第44卷 第2期 西 安 交 通 大 学 学 报 Vol. 44 №2
2010年2月 JOURNAL OF XI′AN JIAOTONG UNIVERSITY Feb. 2010

即时编译器辅助垃圾收集中的对象生命期分析研究

袁丽娜[1]，张昱[1,2]
（1.中国科学技术大学计算机科学与技术学院，230027，合肥；2.中国科学技术大学
苏州研究院软件安全实验室，215123，江苏苏州）

摘要：提出一种对象生命期分析算法，利用即时编译器分析应用程序并在其中安插显式释放对象等指令，通过辅助垃圾收集器改进对象的回收与分配来减轻垃圾收集器自动回收的负担.该算法结合了活跃变量分析和指针逃逸分析，对Java程序的每个方法仅分析一次，而且是过程间的，对域、上下文是敏感的，能够分析识别应用程序中的非全局对象及其死亡位置.实验结果表明：算法的执行时间占总编译时间的3.6%～5.3%；相比一般的Salagnac等指针逃逸分析，能识别出更多的对象生命期信息，而且对象死亡位置能精确到Java方法控制流图中的基本块；在即时编译器辅助的垃圾收集优化中能够显式地回收较多的内存空间.

关键词：对象生命期；活跃变量分析；指针逃逸分析；内存管理优化
中图分类号：TP311；TP314 **文献标志码**：A **文章编号**：0253-987X(2010)02-0050-06

Study on Object Lifetime Analysis for Just-in-Time Compiler Assisted Garbage Collection

YUAN Lina[1], ZHANG Yu[1,2]
(1. School of Computer Science & Technology, University of Science & Technology of China, Hefei 230027, China;
2. Software Security Laboratory, Suzhou Institute for Advanced Study, University of
Science and Technology of China, Suzhou, Jiangsu 215123, China)

Abstract: An object lifetime analysis algorithm is presented, which analyzes Java applications in just-in-time compiler in order to insert instructions to explicitly "free" objects into the applications, and reduces garbage collection overhead by assisting garbage collector in improving object allocation and reclamation. The proposed algorithm combines the live variable analysis and the pointer and escape analysis, and analyzes each method in Java applications only once. Moreover, the algorithm is inter-procedural, field sensitive and context sensitive, which can identify non-global objects and their dead locations in Java applications. Experimental results show that the proposed algorithm consumes 3.6%-5.3% of the total compilation time. The algorithm not only can identify the lifetime information of more objects than traditional ordinary escape analysis (such as that of Salagnac, et al), but also can determine the object dead locations with the basic block precision in control flow graph of a method; Moreover, the algorithm can explicitly reclaims more memory spaces in just-in-time compiler assisted garbage collection.

Keywords: object lifetime; live variable analysis; pointer and escape analysis; memory management optimization

Java语言采用垃圾收集器(GC)在堆上处理 Java应用程序的对象分配请求并自动管理对象的回

收稿日期：2009-07-27. 作者简介：袁丽娜(1985—)，女，硕士生；张昱(联系人)，女，副教授. 基金项目：Intel公司研究基金资助项目；国家自然科学基金资助项目(60673126).

收. GC 减轻了程序员管理内存的负担,但需要耗费大量的时空开销来识别堆中哪些对象是不可达的(死亡的)并回收这些死亡对象,从而成为影响 Java 虚拟机(JVM)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 算法以外,另一类重要的解决方法是编译器辅助的内存管理. 该类技术主要有栈上分配[1-3]、基于区域的内存管理[4-6]、对象的显式回收[7-8]和空间复用[9]. 它们都要先确定对象的分配或回收方式,这就需要分析对象的生命期等特征. 与对象生命期相关的分析技术主要有逃逸分析和指针分析等,现有算法[1-5,10]可以识别出对象是否逃逸出某个作用域(如方法或线程),但对于对象生命期的识别精度不够. 为此,本文提出一种结合活跃变量分析和指针逃逸分析的对象生命期分析算法.

