内存消耗的角度分析出内存消耗较多的数据结构,提供与该数据结构相关的全部内存行为信息,但其无法确定造成程序内存消耗的主要内存行为.在内存分析过程中,如何减少资源开销并准确判别其中的主要内存行为是目前亟需解决的问题.

针对存在的问题,本文提出一种基于对象引用关系的内存行为分析方法.首先,针对程序中存在大量重复的对象引用关系[8],对相同的引用关系进行聚合分析构建对象组引用图(Object-Group Reference Graph,O-GRG)来减小分析过程中的数据量,降低资源开销;其次,分析 OGRG 中各个节点的在内存使用方面的属性,并依据引用关系分析节点间在内存使用上的支配关系,得到节点间联系的强弱;再次,提出一个新颖的度量策略,除了从内存消耗的角度进行分析之外,同时考虑数据结构所支配的内存大小,度量各个节点对程序内存消耗的贡献大小;最后,对贡献较高的节点进行引用分析,通过设置阈值,得出导致程序大量内存消耗的内存行为分析报告.实验结果表明,本文方法能有效地分析程序的内存行为,且对比目前已有方法能提供更加准确的分析结果.

## 2　Java 程序内存行为分析

### 2.1　对象组引用图(OGRG)的构建

#### 2.1.1　对象引用图

**定义 1**　对象引用图(ORG).ORG 是一个有向图,图中节点代表对象,有向边代表对象之间的引用关系.图中的根节点为程序堆内存中的根对象(root object),根节点对所有节点都是直接或间接可达的.

**定义 2**　对象类型.即对象的数据类型,例如类 A 的对象 $a_1,\cdots a_n$,它们对应的对象类型即为类 A.

我们从堆中的根对象开始遍历获取程序执行过程中的对象引用信息,根据对象引用关系得到那些同根对象直接或间接可达的对象.在 ORG 中,有向边的源头是一个对象,指向的是被该对象所引用的对象.例如,设 < obj1,obj2 > 表示对象 obj1 与 obj2 之间的引用关系,其中 obj2 被 obj1 引用,本文方法将在 ORG 中记录一条从对象节点 obj1 指向对象节点 obj2 的有向边.每一条对象引用关系的存储结构如图 1 所示.

```
1.class Org{
2.    long tag_from;          //引用对象的标签
3.    String type_from;       //引用对象的类型
4.    double size_from;       //引用对象的内存大小
5.    long tag_to;            //被引用对象的标记
6.    String type_to;         //被引用对象的类型
7.    double size_to;         //被引用对象的内存大小
8.}
```

图 1　存储 ORG 信息的数据结构

#### 2.1.2　对象组引用图

程序执行过程中涉及大量对象,导致生成的 ORG 非常庞大.我们通过两个步骤对 ORG 中的对象进行分组聚合处理,将 ORG 简化为 OGRG.

**步骤 1**　聚合具有相同对象类型的对象节点

在 Java 程序中,相同对象类型的一组对象往往具有相似的操作,使得 ORG 中包含大量相同的引用关系[8].我们将相同对象类型的对象进行分组,使用单一节点进行表示,将相同的引用关系采用单一的引用边进行表示,消除 ORG 中的冗余引用序列.

通过对象分组聚合构建 OGRG 时,我们统计各个对象组中对象的数量,并保存对象组之间的引用关系,具体存储结构如图 2 所示.算法 1 描述了具体分析过程,将 ORG 作为输入,聚合分析结果 OGRG 作为输出.第 2 行初始化 OGRG.第 3 ~ 11 行对 ORG 中的引用边进行分析.例如对于引用边 < $o_1$, $o_2$ > ,设 $o_1$, $o_2$分别对应的对象类型为 $O_1$, $O_2$,该引用边将对应一条从 $O_1$指向 $O_2$的引用序列.若该序列在 OGRG 中已经存在,则将引用计数 count 值加 1(行 5 ~ 7).若该序列在集合中不存在,则在 OGRG 中添加该序列,并设置序列的引用计数 count 为 1(行 10).节点信息包括对象对应的对象类型和包含的对象数目,边信息为两节点之间引用次数的计数.

```
1.class Ogrg{
2.    int count;              //引用计数
3.    String node_from;       //引用节点名
4.    String node_to;         //被引用节点名
5.}
```

