一种基于逃逸分析的对象生命期分析方法*

刘玉宇[1,2]，张昱[1,2]

[1](中国科学技术大学 计算机科学技术系，安徽 合肥 230027)

[2](安徽省计算与通信软件重点实验室，安徽 合肥 230027)

摘 要: 基于区域的分配和显式的内存回收等研究是当前 Java 内存管理优化的主要途径之一，它们希望尽可能准确地获得对象的生命期信息。逃逸分析可以获得对象是否逃逸出某个方法或线程，但对对象生命期分析还不够精确。文中提出一种基于逃逸分析的增量式过程间对象生命期分析方法，利用称为连接图的程序抽象进行图上结点属性的过程内与过程间传播，设计数据流方程计算非全局对象的生命期在哪个方法中结束。实验结果表明该方法精确性较高。

关键词: 对象生命期; 逃逸分析; 连接图; 属性; 传播算法

中图法分类号: TP314 **文献标识码**: A

1 引言

Java 这类面向对象的程序设计语言采用垃圾收集器（Garbage Collector, 简称 GC）在堆（Heap）上动态管理对象的分配与回收。这种内存管理方式不仅使分配和回收具有灵活性，还使得内存管理对程序员透明并保证编程工作的简易性，然而它也为整个系统运行带来了额外的开销，是影响 Java 虚拟机性能的重要因素之一。为降低 GC 管理内存的时空开销，除了进一步改进 GC 本身的算法外，一些研究者还提出两种不同的解决途径：一种是将一部分对象在堆以外的内存空间中分配，如在运行栈上分配[3]~[7]，或者采用基于区域（Region-based）的内存分配方式[1][11]；另一种是由即时编译器(Just-in-time compiler, 简称 JIT)在代码中安插显式的内存回收指令[13]，从而减轻 GC 自动回收的负担。不论是实现对象在堆外空间分配还是显式的内存回收，首先需要解决的问题是确定哪些对象可以采用这类分配回收方式。

为了分析哪些对象可以在堆外分配或者何时能被显式回收，必须引入与对象生命期相关的数据流分析技术，如逃逸分析（Escape Analysis）和指针分析（Points-to Analysis）等，其中逃逸分析是现有研究工作[4]~[12]中主要的一种分析技术。现有的逃逸分析算法可以识别对象能否被分配在堆以外的空间或被显式回收，但是在确定对象的生命期上精度不够，不能很好地指导显式内存回收优化工作。

本文在现有工作的基础上，重点研究：1)如何识别一个对象能否在堆以外的空间分配以及能否被显式回收；2)对于能被显式回收的对象，其生命期会在哪个方法调用中结束，从而为对象显式回收提供依据。

本文的主要贡献如下：

1) 针对 Java 语言，提出一种基于逃逸分析的对象生命期分析方法，它利用称为连接图（Connection Graph）[7]的程序抽象，并结合属性（Property）的过程内与过程间的传播算法，识别对象的逃逸状态并分析非全局对象的生命期。

2) 提出一个基于连接图的、增量式的、过程间的属性传播算法，以获得对象在部分或全部程序中的相

* Supported by the National Natural Science Foundation of China under Grant No. 60673126, 国家自然科学基金; the Foundation of Intel Corporation, Intel 公司研究基金

作者简介: 刘玉宇(1984), 女, 江西吉安人, 硕士研究生, 主要研究领域为程序设计语言理论与实现; 张昱(1972), 女, 安徽肥东人, 博士, 副教授, 主要研究领域为程序设计语言理论与实现等.

Corresponding author: Phn: +86-551-3603804, Fax: +86-551-3607043, E-mail: yuzhang@ustc.edu.cn

关信息。

3) 将对象的生命期结束时期对应到某个方法调用的生命期结束时期，提出一个计算在方法调用内可能结束生命期的对象集合的数据流方程，并结合连接图中结点的属性进一步判断实际结束生命期的对象集合。

文中通过实验与[11]中的分析结果进行对比，实验结果表明本文的算法更为精确。根据是否考虑多个对象之间存在的相互引用关系，本文实现了两类算法，实验结果表明考虑这种引用关系将会使算法更为精确。

2 连接图（Connection Graph）

2.1 基本结构

我们使用文[7]中提出的称作连接图（Connection Graph）的抽象结构作为分析的基础。每个待分析的 Java 方法都可以被抽象出一个连接图结构 *CG*。这里对 *CG* 给出比[7]中更为细化的定义，以便引出更精确的分析算法。

定义 1 (连接图). 设 *M* 是一个待分析的方法，其对应的**连接图**是一个有向图 $CG=(N, E)$, 其中 $N=N_o \cup N_r$ 表示结点的集合，$E=E_p \cup E_f \cup E_d$ 表示弧的集合。对 N_o、N_r、E_p、E_f 和 E_d 解释如下：

- N_o 表示 *M* 内分配点对应的结点集合。一般地，这些分配点是在 *M* 内创建的，但是也存在一些在 *M* 外创建但在 *M* 内使用的分配点。我们把后者对应的结点集合记为 *Phantom*，有 $Phantom \subseteq N_o$。
- $N_r=N_l \cup N_f \cup N_{sf} \cup N_{fa} \cup N_{ret} \cup N_{aa} \cup N_{dst}$ 表示 *M* 中使用的引用结点的集合，这些结点分为以下七类：
  - N_l 表示局部变量集合；
  - N_f 表示非静态域集合；
  - N_{sf} 表示静态域集合；
  - N_{fa} 表示方法 *M* 的形参集合；
  - N_{ret} 表示方法 *M* 的所有返回值集合；
  - N_{aa} 表示方法 *M* 内各调用点处的实参集合；
  - N_{dst} 表示方法 *M* 内各调用点处的返回值接收者集合。
- $E_p \subseteq N_r \times N_o$，表示指向边（Points-to Edges）集合。弧 $<x, y> \in E_p$ 表示引用结点 *x* 可能指向分配点结点 *y*。
- $E_f \subseteq N \times N_f$，表示域边（Field Edges）集合。弧 $<x, y> \in E_f$ 表示域结点 *y* 是结点 *x*（或其引用的对象）的一个域。
- $E_d \subseteq N_r \times N_r$，表示延迟边（Deferred Edges）集合。弧 $<x, y> \in E_d$ 意味着引用结点 *x* 指向由引用结点 *y* 所指向的对象。