1　即时编译器辅助的垃圾收集框架

在开源的 Java SE 平台 Apache Harmony[11]之上设计实现了即时编译器(JIT)辅助的垃圾收集. 整个框架如图 1 所示,其中阴影部分是本文的重点.

JIT 采用流水线机制来编译优化 Java 字节码. 流水线依次载入待编译方法的字节码,将其先后翻译成与平台无关的高级中间表示(HIR)和与平台相关的低级中间表示(LIR),最后发射为可执行的本地代码,并在 HIR 和 LIR 上分别执行一系列的优化遍.

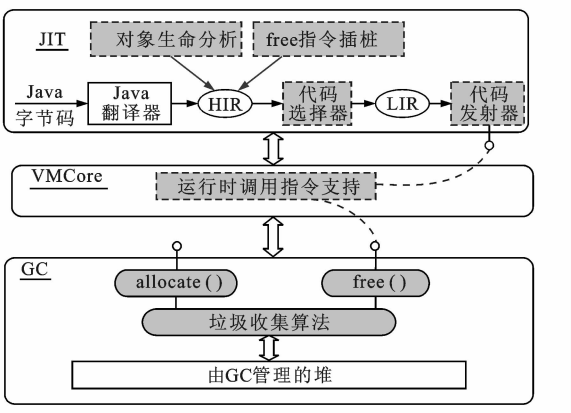

图 1　即时编译器辅助的垃圾收集框架

JIT 负责:①在 HIR 和 LIR 中增加支持对象显式释放的指令;②在 HIR 上添加一个优化遍以执行对象生命期分析和支持显式对象释放等指令的插桩;③修改代码选择器和代码发射器,使之支持从扩展的 HIR 到扩展的 LIR 的变换并能够将显式对象释放指令发射为运行时的指令调用.

VMCore 负责协调 JIT 和 GC 的交互. 当需要添加新的调用时,VMCore 将显式释放指令生成的指令调用映射到 GC 提供的对象显式回收接口 free().

GC 负责为对象显式回收提供 free(),改进对象分配接口 allocate(),以支持显式回收内存空间的及时重用,详见文献[12].

2　指向逃逸图

对象生命期分析采用指向逃逸图(PEG)作为程序抽象. PEG 描述了变量和对象之间的引用关系、对象与对象间的域引用关系以及对象的逃逸信息,每个待分析的 Java 方法都可以抽象出一个 PEG 与之相对应.

2.1　基本结构

定义 1　设 m 是一个待分析的方法,V 和 P 分别表示 m 内的变量和形参集合,对应的指向逃逸图是一个有向图,记为 $G=(N_o\cup N_r, E_p\cup E_f)$,其中的元素描述如下.

(1)$N_o=N_c\cup N_p$ 表示 m 内访问的对象结点集合. 其中:N_c 为 m 内分配点处创建的对象结点集合;$N_p=N_{\mathrm{fp}}\cup N_{\mathrm{in}}$ 表示在 m 外创建、但在 m 内使用的对象结点集合,称这类对象为虚对象,N_{fp} 是在调用 m 的方法中创建并通过 m 的形参引用的对象结点集合,N_{in} 是在被 m 调用的方法中创建并通过 m 内各调用点处返回值的接收者及它的域或实参的域传入到 m 中的对象结点集合.

(2)$N_r=V$ 表示 m 中使用的引用结点集合,一个变量对应一个引用结点.

(3)$E_p\subseteq N_r\times N_o$ 是指向边集合. 弧$\langle v,o\rangle\in E_p$ 表示引用结点(变量)v 可能指向对象结点 o.

(4)$E_f\subseteq N_o\times F\times N_o$ 表示域边集合,其中 F 描述 m 中的非静态域. $\langle o_1,f,o_2\rangle\in E_p$ 表示对象结点 o_1 的域 f 可能指向对象结点 o_2.

在 PEG 中,只有指向边和域边,并没有如文献[2]所述的延迟边,即引用结点指向引用结点的边. 在分析构造 PEG 时,若遇到变量间的赋值,将延迟边转换为指向边可以降低图的规模,减少属性在结点间的传播计算过程.