图 2　存储 OGRG 信息的数据结构

**算法 1**　Obj_grouping

```
输入:对象引用图 ORG
输出:对象组引用图  OGRG
1  Set < Ogrg > Obj_grouping (Set < Org > ORG){
2     Set < Ogrg > OGRG = new Set < Ogrg > ();
3     for each object reference relation Org in ORG do
4        for each reference edge Ogrg in OGRG do
```

```
5          if Ogrg.node_from = = Org.type_from and
6             Ogrg.node_to = = Org.type_to then
7              Ogrg.count + + ;
8              break;
9          else
10        OGRG.add(new Ogrg(1,Org.type_from,Org.type_to));
11     end for
12  end for
13  return OGRG;
14  }
```

**步骤 2** 聚合循环引用结构中的对象节点

程序执行过程中若存在类间循环引用,将导致 ORG 中出现循环引用结构,增加理解内存行为的难度.我们使用 Tarjan 等人[9]强连通结构检测算法的一个变体对 ORG 中的循环结构进行检测,并使用单一节点进行表示.对于代表各个循环结构的节点:①节点的内存是其所包含对象内存大小的总和;②节点包含的对象将作为一个整体;③节点中各个对象的内存大小取平均值.我们使用与步骤 1 类似的方法对包含相同内容的循环引用结构节点进行分组聚合,同样使用单一节点进行表示,该节点存储其所包含循环引用结构的所有信息并命名为 Cycle_$N$($N$ 为节点序号).

通过以上两个步骤,程序的对象引用信息得到极大简化,生成一个体积更小、概括性更强的有向无环图.

## 2.2 内存支配分析

**定义 3** 内存支配.在 OGRG 中,设节点 $x$ 引用节点 $y$(即 $x$ 为父节点,$y$ 为子节点),如果 $y$ 中的对象全部被 $x$ 引用,则称 $x$ 对 $y$ 完全支配(控制).如果节点 $y$ 中的对象只有部分被 $x$ 引用,则称 $x$ 对 $y$ 部分支配(控制).

设 $A$ 和 $B$ 是 OGRG 中的两个节点,节点 $B$ 引用节点 $A$(即 $A$ 为子节点,$B$ 为父节点).在内存支配分析过程中,我们计算四个属性值:

(1)节点消耗内存的大小

节点消耗的内存是节点包含的各个对象内存大小的总和.例如节点 $A$ 内存消耗(Memory Consumption,MC)的计算公式为:

$$\mathrm{MC}_A = \sum_{i=1}^{n} \mathrm{memory}(a_i) \tag{1}$$

其中,$a_1,\cdots,a_n$ 表示节点 $A$ 中包含的所有对象,memory($a_i$)表示每一个对象所占用的内存大小.

(2)父节点对子节点内存支配的比例

每一条引用关系代表子节点中的一个对象被父节点中的一个对象引用.我们对节点中的各个对象的内存大小取平均值,则父节点对子节点内存支配的比例为父节点对子节点的引用次数与子节点中对象数目的比值.例如节点 $B$ 对节点 $A$ 内存支配比例(Dominance Ratio,DR)的计算公式为:

$$\mathrm{DR}_A^B = R_A^B / N_A \tag{2}$$

其中,$N_A$ 表示节点 $A$ 中对象的数量,$R_A^B$ 为节点 $B$ 与节点 $A$ 之间对象引用的次数.

(3)节点支配的内存大小

一个节点所支配的内存大小为其直接和间接支配内存的总和.节点 $B$ 内存支配(Memory Domination,MD)大小的计算公式为:

$$\mathrm{MD}_B = \sum_{i=1}^{n}((\mathrm{MD}_i + \mathrm{MC}_i) \times (R_i^B / N_i)) \tag{3}$$

其中,$(R_i^B/N_i)$为节点 $B$ 对其子节点 $i$ 的支配比例,$(\mathrm{MD}_i + \mathrm{MC}_i)$为子节点 $i$ 内存消耗和内存支配二者的和.MD 的计算过程如算法 2 所示,OGRG 和图中的根节点作为输入,每个节点支配的内存大小的计算结果作为输出.行 1 定义了一个全局数据集合 MD_result,用于存储计算结果,行 5 ~ 12 通过迭代分析,计算每一个节点所支配的内存大小.在 2.1.2 节步骤 2 中已经对循环结构进行了聚合处理,因此在内存支配计算过程中,不需要考虑循环结构的问题.