2.2 连接图上的基本操作

延迟边用于表示从一个变量到另一个变量上的复写关系。为了简化连接图，这些延迟边可以通过[7]中提出的 bypass 函数来消除。在[7]中，作者根据是否是流敏感（flow-sensitive）分析来决定 bypass 函数在逃逸分析中使用的时机与频率；在本文中，则是为了方便之后的连接图上的属性计算，当创建连接图之后会利用 bypass 函数对其进行修剪。由于本文中建立连接图的算法与[7]中有所区别，因此，对 bypass 的定义也有所不同。

定义 2 (bypass 函数). 给定一个连接图 $CG= (N_o \cup N_r, E_p \cup E_f \cup E_d)$及其中的一个引用结点 $p \in N_r$， 假设 $S=\{x \mid <x, p> \in E_d\}$，$R=\{y \mid <p, y> \in E_p\}$，$T=\{t \mid <p, t> \in E_d\}$，$F=\{f \mid <p, f> \in E_f\}$，执行**函数 bypass(** *p* **)**操作后得到的连接图 $CG' = (N_o \cup N_r, E'_p \cup E'_f \cup E'_d)$。

其中：1) $E'_d = E_d \cup \{<x, t> \mid x \in S, t \in T\} - \{<x, p> \mid <x, p> \in E_d\}$；

2) $E'_p = E_p \cup \{<x, y> \mid x \in S, y \in R\}$；

3) $E'_f = E_f \cup \{<t, f> \mid t \in T, f \in F\} \cup \{<y, f> \mid y \in R, f \in F\} - \{<p, f> \mid <p, f> \in E_f\}$。

2.3 结点的属性（Properties）

我们给连接图 *CG* 中的结点 *p* 赋予一些基本属性，并通过传播算法在 *CG* 上计算属性值。

2.3.1 基本属性

1) 结点 *p* 的逃逸状态属性 *Escape*(*p*)

$\forall p \in N$, *p* 的逃逸状态属性 *Escape* (*p*)$\in$*EscState*，表示 *p* 的逃逸状态可能是 *Escape* (*p*)。这里 *EscState* 为逃逸状态集合，它是一个格：No_Field_Escape $\prec$ Field_Escape $\prec$ Global_Escape。其中，Global_Escape 表示全局逃逸状态，具有该状态的对象是全局共享对象，可被多个线程访问，其生命期跨越整个程序的执行期间；No_Field_Escape 和 Field_Escape 则分别表示两类非全局逃逸状态，具有这两类状态之一的对象是只能被一个线程访问的非全局共享的对象，二者区别在于：No_Field_Escape 状态的对象不被任何对象的域引用，而 Field_Escape 状态的对象被存入到非全局共享对象的某个非静态域中。我们的分析将识别这三类对象，并进一步分析 No_Field_Escape 或 Field_Escape 状态的对象的生命期。

2) 结点 *p* 的分配点属性 *Sites*(*p*)

假设 *AS* 是程序中所有对象分配点的集合。$\forall p \in N$, *p* 的分配点属性 $Sites(p) \subseteq AS \cup \{\text{UNKNOWN}\}$，表示结点 *p* 引用的对象可能的分配点集合。UNKNOWN 表示结点对应的对象分配点是未知的，如在过程内分析时，*Phantom* 中的结点所对应的对象分配点是不明确的。

3) 结点 *p* 的来源属性 *Source*(*p*)

$\forall p \in N_{aa} \cup N_{ret}$, *p* 的分配点属性 $Source(p) \subseteq \{s \mid s \in N_{fa} \cup N_{dst}\}$，表示结点 *p* 引用的对象是由哪些结点传递过来的。*p* 结点表示方法内某调用点的实参或方法的返回值，而 *Source*(*p*)则表示 *p* 通过连接图上的边可达的形参类结点（N_{fa}）或返回值接收者类结点（N_{dst}）的集合。该属性将在过程间的分析算法中用于连接图结点属性的局部更新。

4) 结点 *p* 的类型信息属性 *Type*(*p*)

假设 *TYPE* 是程序中所有对象类型的集合。$\forall p \in N$, *p* 的类型信息属性 *Type*(*p*) $\in$ *TYPE*，表示结点 *p* 引用的对象的类型信息。由于多态的存在，在没有方法调用图的情况下，对方法调用点处实际调用方法的确定需要依靠该属性所提供的信息。

5) 域引用图 *RefsMap*

假设 *FIELD* 是所有对象非静态域的集合。域引用图 $RefsMap \subseteq AS \times FIELD \times AS$, 反映的是程序中对象之间的域引用关系。三元组<*x*, *fd*, *y*>$\in$*RefsMap* 表示分配点 *x* 处对象的域 *fd* 引用了分配点 *y* 处的对象。该图用于辅助对 Field_Escape 状态的对象的生命期分析。

2.3.2 基本属性的扩展

为了计算上的方便，我们对基本属性 *Source* 进行扩展，该属性重定义为：

$\forall p \in N$，则有 $Source(p) \subseteq \{s \mid s \in N_{fa} \cup N_{dst}\}$。

2.3.3 扩展后属性上的合并操作

连接图上的传播算法将会对结点扩展后的属性进行一些合并操作。

定义 3 (属性合并操作 combine). 假设连接图中存在边<*x*,*y*>（或<*y*,*x*>），对 *x* 进行**属性合并操作 combine(*x*, *y*)**规则如图 1 所示。