为了进行辅助分析,将指向逃逸图中的对象结点 o 赋予逃逸属性.

定义 2　若方法 m 的指向逃逸图 $G=(N_o\cup N_r, E_p\cup E_f)$,对于 G 中任意对象结点 $o\in N_o$,其逃逸属性 $\xi(o)$ 取满足格 $\mathscr{E}$ 的 2 种值之一,即 $\mathscr{E}_{\mathrm{N}}\prec\mathscr{E}_{\mathrm{G}}$. 其中:$\mathscr{E}_{\mathrm{G}}$ 表示对象是全局逃逸的,全局逃逸的对象可以被

多个线程访问；$\mathscr{E}_N$ 表示对象可能不是全局逃逸的，即对象可能只被一个方法或一个线程访问.

定义 3 对于一个对象 o，$\xi(o)=\mathscr{E}_N$，如果在某程序点 p 之后，没有变量或者其他对象的域引用对象 o，则认为 o 在 p 处死亡，称 p 为 o 的死亡点.

2.2 基本操作

定义 4 对于指向逃逸图 G 中的 2 个对象结点 o_1，$o_2\in N_o$，o_1 的逃逸属性合并到 o_2 的逃逸属性上的合并操作 $A_C(o_1,o_2)$ 规则为

$$\frac{s\in\mathscr{E},\quad s=\xi(o_1),\quad \xi(o_2)\prec s}{\xi(o_2):=s} \tag{1}$$

定义 5 对于指向逃逸图 G 中的 1 个对象结点 o，$\xi(o)=\mathscr{E}_G$，则属性传播操作 $A_P(o)$ 将 G 中所有从 o 可达的对象结点的逃逸属性设置为 $\mathscr{E}_G$.

$A_C(o_1,o_2)$ 操作是对结点 o_2 的逃逸属性有选择地进行重新赋值，而结点 o_1 的逃逸属性值保持不变. 当进行域赋值或者进行过程间信息结合时，执行 $A_C()$ 操作，而在静态域赋值时，一般需要执行 $A_P()$ 操作.

在分析的过程中，当变量 v 不再活跃时，需对指向逃逸图执行修剪操作 $A_D(v)$，即从指向逃逸图中删除所有从 v 出发的指向边. 若一个对象结点 o 死亡，此时也会执行修剪操作 $A_D(o)$，即从指向逃逸图中删除所有从 o 出发的域边.

定义 6 对于指向逃逸图 G 中的 1 个对象结点 $o_1\in N_o$ 和 1 个引用结点 $v\in N_r$，G 和 G' 分别表示执行修剪操作 $A_D()$ 前后的指向逃逸图. $A_D()$ 操作定义如下

$$\frac{G=(N_o\cup N_r,E_p\cup E_f),\ v\in N_r,\ E_p^v=\{\langle v,o\rangle\mid o\in N_o\}}{G'=(N_o\cup N_r',E_p'\cup E_f),\ N_r'=N_r-\{v\},\ E_p'=E_p-E_p^v} \tag{2}$$

$$\frac{G=(N_o\cup N_r,E_p\cup E_f),\ o_1\in N_o,\ E_f^{o_1}=\{\langle o_1,f,o_2\rangle\mid o_2\in N_o\}}{G'=(N_o'\cup N_r,E_p\cup E_f')N_o'=N_o-\{o_1\}E_f'=E_f-E_f^{o_1}} \tag{3}$$

2.3 对象生命期分析结果

定义 7 对于一个方法 m，假设其对应的指向逃逸图 $G=(N_o\cup N_r,E_p\cup E_f)$，$m$ 的对象生命期分析结果是一个二元组 $\langle\mathscr{M},\mathscr{D}\rangle$，其中 $\mathscr{M}$ 和 $\mathscr{D}$ 解释如下.