**算法 2** MD_computeing

```
输入:对象组引用图 OGRG
      OGRG 中的根节点 N
输出:各个节点支配的内存空间  MD_result
1  Map < String,double >  MD_result = new HashMap < String,double > ;
2              //定义一个数据集合 MD_result,存储计算结果
3  double MD_computing ( Set < Ogrg >  OGRG,String N) {
4  double MD = 0;
5  for each reference edge Ogrg in OGRG do
6     if Ogrg.node_from = N then
7     MD = MD + (the MC of Ogrg.type_to + MD_computing(OGRG,
8        Ogre.node_to)) * (Ogrg.count/Ogrg.type_to.sum);
9        //Ogrg.count 为引用计数
10       //Ogrg.type_to.sum 为 N 的子节点 Ogrg.type_to 包含的对
11       象数量
12    MD_result.put(N,MD);
13  end for
14  return MD;
15 }
```

(4)子节点对父节点内存支配的贡献比例

子节点对父节点内存支配的贡献比例即父节点支配该子节点的内存大小与父节点所支配的总的内存大小的比值.例如,子节点 $A$ 对父节点 $B$ 内存支配的贡献比例(Contribution Ratio,CR)的计算公式为:

$$\mathrm{CR}_A^B = ((\mathrm{MD}_A + \mathrm{MC}_A) \times (R_A^B / N_A))/\mathrm{MD}_B \tag{4}$$

其中,$((\mathrm{MD}_A + \mathrm{MC}_A) \times (R_A^B/N_A))$为节点 $B$ 对其子节点

$A$ 的内存支配大小,$MD_B$ 为节点 $B$ 支配的总内存大小.

以上四个属性值将在 2.3 节和 2.4 节中使用.

### 2.3 度量节点贡献大小

我们提出一个度量策略计算 OGRG 中各个节点对程序内存消耗的贡献大小.该度量策略同时考虑内存消耗和内存支配两个因素.

**内存消耗(MC)** 程序出现内存问题时,我们往往考虑对消耗内存较多的数据结构进行分析,研究其包含的对象是否有必要进行创建和使用.对于内存消耗较多的节点,其 MC 值越大,对程序内存消耗的贡献越大.节点的 MC 值使用式(1)计算.

**内存支配(MD)** 支配内存较大的节点对大量的对象保持引用,使这些对象不被垃圾收集器回收.即这些节点是导致程序大量内存消耗的原因所在.因此,在内存行为分析时,节点的内存支配也是需要考虑的因素,且节点的 MD 值越大,其对程序内存消耗的贡献越高.节点的 MD 值使用式(3)计算.

**内存消耗贡献(Memory Consumption Contribution, MCC)** 大小的度量:

内存问题常常发生在内存消耗较多的代码区域,而节点自身内存消耗较大是导致程序内存消耗显著最为直接的原因,因此就影响大小而言,节点的内存消耗比内存支配对程序内存消耗的影响要更大.MCC 的计算公式为:

$$MCC = MC \times (MD + MC) \tag{5}$$

在式(5)中,MCC 定义为 MD 与 MC 的和值同 MC 值的乘积,体现出 MC 对 MCC 的影响更大.同时,对 MCC 的定义还具有以下的特征:

(1)如果节点的 MC 足够小,即使其 MD 值较大,也将使得计算得到的 MCC 较小.该属性可以对内存支配较大但内存消耗较小的节点进行过滤.

(2)如果节点的 MD 足够小,其 MCC 的大小将由 MC 值决定.

(3)如果节点的 MC 值较大,其 MCC 值也将较大.该属性能突出那些内存消耗较大的节点.

(4)如果节点的 MD 较大,其 MCC 值的大小较大程度上将由其 MC 值决定.对于具有较大 MD 值,但 MC 值较小的节点而言,其 MCC 的计算结果也将较小.

根据式(5)计算出的 MCC 值对节点进行排序,排序靠前的节点往往不仅消耗了大量的内存,同时也支配控制了大量的内存,对程序内存消耗的贡献较大.

### 2.4 引用分析

程序的对象引用模式与执行模式非常相似[8],因此可以通过引用分析来研究程序的内存行为.由于我们关注的是那些造成大量内存消耗的内存行为,本文方法只对 MCC 值较大的节点进行引用分析,通过设置阈值,得到导致大量内存消耗的内存行为.引用分析包括前向引用分析和后向引用分析两个方面.