对于结点 *x* 而言，combine(*x*,*y*)操作是将结点 *x* 中各类属性有选择地进行重新赋值，而结点 *y* 的任何属性值仍保持不变。规则(1-1)表示 *Escape*(*x*)属性在合并之后具有原 *Escape*(*x*)和 *Escape*(*y*)二值中的最保守者；规则(1-2)和规则

$$\frac{st \in EscState,\ st = Escape(y),\ Escape(x) \prec st}{Escape(x) := st} \quad (1\text{-}1)$$

$$\frac{as \subseteq AS, as = Sites(y), Sites(x) = \{\text{UNKNOWN}\}}{Sites(x) := as} \quad (1\text{-}2)$$

$$\frac{as \subseteq AS, as = Sites(y), Sites(x) \neq \{\text{UNKNOWN}\}}{Sites(x) := Sites(x) \cup as} \quad (1\text{-}3)$$

$$\frac{y \in N_r}{Source(x) := Source(x) \cup Source(y)} \quad (1\text{-}4)$$

$$\frac{Type(y) \text{ is sub-type of } Type(x)}{Type(x) := Type(y)} \quad (1\text{-}5)$$

图 1 属性合并操作的规则

(1-3)表示 *Sites*(*x*)属性在合并之后需包含原 *Sites*(*x*)和 *Sites*(*y*)中的任何一个元素(UNKNOWN 除外);规则(1-4)表示 *Source*(*x*)属性在合并之后成为原 *Source*(*x*)和 *Source*(*y*)的并集;规则(1-5)表明如果合并前 *Type*(*y*)是 *Type*(*x*)的子类,那么 *Type*(*x*)需被重新赋值为该子类。除图 1 中的规则外,其它情况下 combine(*x*, *y*)并不改变结点 *x* 的属性值。

3 过程内分析(Intra-procedural Analysis)

3.1 连接图的建立

我们假定程序基于静态单赋值(SSA,Static Single Assignment)格式,SSA 格式可以使得分析具有流敏感性。我们还假定所有形如 *x.y.f* 这类多级的变量表示方式将转化为 *t=x. y* 和 *t.f*。

假设当前被分析的 Java 方法为 *M*。在其相应的连接图 *CG* 的建立过程中,算法将对方法 *M* 的代码进行逐行扫描,并在扫描过程中逐步完成连接图的创建。若 *M* 中存在分支情况,则采用保守的分析处理方式,即考虑每个分支都存在的情况;若 *M* 中出现循环,为简化算法,只对循环内代码处理一次。为了简化叙述,这里只给出以下几类影响逃逸分析的基本语句和基本情况的连接图 *CG* 建立方法:

1) ***M* 中的形参(Formal Parameters)**:假定 *FML* 是 *M* 中形参的集合。$\forall fml_i \in FML$,建立一个结点 $x \in N_{fa}$ 以及一个结点 $y \in Phantom$,并增加一条边$<x,y> \in E_p$。
2) ***v* = new C 或 *v* = new C[ ]** :为变量 *v* 建立一个结点 $x \in N_l$,并为等式右边建立一个结点 $y \in N_o$,增加边$<x, y> \in E_p$。
3) $v_1 = v_2$:为变量 v_1 建立结点 $x \in N_l$。假设 *y* 是变量 v_2 对应在 *CG* 中的结点,则增加边$<x, y> \in E_d$。
4) $v = \mathbf{phi}(v_1,v_2,\ldots,v_n)$:为变量 *v* 建立结点 $x \in N_l$,并增加边$<x, x_i> \in E_d$,其中 $x_i \in N_l$ 表示变量 v_i($1 \leqslant i \leqslant n$)对应的结点。
5) $v_1.f = v_2$(***f* 表示非静态域**):假设 *x* 是变量 v_1 对应在 *CG* 中的结点,为其域 *f* 建立一个结点 $x_f \in N_f$,并增加边$<x, x_f> \in E_f$和边$<x_f, y> \in E_d$,其中 *y* 表示变量 v_2 对应的结点。
6) $v_2 = v_1.f$:与 5)中类似,只是在增加的 E_d 类边时边的方向是相反的,即边$< y, x_f > \in E_d$。
7) $v_1.sf = v_2$(***sf* 表示静态域**):为静态域 *sf* 建立结点 $x \in N_{sf}$,并增加边$<x, y > \in E_d$,其中,*y* 表示 v_2 对应的结点。
8) $v_2 = v_1.sf$:与 7)中类似,只是在增加的 E_d 类边时边的方向是相反的,即边$< y, x> \in E_d$。
9) $v = v_0.\mathbf{foo}(v_1,v_2,\ldots,v_n)$:首先,为 foo 的每个实参 p_i($0 \leqslant i \leqslant n$,$p_0$ 表示隐含参数 *this*)建立结点 $act_i \in N_{aa}$,并增加边$<act_i, y_i> \in E_d$,其中,y_i($0 \leqslant i \leqslant n$)表示 v_i 对应的结点。再者,为该调用点的返回值接收者建立一个结点 $dst \in N_{dst}$ 和一个 *Phantom* 结点 *ph*,为变量 *v* 建立一个结点 $t \in N_l$,并增加边$<t, dst> \in E_d$ 和$<dst, ph> \in E_p$。
10) **return *v***:为 *M* 的返回值建立一个结点 $ret \in N_{ret}$,并增加边$<ret, y> \in N_d$,其中 *y* 表示 *v* 对应的结点。

通过在扫描中对与分析相关的语句进行结点和边的创建,可以完成连接图 *CG* 的创建。然而,这样得到的连接图由于延迟边较多而不够直观,并会使其上的算法变得复杂。因此修剪连接图是我们的下一步工作。