(1) $\mathscr{M}=(N_{fp}\cup N_{ret},E'_f)$ 是 m 的方法概要. 其中：N_{fp} 和 N_{ret} 分别记录了 m 的形参和返回值在 G 中对应的对象结点信息；E'_f 是 E_f 的子集，即 $E'_f\subseteq E_f$，且为从对象结点 $N_{fp}\cup N_{ret}$ 出发的域边. 对于 m 来说，它的形参或返回值引用的对象以及这些对象的域所引用的对象一般不能在 m 中显式回收，因为 m 的调用者可能要使用这些对象，所以应把这些对象的信息记录到方法概要中.

(2) $\mathscr{D}$ 描述所有在 m 中死亡的对象信息. $\mathscr{D}$ 是一个三元组集合，每个三元组记为 $\langle o,r,p\rangle$，表示对象结点 $o\in N_o$ 在程序点 p 之后死亡，其中 $r=\langle v,f\rangle$ 表示 o 的一个引用，若 f 为 NULL，说明变量 v 是 o 的引用，否则 $v.f$ 是 o 的引用.

3 对象生命期分析

对象生命期分析需假定程序基于静态单赋值(SSA)格式，且所有形如 $x.y.f$ 这类多级域的访问均转化为 $t=x.y$ 形式和 $t.f$ 形式.

3.1 活跃变量分析

活跃变量分析是后向数据流分析. 如果一个变量 v 在某个程序点之后是活跃的，则表明在这个程序点之后至少存在 1 条路径，在这条路径上使用了 v 且在使用之前没有对 v 重新定值.

对于控制流图 F 中的 1 个基本块 B，用 $L_{entry}(B)$ 和 $L_{exit}(B)$ 分别描述在 B 的入口和出口处的活跃变量信息. 活跃变量分析的数据流方程为

$$L_{exit}(B)=\begin{cases}\varnothing, & B\text{ 是控制流图的退出块}\\ \bigcup L_{entry}(B'), & B'\text{是 }B\text{的后继}\end{cases}$$

$$L_{entry}(B)=L_{exit}(B)-L_{kill}(B)+L_{gen}(B)$$

式中：$L_{kill}(B)$ 表示在 B 中定值且定值前未在 B 中引用的变量集合；$L_{gen}(B)$ 表示在 B 中引用且引用前未在 B 中被定值的变量集合.

假设用 $V(e)$ 表示表达式 e 中出现的变量集合. 对于赋值语句 $e_1=e_2$，$L_{kill}[e_1=e_2]=\{e_1\}$，$L_{gen}[e_1=e_2]=\{v\mid v\in V(e_2)\}$；对于条件表达式 $L_{kill}[e]=0$，$L_{gen}[e]=\{v\mid v\in V(e)\}$；对于调用语句 $v=c(v_0,\cdots,v_n)$，$L_{kill}[v=c(v_0,\cdots,v_n)]=\{v\}$，$L_{gen}[v=c(v_0,\cdots,v_n)]=\{v_i\mid i=0,\cdots,n\}$.

m 中的变量与 m 的指向逃逸图中的引用结点一一对应，对于不再活跃的变量 v，可以在指向逃逸图中执行 $A_D(v)$ 操作，这不仅可以简化指向逃逸图，而且可以使分析精度提高到基本块的级别.

3.2 指向图构造

假设当前被分析的 Java 方法为 m，算法将逐条扫描 m 的指令，逐步完成指向逃逸图的创建. 下面给出影响指向逃逸图构造指令的处理规则. 由于引用结点和变量是一一对应的，因此不需要重新创建引用结点，只需将变量放入引用结点集合即可.

(1)m 中的形参:假定 P 是 m 中的形参集合,$\forall v_i^p \in P$,令 $v_i^p \in N_r$,创建 1 个虚对象结点 $o \in N_{\text{fp}}$,并增加 1 条指向边 $\langle v_i^p, o\rangle \in E_p$. 令 $\xi(o)$ 为 $\mathscr{E}_{\text{N}}$.

(2)v=new C 或 v=new $C[]$:令 $v \in N_r$,并为等式右边新创建的对象建立 1 个对象结点 $o \in N_c$,并增加指向边 $\langle v, o\rangle \in E_p$. 如果 o 是线程对象,设置 $\xi(x)$ 为 $\mathscr{E}_{\text{G}}$,否则设为 $\mathscr{E}_{\text{N}}$.