**前向引用分析** 分析 MCC 值较大的节点被哪些父节点引用,是这些父节点使该节点消耗和支配的内存不被垃圾收集器回收.算法 3 描述了分析的具体过程.该算法将阈值,待分析节点和 OGRG 作为输入,分析出的引用路径集合作为输出.第 2 行初始化一个 Set 集合,用来存储引用路径信息.第 3 ~ 8 行是迭代分析,对于节点 $N$ 的各个父节点,只有当父节点的 DR(使用式(2)计算得到)达到或超过阈值 $t_1$ 时,才继续对该父节点进行引用分析(行 7).阈值的设置可以对不重要的引用路径进行过滤,减少分析过程中的资源开销.算法 3 最终将得到造成节点 MCC 值较大的引用路径.

**算法 3** UW_analysis

```
输入:阈值 t1
     待分析的节点 N
     对象组引用图 OGRG
输出:引用路径集合 ref_path
1  Set<Ogrg> Ref_UW(double t1,String N,Set<Ogrg> OGRG){
2    Set<Ogrg> ref_path = new Set<Ogrg>();
3    for each reference edge Ogrg in OGRG do
4      if Ogrg.node_to = N then
5        if the DR of Ogrg.node_from >= t1 then
6          ref_path.add(Ogrg);
7          Ref_UW(t1,Ogrg.node_from,OGRG);
8    end for
9  return ref_path
10 }
```

**后向引用分析** 分析 MCC 值较大的节点所支配的内存主要来自哪些子节点,这些子节点占据大量的内存空间,造成大量的内存消耗.后向引用分析同前向引用分析非常相似.对于分析节点 $N$ 的各个子节点,只有当子节点的 CR(使用式(4)计算得到)达到或超过设定的阈值时,才继续对该子节点进行引用分析.最终得到 MCC 值较大节点的内存支配主要是由于哪些引用路径造成的.

将两类引用分析的结果结合,将构成程序大量内存消耗的主要引用路径分析报告.由于节点对应于程序中具体的数据结构,因此得到的引用路径即为程序的内存行为.

## 3 实验

为了评估本文方法的有效性,我们进行实验分析.实验运行环境为:Intel(R) core(TM)1.10GHz 处理器,内

存大小为 2GB.

### 3.1 实验对象

本文以 DaCapo[10]测试程序集(DaCapo-9.12-bach 版本)中的 11 个程序作为实验对象(如表 1 所示).所选程序涵盖了 Web 服务器、开发工具、数据库等多个方面,都是实际应用领域使用的大中型开源 Java 程序,且在许多研究工作中已经被使用[7,11].因此,选择这些程序进行内存使用分析具有较好的代表性.

**表 1 实验对象**

| 程序 | 功能描述 | 代码行数 |
|---|---|---|
| Avrora | 模拟程序在 AVR 微控制器网格上运行 | 69393 |
| Batik | 基于 Apache Batik 的单元测试 | 186460 |
| Eclipse | Eclipse 工具平台核心模块 | 289641 |
| Fop | 将 XSL-FO 文件转换为一个 PDF 文件 | 102909 |
| Jython | Java 语言实现的 Python 编译器 | 245016 |
| Luindex | 使用 Lucene 进行文本索引 | 36099 |
| Sunflow | 图片渲染系统 | 21970 |
| Tomcat | 对 Tomcat 服务器进行检索和验证 | 161131 |
| Xalan | 将 XML 文档转换为 HTML | 172300 |
| Lusearch | 使用 Lucene 进行文本查找 | 41153 |
| Pmd | Java class 文件分析器 | 60062 |

### 3.2 方法实现

本文方法使用 Java 虚拟机工具接口(Java Virtual Machine Tool Interface,JVMTI)[12]提供的函数接口创建软件代理来进行信息收集.由于一次堆遍历操作的开销比较昂贵,为了减少开销,我们对程序运行时的 GC 事件进行监控,使用采样的方法,每间隔一定数量的 GC 事件进行一次信息的收集.GC 事件的间隔数量作为一个参数对内存行为分析的精度和资源开销进行控制.在获得对象引用信息之后,我们使用前面描述的分析方法进行数据分析.在引用分析中,我们将阈值以 0.05 为间隔进行 0 到 1 之间的不同设置,以此来寻求阈值的最优设定.通过实验研究发现,当阈值设定为 0.1 时,可以较好地获取程序主要的引用路径分支,同时将分析报告的数据量控制在一个合理的水平,达到检测精度和检测报告信息量之间的平衡.

