第41卷 第6A期 2014年6月
计 算 机 科 学 Computer Science
Vol. 41 No. 6A June 2014

利用双向推导检测Java程序中的内存泄漏

张广梅[1] 李景霞[2]
(山东农业大学信息科学与工程学院 泰安 271000)[1] (安徽农业大学信息与计算机学院 合肥 230036)[2]

摘 要 Java应用程序中大量使用动态内存。Java程序运行过程中会自动对不可达的动态内存进行回收,但不能及时地对应用程序中可达但不活跃的动态内存进行回收,从而造成内存泄漏。为有效地检测内存泄漏,提出了采用双向推导进行内存泄漏检测的方法,在推导的过程中利用分离逻辑理论对应用程序中的动态内存进行分析,确定到达程序中每条语句的可达的动态内存和活跃的动态内存,辅助完成动态内存泄漏的检测。

关键词 内存泄漏,分离逻辑,双向推导,霍尔推理规则
中图法分类号 TP311 **文献标识码** A

Detecting the Memory Leak in Java Program by Bi-deduction

ZHANG Guang-mei[1] LI Jing-xia[2]
(School of Information Science and Technology, Shandong Agriculture University, Taian 271000, China)[1]
(School of Information and Computer Science, Anhui Agriculture University, Hefei 230036, China)[2]

Abstract Lots of dynamitic memories are used in Java program. The unreachable dynamitic memories can be collected by the garbage collector. The reachable but inactive dynamitic memories are free which may lead to memory leak. The bi-deduction method to detect memory leak in Java program is provided. It is based on separation logic. By using separation logic and the rules of Hoare logic, the dynamitic memory's state can be achieved and the leaked memory can be detected.

Keywords Memory leak, Separation logic, Bi-deduction, Inference rule of hoare logic

1 引言

在使用c、c++、Java等语言进行系统开发的过程中大量涉及到动态内存的使用。在某些程序语言中,动态内存完全由程序员进行管理,在需要时,程序员通过函数调用或其他方式进行动态内存的申请;每一块被动态分配的内存,必须由程序员进行回收,否则,系统可用的动态内存将会变少,将对系统运行性能产生影响,甚至导致系统的崩溃[1,2]。

Java有其自身的垃圾回收机制。所谓的垃圾,是指不再被访问的动态内存。Java程序运行过程中,通过垃圾收集器自动管理内存的回收。从表面上看Java不存在内存泄漏问题,实际上Java程序中也存在内存泄漏,只不过表现形式与c、c++程序中的内存泄漏不同。

对于使用到动态内存的Java代码,如果要24小时在服务器上运行,即使最小的内存泄漏也会导致JVM耗尽全部可用内存。在很多嵌入式系统中,内存的总量非常有限,微小的内存泄漏也将对系统的运行产生影响。因此,需要严格地对Java程序中的动态内存进行管理,减少内存泄漏的发生。

本文首先介绍Java内存管理机制,进而对导致Java内存泄漏的原因进行讨论;在此基础上,利用分离逻辑理论对Java应用程序中所涉及的动态内存进行分析;最后提出采用双向推导方法对Java内存泄漏进行检测。

2 Java程序中的内存泄漏及其原因分析

2.1 Java程序中动态内存管理机制

Java的内存管理就是对象的分配和释放问题[3]。Java程序中的对象都是在堆(Heap)中分配的(基本类型除外)。对象的创建通常都是采用new操作完成,对象的回收都是由Java虚拟机通过垃圾收集器去完成。垃圾收集器为了能够正确回收动态内存,必须监控每一个对象的运行状态,包括对象的申请、引用、被引用、赋值等。当某个动态内存不再被引用时,垃圾收集器对该内存进行回收。下面借助于图1所示的动态内存使用状态图对程序1的动态内存使用状态进行描述。

程序1

```
class MyTest{
public static void main(String a[]){
  Object a=new Object();
  Object b=new Object();
  a=b;      ……
}
```

图1中用菱形符号表示程序的开始,矩形框表示对象,开始符号和矩形框之间的带箭头的实线表示在程序中声明了该对象;用圆形符号表示动态内存,矩形框和圆形符号之间的带箭头的实线表示一种指向关系,带箭头的虚线表示指向关系