3.2 连接图的修剪

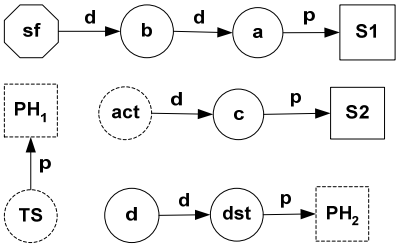
我们利用 2.2 中提到的 bypass 函数实现对连接图的修剪。为了便于之后传播算法的进行,需对连接图进行较为深入的修剪。在经过修剪之后的连接图中,原有的指向边将被保留,而大部分延迟边则会被修剪掉,域边则会被重定向。

我们期望对 $CG = (N_o \cup N_r, E_p \cup E_f \cup E_d)$中的所有引用结点经过多次 bypass 函数处理后得到的连接图 *CG''* 有如下特征: $CG'' = (N_o \cup N_r, E''_p \cup E''_f \cup E''_d)$,其中,1)$E''_d \subseteq E_d$,$E''_d$ 一般为空集,这是由于经过 bypass 处理后,引用结点一般都直接指向 N_o 结点(包括 *Phantom* 结点);当 E''_d 不为空时,该集合中的边$<x, y>$满足 $y \in N_f \cup N_{sf}$; 2)$E''_p \supseteq E_p$,经过 bypass 处理过的部分延迟边会转化为相应的指向边加入到 E''_p 中; 3)E''_f 集合中的边$<x, y>$满足 $x \in N_o$。

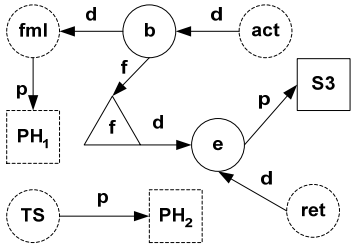
下面，我们举例说明连接图的建立和修剪：在图 2 所示的代码中，类 *A* 中有 *M1* 和 *M2* 两个方法以及一个静态域 *sf*，其中，*M1* 方法虽然形式上没有形参也没有返回值，但有一个隐含的参数即该方法的接收者（*this*），*M2* 方法有两个参数（*this* 和类型为 *B* 的形参）和类型为 *B* 的返回值。类 *B* 中有一个非静态域 *f* 和一个 *foo* 方法；类 *C* 是类 *B* 的子类且重写了 *foo* 方法。图 3 给出了方法 *M1* 和 *M2* 在 bypass 函数修剪前后对应的连接图。

```
class A{                    B M2(B b){                  class B{
  …                           …                           B f;
  static B sf;              S3: B e = new B();            void foo();
  void M1(){                    b.f = e;                  …
    …                           b.foo();                }//end class B
S1: B a = new B();              …                       class C extends B{
    b = a;                      return e;                 …
S2: C c = new C();            }//end M2                   void foo();
    sf = b;                   …                         …
    d = M2(c);              }//end class A              }//end class C
    …
  }//end M1
```

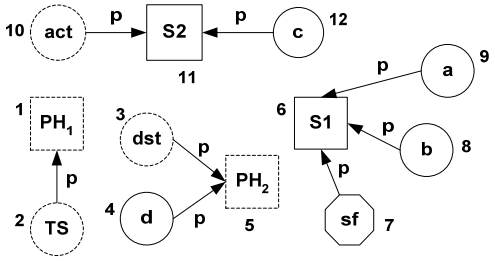
图 2 一个 Java 程序示例



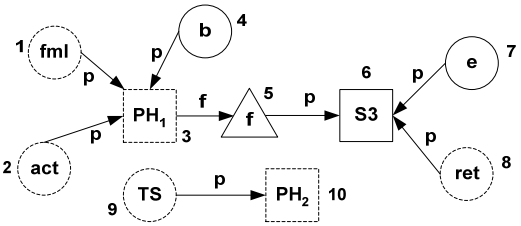
(a) bypass修剪前M1对应的连接图



(b) 修剪前M2对应的连接图



(c) 经过bypass修剪后M1对应的连接图



(d) 经过bypass修剪后M2对应的连接图

图注：

TS : 表示this

PH : 表示Phantom

fml: 表示formal parameter

act: 表示actual parameter

ret: 表示return value

dst: 表示return value receiver

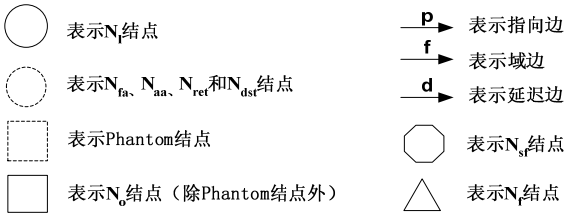


图 3 修剪前后 M1 和 M2 对应的连接图

3.3 连接图上结点属性值的传播

3.3.1 属性初始化

当创建连接图中的结点（假设为 *p*）时，其属性也同时进行初始化。对于不同类型的结点，其属性的初始值会有所不同。

1) 对 *Escape* 属性来说，若 $p \in N_{sf}$，则 *Escape*(*p*):=Global_Escape；若 $p \in N_o$ 且 *p* 为线程类对象，则

Escape(*p*):=Global_Escape；若 $p \in N_f$，则 *Escape*(*p*):=Field_Escape；其它情况下，*Escape*(*p*):=No_Field_Escape。

2) 对 *Sites* 属性而言，只有当 $p \in N_o$ 时，才可以确定其引用的对象的分配点。因此，当 $p \in N_o$ 且 $p \notin Phantom$ 时，*Sites*(*p*):={*as*}，*as*∈*AS* 为 *p* 对应的分配点；当 $p \in Phantom$ 时，*Sites*(*p*):={UNKNOWN}；当 $p \in N_o$ 时，*Sites*(*p*):=∅。