(3)$v_1 = v_2$:令 $v_2 \in N_r$,$v_2 \in N_r$. 假设 $O=\{o \mid \langle v_2, o\rangle \in E_p\}$,对 $\forall o \in O$ 增加指向边 $\langle v_1, o\rangle \in E_p$.

(4)v=phi$(v_1, v_2, \cdots, v_n)$:令 $v \in N_r$. 假设 $O=\{o \mid \langle v_i, o\rangle \in E_p, i=1,2,\cdots,n\}$,对 $\forall o \in O$ 增加指向边 $\langle v, o\rangle \in E_p$.

(5)$v_1.f = v_2$(f 表示非静态域):假设 $X=\{x \mid \langle v_1, x\rangle \in E_p\}$,$Y=\{y \mid \langle v_2, y\rangle \in E_p\}$. $\forall x \in X$,$\forall y \in Y$,增加域边 $\langle x, f, y\rangle \in E_f$,并执行 $A_{\text{C}}(x, y)$,如果 y 的逃逸属性变为 $\mathscr{E}_{\text{G}}$,则执行 $A_{\text{P}}(y)$.

(6)$v_1 = v_2.f$(f 表示非静态域):令 $v_1 \in N_r$. 假设 $X=\{x \mid \langle v_2, x\rangle \in E_p\}$,$Y=\{y \mid \langle x, f, y\rangle \in E_f, x \in X\}$. $\forall y \in Y$,增加指向边 $\langle v_1, y\rangle \in E_p$. 如果 Y 为空集,则说明 $v_2.f$ 域所引用的对象是在方法 m 外创建的,此时创建 1 个虚对象结点 $o \in N_p$. $\forall x \in X$,增加域边 $\langle x, f, o\rangle \in E_f$,增加 1 条指向边 $\langle v_1, o\rangle \in E_p$.

(7)$C.f' = v$(f' 表示类 C 中的静态域):令 $O=\{o \mid \langle v, o\rangle \in E_p\}$,对 $\forall o \in O$,设置 $\xi(o)$ 为 $\mathscr{E}_{\text{G}}$,执行 $A_{\text{P}}(o)$.

(8)$v = C.f'$(f' 表示类 C 中的静态域):建立 1 个虚对象结点 $o \in N_p$,令 $\xi(o)$ 为 $\mathscr{E}_{\text{G}}$,并增加 1 条指向边 $\langle v, o\rangle \in E_p$.

(9)$v = v_0.c(v_1, v_2, \cdots, v_n)$:建立 1 个虚对象结点 $o \in N_p$,增加 1 条指向边 $\langle v, o\rangle \in E_p$. 这里只是初步处理,在过程间进行信息传播时会进一步处理它.

(10)return v:假设 $O=\{o \mid \langle v, o\rangle \in E_p\}$,$\forall o \in O$,令 $o \in N_{\text{ret}}$.

需要说明的是规则(7)和规则(8),当一个对象 o 被某个类的静态域 $C.f'$ 引用时,只是简单地把 o 的逃逸属性设置为 $\mathscr{E}_{\text{G}}$ 并执行 $A_{\text{P}}(o)$,为了简化,没有在 $C.f'$ 和 o 之间建立对应关系. 当访问某个类的静态域 $C.f'$ 所引用的对象时,算法为 $C.f'$ 引用的对象建立一个全局逃逸的虚对象结点,因为类的静态域所引用的对象是全局逃逸的,而全局逃逸对象在本文的算法中是不会判定为死亡的.

3.3　过程间的信息传播

对象生命期分析算法对每个 Java 方法只分析 1 次,在遇到调用点的时候,需要根据被调用者的方法摘要进行过程间的信息结合.

假设当前被分析的方法为 m,其指向逃逸图为 $G=(N_o \cup N_r, E_p \cup E_f)$,若某调用点处的指令 h 为 $v = v_0.c(v_1, v_2, \cdots, v_n)$,被调用方法 c 的摘要为 $\mathscr{M}_c = \langle N'_{\text{fp}} \cup N'_{\text{ret}}, E'_f\rangle$,则过程间信息传播算法 $A_{\text{ipa}}(\eta, \mathscr{M}_c)$ 的步骤如下.