张广梅(1972—),女,博士,副教授,主要研究方向为软件测试,E-mail:lotus@sdau.edu.cn;李景霞(1976—),女,博士,副教授,主要研究方向为模型分析与验证。

的解除。对图中任意的两个节点，如果两个节点之间存在一条带箭头的实线，称两个节点之间有一条边；按任意顺序对图中的顶点进行编号，对于某个节点序列 vivmvn. . vj 而言，如果相邻的两个顶点之间有边，则称 vi 到 vj 是可达的。在动态内存使用状态图中，如果从开始节点到图中的圆形节点之间可达，表示圆形节点所代表的动态内存处于被使用的状态；对于某块内存，不存在从开始节点到该节点的路，表示当前的内存不再被使用。垃圾收集器通过对动态内存的可达性进行分析，及时回收不可达内存，从而将程序员从动态内存管理工作中解脱出来，同时也可以避免由于可用内存减少而造成的系统运行性能降低的危险。

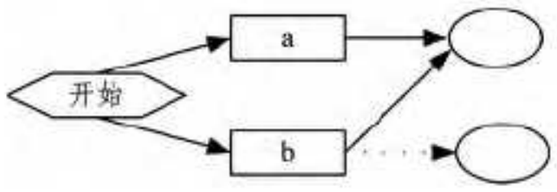

图 1　动态内存使用状态图

2.2　Java 程序中的内存泄漏

为对 Java 程序中的内存泄漏进行解释，首先给出以下两个定义。

定义 1(动态内存可达)　对于 Java 程序中的由 new 操作分配的动态内存，在动态内存使用状态图中，如果在开始节点和代表某动态内存的节点之间存在一条路，则称该动态内存可达；否则，称该动态内存不可达。

定义 2(动态内存的活跃性)　对某个可达的动态内存而言，如果该动态内存在从程序中 p 点开始的路径上被访问(对动态内存进行读写操作)，称该动态内存在 p 处是活跃的，否则称其是非活跃的。

根据动态内存的可达性的定义，动态内存分为两类：可达的动态内存和不可达的动态内存。由于 Java 中的垃圾收集器自动回收不可达的内存，因此，不可达的动态内存会由垃圾收集器进行回收，不会造成内存泄漏。造成内存泄漏的原因在于可达的动态内存。

定义 3(Java 程序中的内存泄漏)　对于 Java 应用程序中的某块动态内存 x，如果在程序中的某点 p，该动态内存可达但不活跃，则称动态内存 x 在 p 处存在泄漏。

2.3　造成 Java 程序中内存泄漏的因素分析

动态内存的访问通过 Java 中定义的对象完成。从图 1 所示的动态内存使用状态图中可以看到，每一个对象对应一块动态内存，一块动态内存可以通过不同的对象访问。一旦动态内存不活跃，该动态内存所对应的对象就成为无用对象。根据上面关于 Java 内存泄漏的定义可知，下面几种情况都可能使得 Java 程序中产生内存泄漏。

2.3.1　单实例模式

单实例模式[4]是系统开发中常用的一种设计模式。这一模式的特点是使得类的一个对象成为系统中的唯一实例。为实现这一目标，需要定义一个静态的变量来保存单实例的引用。一旦单实例类被初始化，在系统结束之前，即使该动态内存在程序中某个位置之后一直处于不活跃的状态，由于该类的对象所占的动态内存一直处于可达状态，垃圾收集器也无法回收该对象所占内存，从而造成内存泄漏。

2.3.2　观察者模式

观察者模式[4]又叫做发布-订阅(Publish/Subscribe)模式。观察者模式定义了一种一对多的依赖模式，让多个观察者同时监听某一个主题对象(被观察者)。为实现对被观察者状态的跟踪，观察者需要将自己注册到被观察对象中，被观察对象将观察者存放在一个容器里。在观察者模式中，被观察者维护着一个观察者的引用，在被观察者释放这个引用之前，即使观察者不再被引用(不活跃)，垃圾收集器也不能对观察者进行回收，从而造成内存泄漏。