3) 对 *Type* 属性而言，当 $p \in N$ 时，*Type*(*p*):= *type*，*type*∈*TYPE* 为 *p* 所声明的类型。

4) 对 *Source* 属性而言，当 $p \in N_{fa} \cup N_{dst}$，其引用的对象来源于本身。因此，当 $p \in N_{fa} \cup N_{dst}$ 时，*Source*(*p*):={*p*}；当 $p \in N-(N_{fa} \cup N_{dst})$ 时，*Source*(*p*):=∅。

3.3.2 过程内属性传播算法

过程内的属性传播算法主要以 N_o 类结点为中心在经过修剪后的连接图 *CG''*上实现，它分为两步：第一步，$\forall <x, y> \in E''_p \cup E''_d$，执行 combine(*y*, *x*)（规则见图 1）；第二步，$\forall <x, y> \in E''_p \cup E''_d$，执行 combine(*x*, *y*)。此外，$\forall <x, y> \in E''_f$，在域引用图 *RefsMap* 中增加相应的三元组$<as_1, fd, as_2>$，其中，$as_1, as_2 \in AS$ 分别表示结点 *x* 和 *y* 引用的对象的分配点，*fd*∈*FIELD* 表示该域边所对应的非静态域。

我们仍以图 2 中的例子说明过程内的属性传播算法。表 1 和表 2 分别给出了对图 3 中(c)和(d)进行过程内属性传播后各结点的属性值。

表 1 过程内属性传播后 M1（图 3(c)）各结点的属性值

属性 / 结点	Escape	Type	Source	Sites
1, 2	No_Field_Escape	{A}	{2}	{UNKNOWN}
3, 4, 5	No_Field_Escape	{B}	{3}	{UNKNOWN}
10, 11, 12	No_Field_Escape	{C}	∅	{S2}
6, 7, 8, 9	Global_Escape	{B}	∅	{S1}

表 2 过程内属性传播后 M2（图 3(d)）各结点的属性值

属性 / 结点	Escape	Type	Source	Sites
1, 2, 3, 4	No_Field_Escape	{B}	{1}	{UNKNOWN}
5, 6, 7, 8	Field_Escape	{B}	∅	{S3}
9, 10	No_Field_Escape	{A}	{9}	{UNKNOWN}

4 过程间分析（Inter-procedural Analysis）

过程内属性传播反映的是在方法内的操作对各结点属性值的影响，其计算是局部的，为了获得在整个程序中结点属性值的变化，必须进行上下文敏感的过程间分析。该分析通过参数、返回值以及调用点处的信息，对结点的属性值进行跨过程的计算。

4.1 调用点处调用方法的确定

Java 是一种具有动态类型（Dynamic Type）系统的语言，其方法重写（Override）、动态绑定等特性使得对调用点处所调用方法的确定变得困难。大部分研究工作，如[7][11][12]等，采用方法调用图（PCG，Program Call Graph）反映程序中方法间的调用关系。PCG 的引入使得调用点处的方法确定变得简单，但该结构的计算与维护会增加额外的开销以及复杂性，且其计算方式也过于保守，因此我们通过利用结点的 *Type* 属性值进行类型传播，以确定被调用的方法，而不是通过建立 PCG 图。

经过 *Type* 属性值传播计算后，根据各个调用点对应的连接图结点的 *Type* 属性信息确定方法调用接收者的类型，进而确定所调用的方法是该类型重写后的方法，还是其某个超类中的相应方法。例如，图 2 中方法 *M1* 内的语句 *d=M2*(*c*)将类型为 *C* 的变量 *c* 传给方法 *M2* 中类型为 *B* 的形参 *b*，而从方法 *M2* 的代码上看，语句 *b.foo*( )调用的是类 *B* 中的方法 *foo*，经过过程间的属性值传播计算后，*M2* 中形参 *b* 的属性值 *Type*(*b*)={*C*}，根据这个属性值即可判断语句 *b.foo*( )处调用的是类 *C* 中的方法 *foo*，而不是类 *B* 中的。

4.2 过程间属性传播

过程间的属性传播主要依靠方法间形参结点与实参结点、返回值结点与其接收者结点之间属性的传播来实现。假设当前被分析的方法为 *M*，*cs* 是 *M* 中的一个调用点，*H* 是 *cs* 处的被调用方法。为进行过程间的

属性传播，分析 *H* 时需获得 *cs* 处所有的实参结点的属性值信息，而分析 *M* 时也需获得对 *H* 的分析结果（即 *H* 中返回值结点和形参结点的属性值信息）以不断更新 *M* 中连接图结点的属性。

