一种基于栈式分配的 JVM 垃圾收集算法①

陈贤闯　徐小良 (杭州电子科技大学 计算机应用技术 浙江 杭州 310018)

摘　要： 为了减少 Java 虚拟机垃圾收集器的开销，对 Java 栈帧进行改造使其支持对象的存储，在此基础上提出一种基于栈式内存分配的垃圾收集算法。算法首先在编译阶段通过对 Java 源代码进行分析确定栈式分配对象，并用扩展指令标识这些对象，程序运行时它们直接被分配到 Java 栈上，这些对象在程序离开其作用域时立即被释放，其它对象则分配到堆上，由垃圾收集器进行回收。实验结果表明，与传统基于堆式的垃圾收集算法相比，新算法内存分配和即时回收性能优，占用内存空间少，垃圾回收更及时，减轻了垃圾收集器的负担，提高了运行速度。

关键词： java 虚拟机;垃圾收集;栈式分配;java 栈;对象生命周期

A Stack-Allocation Based JVM Garbage Collection Algorithm

CHEN Xian-Chuang, XU Xiao-Liang

(Computer Application Technology, Hangzhou Dianzi University, Hangzhou 310018, China)

Abstract: In order to reduce the cost of java virtual machine garbage collector, we propose a stack-allocation based garbage collection algorithm. We improve the stack frame and make it support object storage, analyse and identify stack-allocation object when compiling the java source code, and mark the object with new byte command. Stack-allocation object will be allocated from java stack and released immediately when leaving its scope. Other object will be allocated from heap and released by garbage collector. Experimental results show that, compared with traditional garbage collector, the new algorithm improves the performance of allocating and releasing, reduces the burden of garbage collector and runs faster.

Keywords: java virtual machine; garbage collection; stack allocation; java stack; object lifetime

1　引言

垃圾收集，即自动内存管理，能够使程序员摆脱对显式内存分配和回收的依赖，从而减少了内存相关错误的发生。然而，围绕着垃圾收集存在许多争议，特别是关于它对程序执行效率的影响，历来是争论的热点。

国内外对垃圾收集算法进行了大量深入的研究。文献[1]提出了引用计数垃圾收集算法，这种算法通过为堆中的每一个对象保存一个计数来区分活动对象和垃圾对象，这个计数记录了对象的引用次数，计数为 0 时表示是可回收的垃圾对象。但是这种算法有三个缺点：维护引用计数不变式需要高昂的开销；每个单元需要额外的内存空间来存放引用计数值；无法回收环形的数据结构。文献[2]提出了标记-清除垃圾收集算法，文献[3]提出了节点复制垃圾收集算法，文献[4]提出了分代式垃圾收集算法，这些算法都是基于跟踪的垃圾收集算法。这种算法实际上追踪从根结点开始的引用图，在追踪中遇上的对象以某种方式打上标记，当追踪结束时，没有被打上标记的对象就被判定是不可触及的，可以被当作垃圾收集。这种算法也有三个缺点：在垃圾收集时用户程序的运行必须暂停；追踪式的垃圾收集机制要想有效运行，堆中需要一些净空间；使内存单元更容易产生碎片。上述这些算法中的所有垃圾回收对象都是分配在堆上的，文献[5]则提出

① 基金项目:国家重大专项(2009ZX01039-001-002-004)；浙江省重大科技专项(2007C11070)
收稿时间:2009-10-14;收到修改稿时间:2009-11-10

了通过逃逸分析将生命周期局限于方法内的对象分配到对象栈而非堆上，当程序离开该方法时，即时释放对象栈上的这些对象。该方法有效减轻了垃圾收集器的负担，但是也存在几个缺点：逃逸分析在程序运行时进行，需要额外的开销；需要维护一个对象栈；当程序离开方法时，即使没有对象分配到对象栈上也要进行判断。在目前的 Java 虚拟机规范中，字节码指令集和堆栈模型都不支持对象的栈式分配。

本文研究 Java 栈帧的改造使其支持对象的直接存储，并提出一种利用 Java 栈内存分配的 JVM 垃圾收集算法。该算法首先在编译阶段识别出可以进行栈式分配的 Java 对象，程序运行时将其分配到改造后的栈上，这些对象在程序离开其作用域时会立即被释放，其它非栈式分配的 Java 对象则分配到堆上，由垃圾回收器进行自动回收。与基于堆式内存分配的垃圾收集算法相比，该算法将有效减轻垃圾收集器的负担，优化 Java 的内存分配和即时回收等运行性能。