2.3.3　静态对象

在 Java 应用程序中，静态对象具有长生命周期。如果长生命的对象持有短生命周期对象的引用就很可能发生内存泄漏，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收，从而造成内存泄漏。

3　Java 程序中的内存泄漏的检测

根据第 2 节中关于内存泄漏的讨论，当某个动态内存处于可达但不活跃的状态时，表示该动态内存处于泄漏状态。为有效地对内存泄漏进行检测，首先对分离逻辑的基本原理进行介绍[5]。

3.1　分离逻辑

分离逻辑是霍尔逻辑的一种扩展[6]。霍尔逻辑是广泛应用的程序验证逻辑系统，用于对命令式语言程序进行推理验证。分离逻辑采用霍尔三元组 $\{P\}C\{Q\}$ 进行程序的推理验证，在霍尔三元组中，P 表示前置条件，Q 表示后置条件，C 表示代码段。霍尔三元组描述一段代码的执行如何改变计算的状态：只要 P 在 C 执行前的状态下成立，则在执行之后 Q 也成立。分离逻辑对霍尔逻辑最重要的扩展在于引入了两个新的分离逻辑连接词：分离合取 $*$ 和分离蕴含 $-\!\!*$。其中，分离合取 $*$ 的定义如下：

令 s 表示栈，h 表示堆，$h_0 \perp h_1$ 表示堆 h_0 和 h_1 不相交，$h_0 \cdot h_1$ 表示堆 h_0 和 h_1 的联合，则：

$$[P * Q]sh \overset{def}{=} \exists\, h_0 h_1, h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h \text{ and } Psh_0 \text{ and } Qsh_1$$

3.2　利用分离逻辑原理进行动态内存的可达性和活跃性分析

根据动态内存的可达性和活跃性的定义，可以在双向推导过程中完成动态内存的可达性和活跃性分析。双向推导分为正向推导和反向推导。正向推导从程序的入口语句开始计算程序执行到某一条语句时的动态内存状态，完成动态内存的可达性分析。反向推导是从程序的出口语句开始计算程序执行该语句时所必须的动态内存，反向推导完成动态内存的活跃性分析。利用正向推导和反向推导的结果进行动态内存的检测。

3.2.1　利用正向推导构造可达动态内存集合

从程序入口开始的推导称为正向推导。可以采用分离逻辑中的空间谓词对动态内存状态进行描述[5-7]。如果程序中存在形如：[引用名=值] 类型的赋值操作，采用分离逻辑的指向断言对该语句执行所造成的动态内存状态的变化进行描述，指向断言的形式如下：

$$e1 \mapsto e2$$

该断言表示一个仅含一个空间的动态内存，$e1$ 是动态内存的地址（变量名），$e2$ 是动态内存中的内容。在 Java 应用程序中，存在形如［变量名 $=$ new 类型］的语句，该语句会在堆上进行动态内存的申请操作，此时不能确定动态内存空间中的内容，可以采用如下的谓词对其进行描述：

$e1 \mapsto -$

利用正向推导，可以获得程序中某条代码执行之后的动态内存分配的状态。如果用 H 表示在执行形如［变量名 $=$ new 类型］的语句之前动态内存的状态（即，已经在堆上完成了动态内存的分配操作），在执行 new 操作之后，根据分离逻辑中分离合取运算符的定义，采用以下的分离逻辑中的分离断言形式对进行 new 操作之后的动态内存状态进行描述：

$H * e1 \mapsto -$

正向推导是从程序入口处开始的分析过程。在程序中第一条语句执行之前，没有进行动态内存的分配操作，采用分离逻辑中的断言 emp 表示此时的动态内存状态。除了上述空间谓词之外，在分离逻辑中还采用谓词 $ls(x,y)$ 和 $ls(O,x,y)$ 描述采用链式存储结构所占的动态内存，其中 x 是起点，y 是终点。$ls(x,y)$ 的定义如下：

$$ls(x,y) \Leftrightarrow (x=y \wedge emp) \vee (\exists x', x \mapsto x' * ls(x',y))$$

谓词 $ls(O,x,y)$ 与 $ls(x,y)$ 相似，同样描述了一段链表，除此之外，用 O 记录了链表中的数据。$ls(O,x,y)$ 的定义如下：

$$ls(O,x,y) \Leftrightarrow (x=y \wedge emp \wedge O=\Phi) \vee (\exists x', o', O', union(o', O')=O \wedge x \mapsto o', x' * ls(O', x', y))$$