图 4 描述了过程间属性传播算法。对该算法解释如下：第 1 行为 *M* 建立并修剪连接图，若 *M* 对应的连接图已经建立，则直接获得即可。第 2 行先取得 *CG* 中的形参结点集合 *FML*，再利用 combineArgNodes 函数将 *FML* 中的每一结点 *fml* 与传入的实参结点集 *InAANodes* 中对应的实参结点 *aa* 进行结点属性的合并操作 combine(*fml, aa*)（规则见图 1）。第 3 行是对 *CG* 进行过程内的属性传播。由于 Java 语言支持动态类载入(Dynamic Class Loading)，*H* 可能是尚未载入到虚拟机中的方法，也可能是本地方法（Native Method），算法对这两类方法采取最保守的处理方式（见第 7 行）。为简便起见，用 combineNode(*node, ndSet*)表示将结点集 *ndSet* 中各结点 *nd* 的属性值依次合并到结点 *node* 的属性中，即 $\forall nd \in ndSet$，执行 combine(*node, nd*)。引入 *Source* 属性是为了辅助连接图结点属性的局部更新，第 9 行通过 combineNode 函数和 *Source* 属性更新 *cs* 处的实参集 *ACT* 中各结点的属性，而第 14 行同样利用它们更新 *M* 的返回值结点的属性。第 11 行通过 combineNode 函数将从 *H* 传出的返回值结点集的属性值合并到调用点的返回值接收者结点 *dst* 的属性中。第 12 行的 combineNodesOnEscape 函数与 combineArgNodes 函数处理过程类似，区别在于前者只对结点的 *Escape* 属性进行合并，这一步的意义在于保证 *M* 中的结点能够获得其引用的对象在所有分析过的程序部分中的逃逸状态。第 16 行的 calculateDeadObjectsInM 函数则通过引入一个数据流方程计算在 *M* 中死亡（即生命期结束）的对象分配点集合。

```
interProceduralAnalysis (M, InAANodes , &OutRETNodes, &OutFANodes)
输入：M – 待分析的方法，InAANodes – M 的调用者连接图中调用 M 时的实参结点集
输出：OutRETNodes – M 的连接图中的返回值结点集，OutFANodes – M 的连接图中的形参结点集
{
1    CG = initializeStructure(M);
2    FML = getFANodes(CG); combineArgNodes(FML, InAANodes);
3    intraProceduralPropertiesPropagation(CG);
4    CS = getCallSites(M)
5    foreach cs in CS    {
6      ACT = getAANodes(cs); dst = getDSTNode(cs); H=getCalleeMethod(cs);
7      if ( isNative(H) or isUnloaded(H) ) setConservativeProperties(ACT∪{dst});
8      else{
9          foreach act in ACT    combineNode(act, Source(act) )
10         interProceduralAnalysis (H, ACT, InRETNodes, InFANodes)
11         combineNode(dst, InRETNodes);
12         combineNodesOnEscape(ACT, InFANodes);     }
       }
13   RET=getRETNodes(CG);
14   foreach ret in RET    combineNode(ret, Source(ret) );
15   OutRETNodes = RET; OutFANodes = FML;
16   deadobjs = calculateDeadObjectsInM();
}
```

图 4　过程间的属性传播算法

由于连接图是独立于调用上下文对方法进行描述的结构，因此，对先前未载入虚拟机的方法或由于某些原因没有被分析的方法，在以后被分析时可以很方便地利用图 4 中的算法再进行属性值的更新计算与传播，达到增量式处理。

4.3 对象分配点生命期结束的确定

计算方法 *M* 中死亡（即生命期结束）的对象分配点，可以分为两步。首先，我们引入一个数据流方程（图 5）来计算可能在 *M* 中死亡的对象分配点集合。在 *M* 中死亡的对象，其分配点只可能在 *M* 或被 *M*（直接或间接）调用的方法中创建，这是因为在直接或间接调用 *M* 的方法中创建的对象，在 *M* 方法调用结束后，

仍然可以被其它方法所访问，因而我们认为其生命期超过了 *M* 的生命期。图 5 中 IN(*M*)表示的是从被方法 *M* 直接或间接调用的方法传递到 *M* 的对象分配点集合，OUT(*M*)表示的是方法 *M* 调用结束后返回到调用者的分配点集合，DEAD_SITE_CANDIDATES(*M*)则表示的是可能在 *M* 中死亡的对象的分配点集合，它由在 *M* 中创建的分配点集合并上 IN(*M*)集合，再减去 OUT(*M*)集合构成。

$$\mathrm{IN}(M) = \bigcup_{n_d \in N_{dst} \text{ in } M} Sites(n_d) \tag{2-1}$$

$$\mathrm{OUT}(M) = \left(\bigcup_{n_f \in N_{fo} \text{ in } M} Sites(n_f)\right) \cup \left(\bigcup_{n_r \in N_{ret} \text{ in } M} Sites(n_r)\right) \tag{2-2}$$

$$\mathrm{DEAD\_SITE\_CANDIDATES}(M) = \left(\bigcup_{n_o \in N_o \text{ in } M} Sites(n_o)\right) + \mathrm{IN}(M) - \mathrm{OUT}(M) \tag{2-3}$$

图 5　确定对象生命期在方法 M 中结束的数据流方程

再者，通过数据流方程计算得到的分配点集合只反映了可能在 *M* 中死亡的对象。判断该集合中的元素 *s* 是否真在 *M* 中死亡，还需根据其在连接图上的对应结点 *n* ($n \in N_o$)的 *Escape* 属性进行判断。假设 *obj* 是 *s* 处分配的对象：1)若 *Escape*(*n*)=Global_Escape，说明 *obj* 是全局逃逸的，即可以被 *M* 所在线程之外的线程所访问，其生命期不可能在 *M* 中结束；2) 若 *Escape*(*n*)=No_Field_Escape，由于 *obj* 对象没有被其它任何对象的域引用，故可以判定它在 *M* 中死亡；3) 若 *Escape*(*n*)=Field_Escape，由于 $s \in$ DEAD_SITE_CANDIDATES(*M*)表明假如 *obj* 没有被其它对象的域引用，则它将会在 *M* 中死亡，因此其在 *M* 中死亡的充分必要条件是：$\forall <x, fd, s> \in RefsMap$，$x \in AS$，$fd \in FIELD$，如果有 *x* 处对象在方法 *M* 中或被 *M* 调用（直接或间接）的方法中死亡，那么 *obj* 在 *M* 中死亡。

综上所述，拥有 Global_Escape 状态的对象分配点是全局分配点，我们无法判断其生命期；而拥有 No_Field_Escape 状态或 Field_Escape 状态的对象分配点属于线程局部的分配点，是可判断生命期的分配点。

5　实验结果与分析

表 3　实验结果比较

程序	分析时间(s)	分配点总数	全局分配点	可判断生命期分配点		G&S&D 分析	
				非相互引用模式	相互引用模式	分配点总数	内部分配点
bh	0.728567	75	23	34	52	41	21
bisort	0.100093	14	1	13	13	10	7
em3d	0.167738	34	4	22	30	26	11
health	0.392583	56	27	29	29	28	13
mst	0.289910	15	1	11	14	16	8
perimeter	0.095636	11	5	6	6	13	7
power	0.514750	27	4	16	23	21	9
treeadd	0.021868	10	4	6	6	11	6
tsp	0.245009	13	2	11	11	12	7
voronoi	4.986505	185	25	110	160	35	20