### 3.3 实验分析

实验分析包含两个部分:一是通过实例分析验证方法的有效性;二是将本文方法同已有方法进行对比.

#### 3.3.1 实例分析

我们对测试程序运行时的对象引用信息进行收集.由于篇幅限制,表 2 中我们列出了各个程序内存使用达到峰值时的数据信息.例如,第二行显示了 Avrora 完成第 12 次 GC 事件时的对象引用信息.11 个测试程序中,Fop 和 Jython 在类的数量、堆内存消耗、引用次数和对象数目等方面的数据量都居于前列,因此,本文选取这两个具有代表性的程序进行详细分析.

**Fop.** 对表 2 中 Fop 的对象引用信息进行分析后得出的节点内存消耗贡献排名中,排名最高的节点对应的类为 LineLayoutManager $ Pragraph,用于对大量格式文件被处理之后的数据进行存储.该节点支配的内存(MD)为 8.52MB,占堆内存消耗总量的 50.56%,节点自身内存消耗(MC)为 247.7KB.对该节点进行引用分析,得到该节点的内存消耗主要引用路径如图 3 所示.图 3 中节点为类名,有向边表示节点之间的引用关系,并标记了对象引用的统计次数.由图 3 可以看出,导致该节点排名靠前的主要原因是其自身占用了大量内存并对大量类 KunthGlue、KnuthIllineBox、KnuthPenalty 的对象存在引用操作.

为了验证分析结果的有效性,我们对 Fop 的源代码进行分析.程序中类 LineLayoutManager 负责对格式文件进行布局管理,其引用大量子布局管理器类 LineLayoutManager $ Paragraph 的对象建立多个含有内嵌区域的文本行.LineLayoutManager $ Paragraph 则引用大量类 KnuthInlineBox 的对象创建内嵌框,引用大量类 KnuthGlue 的对象对文本内容进行宽度调节,引用大量类 KnuthPenalty 的对象表示切入点信息,对文本进行布局操作.KunthGlue、KnuthInlineBox、KnuthPenalty 都引用类 LeafPosition 的对象作为基本元素,存储文本位置信息.通过对源程序进行分析,验证了本文方法的分析结果同程序的实际情况是相符的.

**表 2 获取的程序运行时引用信息**

| 程序 | 堆中包含的类的数量 | 堆内存消耗(MB) | 对象引用数目 | 堆中的对象数目 | 信息收集时的 GC 次数 |
|---|---|---|---|---|---|
| **Avrora** | **1140** | **2.08** | **178462** | **36125** | **12** |
| Batik | 2263 | 23.82 | 1266200 | 365519 | 65 |
| Eclipse | 3252 | 22.76 | 1167795 | 386028 | 870 |
| **Fop** | **1951** | **16.85** | **1290658** | **441151** | **102** |
| **Jython** | **3133** | **20.74** | **1738092** | **570856** | **820** |
| Luindex | 894 | 1.18 | 37890 | 11634 | 11 |
| Sunflow | 1121 | 2.35 | 59108 | 20288 | 1400 |
| Tomcat | 2920 | 7.80 | 301136 | 103423 | 500 |
| Xalan | 1374 | 2.35 | 84681 | 26032 | 720 |
| Lusearch | 836 | 1.73 | 71345 | 25972 | 4620 |
| Pmd | 1585 | 10.47 | 783421 | 249758 | 280 |