(1)结合形参和实参. 假设 $v_i^p \in N'_{\text{fp}}$ 是 $\mathscr{M}_c$ 的形参结点,对于其对应的实参 v_i 以及 $\langle v_i, o\rangle \in E_p$,执行 $A_{\text{C}}(v_i^p, o)$.

(2)结合返回值和返回值接收者. 假设 $r \in N'_{\text{ret}}$ 是 $\mathscr{M}_c$ 中的返回值结点,对于该返回值的接收者 v 以及 $\langle v, o\rangle \in E_p$,执行 $A_{\text{C}}(r, o)$.

(3)映射域边. 对于在 E'_f 中的任一条域边,在 E_f 增加相应的域边.

(4)属性传播. 在前面的 3 个步骤中,如果有某个对象结点 o 的逃逸状态变为 $\mathscr{E}_{\text{G}}$,执行 $A_{\text{P}}(o)$.

3.4　对象生命期分析算法

对象生命期分析算法 $A_{\text{ola}}()$ 的输入为待分析方法 m 的控制流图 F,输出为 m 的方法摘要和所有在 m 中死亡的对象信息.

$A_{\text{ola}}()$ 依次扫描 m 的指令,根据 3.2 节的规则构造指向逃逸图. 如果遇到调用指令,若 c 是尚未载入到 JVM 中的方法或者是本地方法,$A_{\text{ola}}()$ 对这 2 类方法采取最保守的处理方式,如图 2 的 7~8 行所示;若 c 尚未被分析,则先启动对 c 的对象生命期分析,接着进行过程间的信息传播 $A_{\text{ipa}}()$.

在每个基本块的尾部,会根据活跃信息对指向逃逸图进行修剪,如图 2 的 14~15 行所示. 在 m 中可能死亡的对象分为 2 类:在 m 内的分配点创建的对象;在 m 外创建并引入到 m 中的对象,这些对象通过 m 内的调用点处的返回值接收者及其域、实参的域被引入至 m 中. 在修剪指向逃逸图后,会判断哪些对象死亡,这些对象结点属于 $N_c \cup N_{\text{in}} - N_{\text{ret}}$ 集合,如图 2 的 16~19 行所示. 如果对象 o 死亡,则将死亡对象信息记录到 $\mathscr{D}_m$ 中. $A_{\text{recDead}}(o, \mathscr{D}_m)$ 是一个递归方法,它首先调用 $A_{\text{D}}(o)$ 修剪指向逃逸图,然后检查此次修剪指向逃逸图是否会产生新的死亡对象,如果有新的死亡对象,它们一定属于 o 的域所引用的对象,需递归调用下去. 在记录一个对象死亡时总是先检查它的域所引用的对象是否死亡. 在对 m 的每条指令进行分析之后,应将方法摘要信息记录到 $\mathscr{M}_m$ 中.

对于一个 m,假设其对应的控制流图中有 n_b 个基本块(即结点). 过程内分析会对控制流图中的结

对象生命期分析算法A_{ola}：逐条扫描方法m的控制流图F的每条指令，进行对象生命期分析。
输入：F_m－待分析方法m的控制流图
输出：$(\mathcal{M}_m, \mathcal{D}_m)$
1: $L=A_{liveness}(F_m)$; //对m进行活跃分析
2: for each basic block B in F_m {
3: for each instruction η in B {
4: $A_{PEG}(\eta)$; //根据规则构造指向逃逸图
5: if (η is a call instruction){
6: get the current callee c in η ;
7: if (c is a native method or is unloaded)
8: continue;
9: if (c is not analyzed)
10: $(\mathcal{M}_c, \mathcal{D}_c)=A_{ola}(F_c)$;
11: $A_{ipa}(\eta, \mathcal{M}_c)$;
12: }
13: }// end for instruction η in B
14: for each v in N_r which is not live at the exit of B accroding to L {
15: $A_D(v)$; // 指向逃逸图的修剪操作
16: for each o in $N_c \cup N_{in}$-N_{ret} {
17: if (o is dead)
18: $A_{recDead}(o, \mathcal{D}_m)$;
19: }
20: }
21: }//end for each basic block B in F_m
22: record the information of formal parameters, return values and field references into $\mathcal{M}_m$