正向推导从 emp 状态出发，根据程序语句的特点，推导出程序中各语句执行之后的动态内存状态。对于程序中的每条语句而言，通过正向推导可以得到该语句的可达动态内存集合。

3.2.2 利用反向推导构造活跃动态内存集合

反向推导是从程序出口开始的递推过程，反向推导用来确定到达每条语句的活跃的动态内存集合。反向推导在逐步对程序中的语句进行综合的过程中完成。根据 2.2 节中动态内存活跃性的定义，在最后一条语句执行前活跃的动态内存可以通过对该语句的分析获得，到达其他语句的活跃动态内存需要经过不断对程序中的语句进行综合分析完成。

反向推导过程中，首先对程序中的最后两条语句进行综合，并根据正向推导的结果确定综合之后的两条语句的霍尔三元组 $\{P\}C_{n-1}, C_n\{Q\}$，其中，C_n 是程序中的最后一条语句，C_{n-1} 是倒数第二条语句，$\{P\}$、$\{Q\}$ 是空间谓词，表示在这两条语句执行之前、后的可达动态内存状态，可达动态内存可以通过正向推导获得。对于 C_{n-1}、C_n 语句而言，通过独立分析这两条程序代码可以确定到达该语句的部分活跃动态内存（保证该语句正确执行），采用霍尔三元组 $\{P_1\}C_{n-1}\{Q_1\}$ 和 $\{P_2\}C_n\{Q_2\}$ 分别对其进行描述。根据程序中顺序语句执行的特点，在 C_{n-1} 语句执行之后的动态内存状态和 C_n 语句执行之前的动态内存状态是相同的，也就是说，C_{n-1} 的后置空间谓词和 C_n 的前置空间谓词应该是等价的。反向推导的作用就是用来确定空间谓词 A 和 B 使得 $Q_1 * A \Leftrightarrow P_2 * B$。

A 和 B 的推导借助于分离逻辑中的 Frame 规则完成。Frame 规则的定义如下[3,5,6]：

$$\frac{\{P\}C\{Q\}}{\{P*R\}C\{Q*R\}}$$

该规则的含义为：如果 $\{P\}Q\{R\}$ 成立，则 $\{P*R\}C\{Q*R\}$ 也成立。在该推理规则中，断言 P,Q,R 均为空间谓词，其中，代码段 C 不会对断言 R 中的自由变量赋值。应用该规则，对三元组 $\{P_1\}C_{n-1}\{Q_1\}$ 和 $\{P_2\}C_n\{Q_2\}$，下式成立

$$\frac{\{P_1\}C_{n-1}\{Q_1\}}{\{P_1*A\}C_{n-1}\{Q_1*A\}}$$

$$\frac{\{P_2\}C_{n-1}\{Q_2\}}{\{P_2*B\}C_n\{Q_2*B\}}$$

由于 C_{n-1}, C_n 是两条顺序执行的语句，因此，$Q_1 * A \Leftrightarrow P_2 * B$。应用霍尔逻辑的顺序规则[8]，以下成立：

$$\frac{\{P_1*A\}C_{n-1}\{Q_1*A\}, \{P_2*B\}C_n\{Q_2*B\}}{\{P_1*A\}C_{n-1}, C_n\{Q_2*B\}}$$

根据前面关于后向推导方法的讨论，反向推导是在对语句进行综合的过程中完成的。根据前向推导的结果，得到 $\{P\}C_{n-1}, C_n\{Q\}$ 三元组，该三元组与 $\{P_1*A\}C_{n-1}, C_n\{Q_2*B\}$ 等价，因此可以利用空间谓词 $\{P\}$、$\{Q\}$、$\{P_1\}$ 和 $\{Q_2\}$ 推导出空间谓词 A、B。其中空间谓词 $\{P_2*B\}$ 是在后向推导的过程中所确定的、满足 C_n 语句正确执行动态内存状态即到达 C_n 语句的活跃动态内存集合。