笔者在开源的 Java SE 平台 Apache Harmony[14]上，用 C++在 Harmony 的 JIT 中实现了基于逃逸分析的对象生命期分析算法的原型。分析是基于 JIT 中 SSA 格式的高级中间表示 HIR 实现的。实验平台为 Windows XP 操作系统，CPU 为 P4，主频为 2.8G，内存为 512M，测试用例为 Jolden benchmark。

本文的逃逸分析方法以[7]为基础，并以此来实现对象的生命期分析，而[7]则利用逃逸分析展开对象的栈上分配和同步消除的优化应用。由于两者在逃逸分析的目的上不同，从而导致两者的实验内容和测试角度也不一样。[7]中的实验是围绕栈上分配和同步消除的优化效果而展开的；而本文的实验则应围绕着测试哪些对象可以在编译时检测出其生命期在哪些方法中结束而展开的，因此与[7]中的实验结果可比性较差。笔者选择[11]中的实验数据作为本文实验的对比参照，理由有二：[11]中提出了一种以基于区域的对象分配

优化为目的的逃逸分析方法，由于基于区域的对象分配是对象生命期分析的一个重要的优化应用，其较之栈上分配和同步消除优化需要更多有关对象生命期的信息，因此从这个意义上讲，[11]中的分析较之[7]中更为精确。其二，[11]中的测试结果偏重于反映分析本身的精确性，其测试出来的内部分配点（Inside Sites）包括了可以在编译时检测出其生命期结束时所在的方法的分配点，可以直接与本文的实验结果进行比较。

表 3 给出了我们的实验结果以及与[11]中实验结果的对比。由于程序中可能会出现递归调用等情况使分析陷入死循环，为此我们在实现上暂且对方法调用路径上第二次出现的相同方法予以最保守的处理，并且只对应用程序中的方法进行分析。表 3 中前两列分别表示测试用例的程序名、分析时间，最后两列则是[11]中的分析结果，其中，最后一列的内部分配点在[11]中表示可以在区域（Region）上进行分配的分配点。本文中的可判断生命期的分配点（即拥有 No_Field_Escape 状态或 Field_Escape 状态的分配点）是可以在区域上进行分配的，属于内部分配点。第 3 列与第 7 列分配点总数的比较，说明我们的分析相较之下能找出程序中更多的分配点，这是由于我们对程序中不同调用点处的相同方法均予以分析处理。第 5 列与第 6 列则是我们在是否考虑域引用图 *RefsMap* 中出现多个分配点之间相互引用的情况下所作的不同的分析，结果表明考虑分配点之间的相互引用的关系（第 6 列）会使得分析更为精确；在其二者与最后一列的比较中同时也表明我们的分析较之更为精确。

6 相关工作

[2]提出了一个在静态编译时对动态分配对象的生命期分析方法，但只适用于具有静态类型系统的语言。逃逸分析判断对象的生命期是否逃逸出一个方法或一个线程：[5]提出了一个快速的逃逸分析算法，但认为那些存入域中的对象逃逸了线程，分析较为保守且不精确；[7]提出了连接图的程序抽象，用于确立对象和对象引用之间的可达性关系，并在图上实现逃逸分析的传播算法，但其只识别三类逃逸状态，并且不能更精确地讨论对象的生命期。还有部分研究工作结合了指针分析和逃逸分析对对象生命期信息进行分析，如[3][6]的分析基于一个结合了指针分析和逃逸分析信息的称为指向逃逸图（points-to escape graph）的程序抽象；[1]提出了一个合并了对象之间引用关系和逃逸信息的称为并行交互图（parallel interaction graph）的程序抽象用于分析多线程之间对象信息的交互。[12]利用称为方法概要（Method Summary）的结构结合方法调用图（Call Graph）实现过程间的逃逸分析。这些分析采用了不同的结构和方法计算对象的逃逸状态并分析对象是否逃逸出创建它的方法和线程，而对未逃逸出线程的对象生命期在哪个方法内结束并没有作进一步的分析。我们的分析基于具有灵活性与上下文独立性的连接图结构，通过图中结点的逃逸属性识别对象的逃逸状态，并结合其它属性进一步分析非全局对象的生命期。[11]为程序中每个变量赋予一些事先定义的属性，在基于变量的过程内分析中计算这些属性值，并展开过程间的指针分析，从而为基于区域的分配优化提供必要的对象生命期信息。

逃逸分析主要应用于对象的栈上分配[3]~[7]和不必要同步的消除[1][3][4][7][10][12]，近年来也被应用于标量替换（Scalar Replacement）[10]、减小竞争检测范围[9]和基于区域的内存分配[1][11]等优化工作中。我们的分析同样可以应用在这些优化中，此外还可以用于辅助 GC 的对象显式回收[13]优化中。对本文中分析方法的应用将是我们的下一步工作。

7 结束语

本文提出的针对 Java 语言的、基于逃逸分析的、增量式过程间的对象生命期分析方法，不仅能分析对象是否全局共享，而且能计算和分析非全局共享对象的生命期在线程中的哪个方法内结束。该分析可以运用到基于区域的对象分配、对象显式回收等优化工作中。

我们也注意到目前我们的分析在更为精确地分析对象的活跃性上有很多的改进余地，这可以作为下一步的工作之一。此外，我们还将开展本分析的一个优化应用：辅助 GC 工作的对象显式回收优化。

References:

[1] J.Bogda, U.Holzle. Removing unnecessary synchronization in Java. In: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. Denver: 1999. pages:35-46

[2] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. San Diego: 1988. pages:285-293

[3] J.Whaley, M.Rinard. Compositional pointer and escape analysis for Java programs**.** In: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. Denver: 1999. pages: 187-206