图2 对象生命期分析算法

点扫描2次：1次是逆向的活跃变量分析；1次是正向地创建并修剪指向逃逸图的过程，从而过程内分析的时间复杂度是$O(n_b)$，n_b为被调用方法的平均基本块数. 过程间的信息传播是将被调用方法c的方法摘要信息$(N'_{fp} \cup N'_{ret}, E'_f)$结合到$m$的指向逃逸图中. 该过程需依次扫描$N'_{fp} \cup N'_{ret}$和$E'_f$，并修改$m$的指向逃逸图，其时间复杂度为$O(n_p)$，其中$n_p=|N'_{fp} \cup N'_{ret}|+|E'_f|$为$c$的方法摘要规模.

假设调用点处的被调用方法c未被分析，就需要启动对c的对象生命期分析. 如果一直这样进行下去，会造成在分析m时需要分析应用程序中的绝大多数的方法. 假设从m出发的调用图上的结点数为n_c，边数为α_c，则最坏情况下分析m的时间复杂度为$O(n_c n_b+\alpha_c n_p)$. 启动对被调用方法的对象生命期分析能够提高分析精度，但导致效率降低，甚至运行时出现栈溢出. 精确性和效率是需要平衡的，采取的方法是在层次调用达到门限(经验值是5)以后，就不再对被调用方法进行分析.

3.5 特殊处理

(1)线程对象. 当一个线程o_t启动后，一般很难判断它什么时候运行结束，因此它所占用的内存空间一般不会被释放. 对象生命期分析通过类层次图来识别线程对象，并将逃逸状态初始化为$\mathcal{E}_G$. 当o_t运行结束时(一般通过线程对象的join()方法判断)，o_t退化为普通对象，根据此时指向逃逸图中引用它的对象逃逸状态信息可对$\xi(o_t)$进行重新设置.

(2)数组. 将一个数组中所有的数组元素抽象为一个对象，对数组元素的访问视为特殊的域访问. 这样的处理可能会影响分析精度，如数组中的某些元素可能是死亡的，但由于将所有元素视为一个整体，所以此时并不会判断出数组及其元素死亡.

(3)循环. 对程序中的循环结果只分析一次，一般不会影响对象生命期分析的精度.

4 实验结果与分析

在Apache Harmony上用C++实现了对象生命期分析算法. 实验平台为Windows XP，AMD Athlon™ 64×2 Dual Core Processor 3800，2.0 GHz，896 MB. 测试用例为Jolden[13]中的4个基准程序.

表1为应用程序对象生命期分析的执行时间与总编译时间. 总编译时间为方法的Java字节码编译成本地码的总时间. 可以看出，对象生命期分析算法的执行时间约占总编译时间的3.6%～5.3%.

表1 对象生命期分析的执行时间与总编译时间

基准程序	对象生命期分析时间/ms	总编译时间/ms	对象生命期分析时间与编译时间之比/%
BH	23	537	4.3
TSP	11	207	5.3
Power	12	332	3.6
Health	14	309	4.5

表2给出了本文与文献[5]算法的分析结果对比，其中内部分配点是文献[5]中提到的概念，表示可以在区域上进行分配的分配点，而本文识别的可判断生命期的分配点(即逃逸状态为$\mathcal{E}_N$)是可以在区域上进行分配的，属于内部分配点. 通过对比可以看出，本文算法相比文献[5]算法能识别出更多可判断生命期的分配点，而且对对象死亡位置的定位可精确到基本块.