上述过程是对程序中最后两条语句的综合。不断利用该过程对程序中的语句进行综合，可以得到满足每条语句正确执行动态内存状态

3.3 利用双向推导的结果诊断被泄漏的内存

3.2 节对双向推导的方法进行了介绍。正向推导在对程序静态分析的基础上，推导出到达每条语句的可达动态内存集合，为讨论方便，用 HF_i 表示到达第 i 条语句的可达动态内存集合；后向推导在对语句的综合中完成，对程序中第 i 条语句而言，后向推导可以得到到达该语句的活跃的动态内存集合，为讨论方便，用 HB_i 表示该集合。如果 $HF_i = HB_i * R$（R 是一个堆），则表示在第 i 语句执行之前存在内存泄漏，其中 R 为被泄漏的内存，可以在第 i 条语句执行之前对 R 所代表的动态内存进行回收，避免内存泄漏。

3.4 Java 内存泄漏检测的基本过程

根据上文对双向推导方法的介绍及 Java 内存泄漏的定义，要进行 Java 动态内存泄漏的检查，首先要对程序中的语句进行编号，接下来需要针对每条语句构造霍尔三元组，利用前置谓词和后置谓词对该语句执行前后的动态内存状态进行描述，该三元组中的前置谓词和后置谓词只与语句本身有关。在此基础上，首先进行正向推导，继而进行后向推导。最后，根据双向推导的结果进行内存泄漏的诊断。用下面的程序段对内存泄漏检测过程进行说明。

```
1. Class1 x=new Class1();
2. Class2 y=new Class2();
3. x.update(val);
```

针对上述三条代码，分别构造对应的三元组：

$\{emp\}c1\{x \mapsto -\}$，

$\{emp\}c2\{y \mapsto -\}$，

$\{x \mapsto -\}c1\{x \mapsto val\}$

应用正向推导，可以得到如下结果：

$\{emp\}c1\{x \mapsto - \}c2\{x \mapsto - * y \mapsto - \}c3\{x \mapsto val * y \mapsto - \}$

由此,可以得到到达每条语句的动态内存集合,分别为:

1. $\{emp\}$

2. $\{x \mapsto - \}$

3. $\{x \mapsto - * y \mapsto - \}$

接下来进行后向推导。首先进行 c1,c2、c3 的综合。综合之后的三元组如下:$\{x \mapsto - \}C_2, C_3\{x \mapsto val * y \mapsto - \}$。

$c2$、$c3$ 这两条语句所对应的三元组分别为 $\{emp\}c2\{y \mapsto - \}$,$\{x \mapsto - \}c3\{x \mapsto val\}$。应用霍尔逻辑中的顺序规则,下式成立:

$$\frac{\{emp * A\}C_2\{y \mapsto - * A\}, \{x \mapsto - * B\}C_3\{x \mapsto val * B\}}{\{x \mapsto - \}C_2, C_3\{x \mapsto val * y \mapsto - \}}$$

由此得到 $A = x \mapsto -$,$B = y \mapsto -$。因此,保证 C_2 能正确执行的活跃动态内存为 $x \mapsto -$。

继续进行 $c1$、$c2$ 和 $c3$ 的综合,由前向推导可知综合之后的三元组为:$\{emp\}C_1, C_2, C_3\{x \mapsto val * y \mapsto - \}$。应用霍尔逻辑的顺序规则,下式成立:

$$\frac{\{emp * A\}C_1\{x \mapsto - * A\}, \{x \mapsto - * B\}C_2, C_3\{x \mapsto val * B\}}{\{emp\}C_1, C_2, C_3\{x \mapsto val * y \mapsto - \}}$$

由此得到 $A = emp, B = emp$。因此,保证 C_1 能正确执行的活跃动态内存为 emp。

根据后向推导的结果,得到确保每条语句正确执行的活跃动态内存为:

1. $\{emp\}$

2. $\{x \mapsto - \}$

3. $\{x \mapsto - \}$(直接通过语句本身的分析得到)