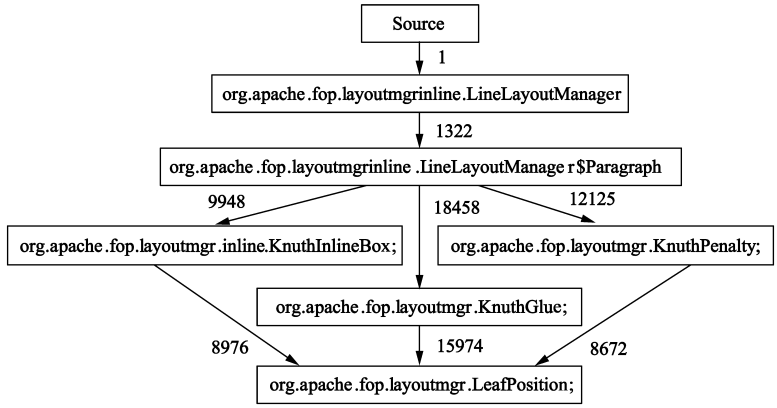



图3　类LineLayoutManager$Paragraph引用分析结果

**Jython**　对表 2 中 Jython 的对象引用信息进行分析得出的节点内存消耗贡献排名中,排名最高的节点对应的类为 PyString,是一个基础元素类.该节点支配的内存大小(MD)为 4.23MB,占堆内存消耗总量的 20.40%,自身内存大小(MC)为 2.61MB.对该节点进行引用分析,得到该节点的内存消耗引用路径如图 4 所示.从图 4 可以看到,导致该节点排名靠前的主要原因是其自身内存消耗较大且对大量类 String 和 PyType 的对象存在引用操作.

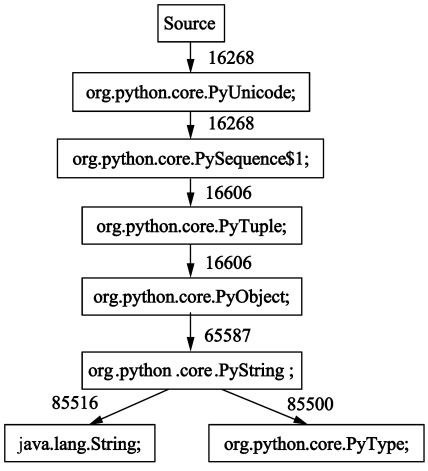



图4　类PyString引用路径分析结果

为了验证分析结果的有效性,我们对 Jython 的源代码进行分析.程序中类 PyString 用于将 Python 类型的字符串转变为符合 Java 规范的字符串,其引用了大量类 String 和 PyType 的对象作为其对象元素.类 PyUnicode 对文件中的对象进行编码操作,其引用大量类 PySequence 的对象来存储对象抽象操作后的序列.PySequence 则引用大量类 PyTuple 的对象对 PyObject 包含的数据进行分析处理.类 PyObject 对 Python 源码中的所有对象进行包装,涉及对大量类 PyString 对象的引用.

通过案例分析,验证了本文方法的分析结果符合程序实际执行情况.本文方法能从程序大量的内存行为中识别出导致大量内存消耗的内存行为,极大地减少程序员在内存行为分析时所需要的工作量.

### 3.3.2　对比分析

目前内存行为分析的相关研究主要是从内存消耗的角度分析程序中消耗内存较多的数据结构,提供与其相关的所有引用信息.本文方法除了考虑内存消耗因素之外,同时还考虑内存支配因素的影响,将二者结合来度量各个数据结构对程序内存消耗的影响.通过分析数据结构之间联系的强弱和对重要数据结构进行引用分析找出主要的内存行为.为了能与当前相关工作进行较好的比较,我们选择 MAT[6] 作为对比对象,出发点是因为 MAT 实现了基于内存消耗因素的统计分析,并包含对数据结构的引用分析,能较全面的涵盖相关工作中涉及的内容.

由于前向引用分析和后向引用分析非常的相似,我们选择前向引用分析对两种方法进行对比.表 3 列出了本文方法与 MAT 对表 2 中 Fop 和 Jython 的引用信息进行分析后的结果.“节点”列显示两种方法根据不同的度量方法得到的排名前 10 位的节点.“分支”列为节点的引用分支数目.“支配比例”列为左侧“分支”列中的分支对该节点内存总的支配比例.在节点排名结果上,两种方法得到的节点范围有部分重合,但在排名上有较大差别.原因是 MAT 仅仅将内存消耗作为影响节点排名的唯一因素,因此在其节点排名中,只要节点内存消耗足够大,该节点就会排名前列.而本文方法在突出内存消耗因素的同时,还考虑内存支配的影响,使得排名靠前的节点不仅消耗大量内存,同时也控制大量内存的消耗,对比目前单一考虑内存消耗因素的分析方法,在节点对程序内存消耗重要性排名结果上更加准确.