将对象生命期分析结果应用到即时编译器辅助的垃圾收集系统中，其统计结果如表3所示. 通过对比可以看出，大部分程序的内存空间都可以被显式地回收. Health释放了23%的对象空间，Power分配了24 MB的内存，显式释放了23 MB的内存空

表 2　本文与文献[5]算法的分析结果对比

基准程序	文献[5]识别的内部分配点数	本文识别的可判断生命期的分配点数
BH	21	50
TSP	7	14
Power	9	16
Health	13	28

表 3　显式回收和重用的对象情况

基准程序	分配对象数，空间大小/MB	显式回收对象数，空间大小/MB	重用对象数，空间大小/MB
BH	2 701 783,67	2 438 018,60	2 438 013,60
TSP	1 599 242,51	1 048 578,28	1 048 575,28
Power	809 979,24	760 004,23	760 000,23
Health	3 388 012,60	600 009,14	600 004,14

间. 这说明，对象生命期分析得到的死亡对象信息使得其在即时编译器辅助的垃圾收集系统中的应用情况非常好. 从重用的对象信息可以看出，超过 90% 显式回收的对象空间都能被重用. 通过分析源程序得出，这些死亡对象的分配点一般位于循环或递归方法中，这些地方显式回收的空间基本上可被重用.

5　结束语

本文提出一种结合了活跃变量分析和指针逃逸分析的对象生命期分析算法，能够获得精确的对象生命期信息. 算法在实际的 JVM 中得以实现并应用于即时编译器辅助的垃圾收集. 实验结果表明，算法的执行时间占总编译时间的 3.6%～5.3%，分析精确度比一般的逃逸分析更好，并能指导优化工作.

参考文献：

[1] WHALEY J, RINARD M. Compositional pointer and escape analysis for java programs [J]. ACM SIGPLAN Notices, 1999, 34(10): 187-206.

[2] CHOI J D, GUPTA M, SREEDHAR V C, et al. Escape analysis for Java [J]. ACM SIGPLAN Notices, 1999, 34(10): 1-19.

[3] CHOI J D, GUPTA M, SERRANO M J, et al. Stack allocation and synchronization optimizations for java using escape analysis [J]. ACM Trans on Programming Languages and Systems, 2003, 25(6): 876-910.

[4] SALCIANU A, RINARD M. Pointer and escape analysis for multithreaded programs [C]// ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming. New York, USA: ACM, 2001: 12-23.

[5] SALAGNAC G, YOVINE S, GARBERVETSKY D. Fast escape analysis for region-based memory management [C]// 1st International Workshop on Abstract Interpretation for Object-Oriented Languages. Paris, France: ENTCS, 2005: 99-110.

[6] GAY D, AIKEN A. Language support for regions [C]// ACM SIGPLAN Conference on Program Language Design and Implementation. New York, UAS: ACM, 2001: 70-80.

[7] GUYER S Z, MCKINLEY K S, FRAMPTON D. Free-me: a static analysis for automatic individual object reclamation [C]// ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, USA: ACM, 2006: 364-375.

[8] CHEREM S, RUGINA R. Uniqueness inference for compile-time object deallocation [C]// 6th International Symposium on Memory Management. New York, USA: ACM, 2007: 117-128.

[9] LEE O, YI K. Experiments on the effectiveness of an automatic insertion of memory reuses into XSML-like programs [C]// The 3rd International Symposium on Memory Management. New York, USA: ACM, 2004: 97-108.

[10] VIVIEN F, RINARD M. Incrementalized pointer and escape analysis [C]// ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, USA: ACM, 2001: 35-46.

[11] Apache Software Foundation. Harmony-open source Java SE implementation [EB/OL]. [2009-09-12]. http://harmony. apache. org/index. html.

[12] 吴廷鹏，张昱，刘玉宇. 基于即时编译器辅助的并行垃圾收集器 [J]. 计算机工程，2009，35(10)：86-88.
WU Tingpeng, ZHANG Yu, LIU Yuyu. Parallel garbage collector based on just-in-time compiler assistance [J]. Computer Engineering, 2009, 35(10): 86-88.

[13] CAHOON B, MCKINLEY K S. Jolden benchmarks [EB/OL]. [2009-09-12]. ftp://ftp. cs. umass. edu/pub/osl/benchmarks/jolden. tar. gz.

（编辑　苗凌）