由双向推导结果可知,在 C_3 语句执行前存在可达不活跃的动态 $y \mapsto -$,根据内存泄漏的定义,该动态内存为被泄漏的动态内存。

4 实例分析

前面对 Java 内存管理方式、Java 程序中内存泄漏的原因及内存泄漏的检测方法进行了讨论。本节将以一个 Java 程序为例,采用上面介绍的方法对程序中的动态内存泄漏进行检查。在下面的例子中,定义了一个被观察者类 DataBag 和两个观察者类 Show 和 Adde。被观察者类中声明了两个私有的数据成员 list 和 observers(均为 ArrayList 类型),其中,list 用于存储数据,observers 维护了一个监听队列(用 ArrayList 实现),将监听该类对象的观察者放入该监听队列中。被观察者 DataBag 类中定义了 add 方法,一旦调用该方法,将向观察者发送消息,观察者一旦收到这一消息,立即给出响应。由于篇幅限制,下面代码中不包含观察者类 Show 和 Adder 以及被观察者类 DataBag 的实现。

```
public class Test {
  public static void main(String[] args)
  {
  1. Integer i1 = new Integer(1);
  2. Integer i2 = new Integer(2);
  3. Integer i3 = new Integer(3);
  4. DataBag bag = new DataBag();//定义了一个被观察者
  5. bag.add(i1);//调用 add 方法,并向被监听者发送消息,此时,bag 对象没有被监听,所以没有响应
  6. Adder adder = new Adder(bag);//将 bag 添加到观察者 adder 的监听队列中
  7. Show printer = new Show(bag);//将 bag 添加到观察者 printer 的监听队列中
  8. bag.add(i2);//调用 add 方法,并向被监听者发送消息,此时,两个观察者将对该消息进行响应
  9. adder = null;//企图清除观察者
  10. printer = null;//但由于已经将观察者加入监听队列中,因此在监听队列中仍然存在该对象的引用,观察者并没有被清除
  11. bag.add(i3);//调用 add 方法,并向被监听者发送消息,此时,两个观察者将对该消息进行响应
  12. bag.pringBag();
  }
}
```

对于上述程序中的第 12 条代码而言,经过正向推导,到达该语句的动态内存为

$\exists bag, list \mapsto x' * ls(x', NULL) * bag.observers \mapsto y' * ls(y', \mathrm{NULL})$

经过后向推导,到达该语句的活跃动态内存为:

$bag.list \mapsto x' * ls(x', \mathrm{NULL})$

因此,$bag.observers \mapsto y' * ls(y', \mathrm{NULL})$ 该内存为被泄漏的内存。

结束语　本文对介绍 Java 内存管理机制、Java 内存泄漏的原因进行讨论,进而提出了采用双向推导的方法,对程序中的可达动态内存和活跃动态内存进行分析,在动态内存状态分析的过程中,利用分离逻辑对双向推导过程中的动态内存进行描述。实例表明,采用双向分析的方法能有效地对程序中的可达动态内存和活跃动态内存进行描述,并能准确定位内存泄漏的位置以及被泄漏的内存,有效地检测动态内存故障。

参 考 文 献

[1] Wikipedia. Memory leak[OL]. http://en.wikipedia.org/wiki/Memory leak,2012-11-09

[2] Clause J, Orso A. Leakpoint: Pinpointing the Causes of Memory Leaks[C]// International Conference on Software Engineering, 2010. Cape Town, ACM New York, 2010: 515-224

[3] Distefano D, Filipovi'c I. Memory Leaks Detection in Java by Bi-Abductive Inference[C]// Fundamental Approaches to Software Engineering, 2010. Paphos, Cyprus, 2010: 278-292

[4] 刘伟. 设计模式[M]. 北京: 清华大学出版社, 2011: 186

[5] Reynolds J C. Separation Logic: A Logic for Shared Mutable Data Structures[C]// 17th IEEE Symposium on Logic in Computer Science, 2002. Los Alamitos, IEEE Computer Society, 2002: 55-74

[6] 黄达明, 曾庆凯. 基于分离逻辑的程序验证技术[J]. 软件学报, 2009, 20(8): 2051-2016

[7] Reynolds J C. An Overview of Separation Logic[M]// Meyer B, Woodcock J. Verified Software: Theories, Tools, Experiments. Berlin Heidelberg, Springer, 2008: 460-469

[8] http://zh.wikipedia.org/wiki/霍尔逻辑