在引用分析结果中,MAT 提供与节点相关的所有引用路径信息.例如在对 Fop 的分析结果中,其提供关

于 String 的前向引用路径分支多达 949 条.虽然这些分支涉及节点 100%的内存使用,但对所有的这些分支进行分析,程序员将面临巨大的数据分析量.本文方法依据节点之间支配关系的强弱进行引用分析,得出同节点相关的重要引用路径.例如同样对 String 进行分析,得出的主要引用分支为 4 条,而这 4 条分支控制了 String 超过 90%的内存消耗.与 MAT 相比,本文方法分析结果更加准确,能显著地减轻后续分析过程中的负担.从"支配比例"列可以看出,本文方法得到的引用分支对节点的内存支配比例几乎都在 80%以上,即获得的引用分支是同程序内存消耗最相关的引用操作,对比 MAT,本文方法的分析结果更具有实用性.

**表 3 本文方法与 MAT 分析结果**

| 节点排名 | Fop | | | | | | Jython | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 本文方法 | | | MAT | | | 本文方法 | | | MAT | | |
| | 节点 | 分支 | 支配比例 | 节点 | 分支 | 支配比例 | 节点 | 分支 | 支配比例 | 节点 | 分支 | 支配比例 |
| 1 | LineLayoutManager * | 1 | 99% | char | 43 | 100% | PySequence * | 1 | 99% | char | 37 | 100% |
| 2 | char | 1 | 89% | MinOptMax | 46 | 100% | PyTuple | 1 | 99% | String | 1846 | 100% |
| 3 | AreaInfo | 1 | 99% | LeafPosition | 44 | 100% | char | 1 | 97% | PyString | 208 | 100% |
| 4 | TextLayoutManager | 1 | 82% | **String** | **949** | **100%** | String | 3 | 84% | PySequence * | 19 | 100% |
| 5 | KnuthGlue | 1 | 99% | KnuthGlue | 9 | 100% | PyString | 3 | 92% | HashEntry | 9 | 100% |
| 6 | **String** | **4** | **91%** | AreaInfo | 5 | 100% | PyFunction | 1 | 99% | PyObject | 315 | 100% |
| 7 | CondLength * | 1 | 98% | WordArea | 14 | 100% | PyObject | 1 | 99% | PyInteger | 18 | 100% |
| 8 | Block | 4 | 79% | **KnuthPenalty** | **25** | **100%** | PyTuple | 2 | 99% | PyUnicode | 22 | 100% |
| 9 | KnuthInlineBox | 1 | 99% | SpaceProperty | 12 | 100% | HashEntry | 2 | 85% | PyTuple | 57 | 100% |
| 10 | LeafPosition | 3 | 92% | CondLength * | 14 | 100% | PyUnicode | 1 | 99% | Method | 34 | 100% |

表注:lineLayoutManager * 代表类 LineLayoutManager $ Paragraph;CondLength * 代表 CondLengthProperty;PySequence * 代表类 PySequence $ 1.

本文方法还可以通过阈值的自定义设置对内存行为进行选择性的过滤.例如 MAT 的分析结果中 KnuthPenalty 排名前列,但其不存在对其内存支配比例(DR)达到或超过 0.1 的引用分支,使得其在本文方法的分析结果中被过滤掉.

在时间与空间资源的开销上,本文方法与 MAT 都需收集程序的引用信息.在信息收集过程中,本文方法与 MAT 在时空开销上是相当的,都是从堆内存中读取信息.二者时空开销的差异在于对获得的信息进行引用分析的过程.与 MAT 直接对对象引用图进行引用分析不同,本文方法首先会对信息进行预处理,通过聚合分析构建 OGRG 来大幅减小引用分析过程中的数据量,对比 MAT,本文方法降低了在数据存储和引用分析过程中的资源开销.

## 4 相关工作

Rayside 等人[3]将程序堆内存使用以对象引用图的方式表示,通过对象所有权剖析来分析程序的异常内存使用情况,检测内存问题.Kelley 等人[4]对程序执行过程中的堆转储文件进行分析,将程序的堆内存以有向图的方式进行全局性的表示.这两种方法在分析的过程中,没有对对象引用图进行简化,导致分析的数据量过大,需要大量资源开销.本文方法则通过聚合分析,将对象引用图抽象为对象组引用图,极大的减少了数据量,节约了引用分析过程中的资源开销.