[4] Bruno Blanchet**.** Escape analysis for object-oriented languages: application to Java**.** ACM SIGPLAN Notices, 1999, Volume 34 (Issue 10): 20-34

[5] D.GAY and B.STEENSGAARD. Fast escape analysis and stack allocation for object-based programs. In: International Conference on Compiler Construction. Berlin: 2000. pages: 82-93

[6] F. Vivien and M. C. Rinard. Incrementalized pointer and escape analysis. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. Snowbird: 2001. pages:35-46

[7] J.D.Choi, M.Gupta, M.J.Serrano,V.C.Sreedhar and S.P.Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. ACM Transactions on Programming Languages and Systems (TOPLAS), 2003, Volume25(Issue6): 876-910

[8] A.Salcianu and M.Rinard. Pointer and escape analysis for multithreaded programs. In: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming. Snowbird: 2001. pages:12-23

[9] H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In: Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM'04). California: 2004. pages :127-138

[10] T.Kotzmann, H.Mossenbock. Escape Analysis in the Context of Dynamic Compilation and Deoptimization. In: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments. Chicago: 2005. pages:111-120

[11] G.Salagnac, S.Yovine and D.Garbervetsky. Fast Escape Analysis for Region-based Memory Management. Electronic Notes in Theoretical Computer Science, 2005, pages:99-110

[12] Lei Wang,Xikun Sun. Escape Analysis for Synchronization Removal. In: Proceedings of the 2006 ACM symposium on Applied computing, Session: Object-oriented programming languages and systems (OOPS). Dijon: 2006. pages: 1419-1423

[13] S.Z.Guyer, K.S.McKinley and D.Frampton. Free-Me: a static analysis for automatic individual object reclamation. ACM SIGPLAN Notices, 2006,Volume41(Issue 6): 364-375

[14] http://harmony.apache.org/

附:

An Object Lifetime Analysis Method Based on Escape Analysis

Liu Yu-yu [1, 2], Zhang Yu [1, 2]

[1](Department of Computer Science & Technology, University of Science & Technology of China, Hefei 230027,China)

[2](Anhui Province Key Lab of Software in Computing and Communication, Hefei 230027,China)

Abstract: Region-based allocation and explicit garbage collection are currently two key ways for Java memory management optimizations. And these two kinds of techniques depend on object lifetime information. Escape analysis can capture the information whether an object escapes the method or thread where it is created. However, escape analysis is not precise enough for capturing the information on object lifetime and can not tell when and where a non-global object would be dead. In order to analyze the object lifetime in Java programs, a new incremental, compositional and inter-procedural analysis method is proposed here. It bases on escape analysis and uses a program abstract called connection graph. Intra-procedural and inter-procedural analysis algorithms

based on connection graph are brought forward, which propagate within and across methods four kinds of properties associated with each node in connection graph. Together with the escape property of the nodes in connection graph, data flow equations are also introduced to decide which objects in the Java methods analyzed are non-global and in which method they would be dead. The proposed method for object lifetime analysis are implemented on Apache Harmony platform and compared with the previous work for escape analysis. The experimental results show that our method can analyze more precisely than the previous work.

Key words: object lifetime; escape analysis; connection graph; property; propagation algorithm

研究背景

Java 语言的一个重要特点就是对象主要在内存堆（简称堆）中分配并同时进行初始化，它通过垃圾收集器(Garbage Collector，GC)自动管理对象的分配与回收。程序员使用 Java 语言进行程序开发时，只需考虑对象的创建，而不必考虑对象空间的释放。在 Java 程序运行时，GC 负责对应用程序中的对象分配空间，并主动回收不再需要的对象空间以供重复使用。GC 这种自动的内存回收机制在简化了 Java 程序开发工作的同时，也为系统的运行带来了额外的开销。

GC 在 Java 虚拟机中一般由一个或一组进程（线程）来实现，其本身也和用户程序一样需要占用内存空间，运行时也消耗 CPU 资源。在传统的 GC 实现中，当 GC 线程运行时，需要暂停应用程序的运行，若 GC 进行垃圾回收的过程较长，将会影响到应用程序的执行效率；而现有支持并行无用单元收集技术的 GC 虽然可以和应用程序同时运行，但需要考虑内存空间的同步等问题，这将会增加自动内存回收机制实现上的复杂性。因此，我们提出一个基于程序分析的显式内存回收机制以辅助 GC 的自动内存回收机制，其旨在减少 GC 处理的无用对象的数量，从而减少 GC 线程运行的时间。该项工作的实现必须依赖于程序分析技术获得可以进行显式回收的对象信息，而本文就是针对于此提出了一个基于逃逸分析的对象生命期分析技术。

我们选择 Apache Harmony 作为实现平台。Apache Harmony 提供了一个可扩展的、独立的、开源的 Java SE 运行体系框架，而 JIT（Just-in-time）即时编译器是其中的一个重要组成部分。JIT 具有一个平台无关的 Java 前端（front-end）以及一个平台相关的 Java 后端（back-end），且支持两级中间代码：高级中间代码（HIR）和低级中间代码（LIR）。HIR 是 JIT 前端的中间表示，形式上接近于 Java 字节码，且为静态单赋值格式（SSA），易于程序分析工作的进行。本文的工作则是基于 HIR 之上在 JIT 前端以优化遍的形式进行实现的。

此外，本文的工作成果还可以应用到其他诸如基于区域的对象分配、同步优化和并发性分析等分析优化工作上。其中，基于区域的对象分配也是减少堆中对象、减轻 GC 负担的一种有效的机制，其基本思想是将部分对象分配在方法或线程的私有区域中。在支持该分配机制的系统中，每个方法或线程都对应着一个私有的内存区域，可供其局部数据的空间分配使用，这些区域随着相应的方法或线程的创建而创建，并随其结束而被释放，因此在区域中分配的对象，其访问和回收操作相比于堆中对象开销要小。该分配方式也同样需要有对象生命期分析的支持。