Mitchell 等人[5]依据对象所有权关系分析堆内存中消耗内存较多的数据结构.Reiss 等人[7]依据对象类型将对象引用图进行聚合处理,抽象和构建程序的堆模型,并将分析结果进行可视化显示.但是,这两种方法并没有对造成程序大量内存消耗的具体原因进行分析.与之相比,本文方法在通过内存消耗和内存支配两种因素分析出重要的数据结构之后,继续通过引用分析找出了导致这些数据结构大量内存消耗的具体原因,得到了程序执行过程中的主要引用路径分析报告,结果更加准确有效.

## 5 总结

本文研究了 Java 程序内存行为的分析方法.通过获取对象引用信息建立 OGRG,并在此基础上分析节点间在内存使用上的支配关系,对各个节点进行重要性度量和引用路径分析,得到程序主要内存行为分析结果.本文方法同时考虑内存消耗和内存支配两种因素确定一个数据结构在程序执行过程中的重要性,并通过引用分析和设置阈值得到主要的内存行为.实验结果表明本文方法能有效的分析出引发程序大量内存消耗的内存行为,且对比目前已有的方法,结果更加精确.

## 参考文献


[1] 王雷,李吉,李博洋.缓冲区溢出漏洞精确检测方法研究[J].电子学报,2008,36(11):2200-2204.
Wang Lei, Wang Ji, Li Boyang. Precisely detecting buffer overflow vulnerabilities[J]. Acta Electronica Sinica, 2008, 36(11): 2200 – 2204. (in Chinese)

[2] Yan D, Xu G, Rountev A. Uncovering performance problems in Java applications with reference propagation profiling[A]. Proceedings of the 34th International Conference on Software Engineering[C]. Zürich, Switzerland: IEEE, 2012. 134 – 144.

[3] Rayside D, Mendel L. Object ownership profiling: A technique for finding and fixing memory leaks[A]. Proceedings of the 22nd International Conference on Automated Software Engineering[C]. Atlanta, Georgia, USA: IEEE, 2007. 194 – 203.

[4] Kelley S, Aftandilian E, Gramazio C, et al. Heapviz: Interactive heap visualization for program understanding and debugging [J]. Information Visualization, 2013, 12(2): 163 – 177.

[5] Mitchell N. The Runtime Structure of Object Ownership[M]. Berlin Heidelberg: ECOOP 2006-Object-Oriented Programming, Springer Berlin Heidelberg, 2006. 74 – 98.

[6] Eclipse. MAT[OL]. www.eclipse.org/mat/. 2014-12-13.

[7] Reiss S P. Visualizingthe Java heap[A]. Proceedings of the 32nd International Conference on Software Engineering[C]. Cape Town, South Africa: ICSE, 2010. 251 – 254.

[8] De Pauw W, Sevitsky G. Visualizing Reference Patterns for Solving Memory Leaks in Java[M]. Berlin Heidelberg: ECOOP 1999-Object-Oriented Programming. Springer Berlin Heidelberg, 1999. 116 – 134.

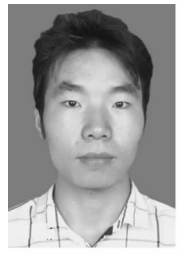
[9] Tarjan R. Depth-first search and linear graph algorithms[J]. SIAM Journal on Computing, 1972, 1(2): 146 – 160.

[10] Blackburn S M, Garner R, Hoffmann C, et al. The DaCapo benchmarks: Java benchmarking development and analysis [A]. Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications[C]. Portland, Oregon, USA: ACM, 2006. 169 – 190.

[11] Xu G, Rountev A. Precise memory leak detection for java software using container profiling[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2013, 22 (3): 17_1 – 17_28.

[12] JVMTI[OL]. http://docs.oracle.com/javase/7/docs/ platform/jvmtijvmti.html. 2014-12-13.


## 作者简介




**李文杰**　男,1990 年 3 月出生于湖北监利.中国矿业大学研究生,主要研究领域为软件测试、程序分析.
E-mail:zs13170017@cumt.edu.cn



**姜淑娟(通信作者)**　女,1966 年 12 月出生于山东莱阳.现为中国矿业大学计算机科学与技术学院教授、博士生导师,CCF 会员.主要研究领域为编译技术、软件工程等.
E-mail:shjjiang@cumt.edu.cn