2 JVM堆栈模型

堆[6]：Java 程序在运行时创建的所有类实例或数组都放在同一个堆中。一个 Java 虚拟机实例中只存在一个堆空间，所有线程都是共享这个堆的。Java 虚拟机有一条在堆中分配新对象的指令，但没释放内存的指令。虚拟机自己负责决定如何以及何时释放不再被运行的程序引用的对象所占据的内存。程序本身不用去考虑何时需回收对象所占用的内存，虚拟机通常把这个任务交给垃圾收集器。垃圾收集器的主要工作就是自动回收不再被运行的程序引用的对象所占用的内存。此外，它也可能去移动那些还在使用的对象，以减少堆碎片。

Java 栈[6]：每当启动一个新线程时，Java 虚拟机都会为它分配一个 Java 栈。Java 栈以帧为单位保存线程的运行状态。虚拟机只会直接对 Java 栈执行两种操作：以帧为单位的压栈或出栈。线程调用一个 Java 方法时，虚拟机都会在该线程的 Java 栈中压入一个新帧。而这个新帧就成了当前帧。在执行这个方法时，它使用这个帧来存储参数、局部变量、中间运算结果等数据。

栈帧[6]由三部分组成：局部变量区，操作数栈和帧数据区。局部变量区和操作数栈的大小要视对应的方法而定，是按字长计算的，这些值在程序编译时被确定并放在 class 文件中。帧数据区的大小依赖于具体的实现。虚拟机调用一个 Java 方法时，它从对应类的类型信息中得到此方法的局部变量区和操作数栈的大小，据此分配栈帧内存，然后压入 Java 栈中。Java 栈帧的局部变量区被组织为一个以字长为单位、从 0 开始计数的数组，字节码指令通过从 0 开始的索引来使用其中的数据。类型为 int、float、reference 和 returnAddress 的值在数组中只占据一项，而类型为 long 和 double 的值在数组中却占据连续两项。

下面以图 1 中所示的一段 Java 程序为例，来描述 Java 程序运行时内存分配和栈帧的变化情况。如图 2 所示，直观描述了图 1 程序中 6 个执行点相对应的栈帧和堆内存空间的变化情况。

```
{
    Obj1 S1= new Obj1;①
    {
        Obj2 S2= new Obj2;②
    }③
    {
        Obj3 S3 = new Obj3;④
    }⑤
}⑥
```

图 1 Java 程序

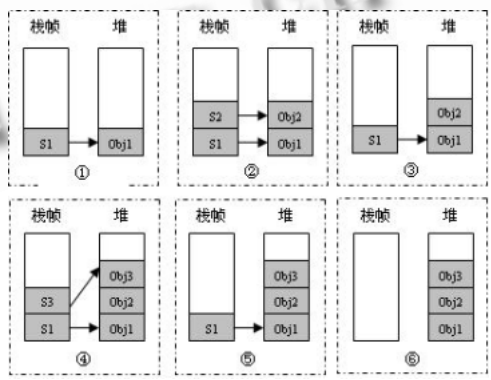

图 2 栈帧和堆内存空间的变化

Java 栈帧的局部变量区支持基本类型和引用类型数据的存储，但是不支持对象的直接存储。由图 2 所示，JVM 针对每一个新创建的 Java 对象都需要在堆上分配内存空间，同时在栈帧上为指向这个对象的引用变量分配内存空间。这种内存分配情况存在以下几个缺点：内存分配和对象访问效率都不高，内存占用空

间比较大，垃圾回收效率不高。针对图 1 程序中局部作用域对象的内存分配，一个比较好的解决方案是：在 Java 栈帧上直接为这些对象分配内存空间，这样内存分配效率将提高，占用内存空间将减少，另外内存空间回收也更及时。

3 栈式分配的JVM垃圾收集算法

为了高效经济地实现JVM 中的局部作用域对象的内存分配，提出了基于栈式内存分配的 JVM 垃圾收集算法。算法主要解决了 Java 栈式分配对象的识别、Java 栈帧的改造、以及内存分配和回收算法。

3.1 Java 栈式分配对象识别

C++可以先声明局部对象，然后使用局部对象。这些局部对象都被分配到运行栈上，局部对象的生命周期、分配指令和回收指令在编译阶段被确定，执行效率很高。Java 不支持局部对象语法，所有的对象都是通过 new 语句从堆上动态分配，分配的对象都不需要显示指令进行回收，由垃圾收集器进行回收。Java 虽然不支持局部对象语法，但大部分动态分配的对象都具有局部对象的特征，它们都具有很短的生命周期，对象序列具有先进后出的特点，这些对象都可以被分配到栈上，在本文中把这些对象称为 Java 栈式分配对象，或者局部作用域对象。

针对 Java 栈式分配对象的识别问题，可以在 Java 程序语言中引入新的语法来显式标识这些对象，程序员可以灵活的控制哪些对象分配到栈上，哪些对象需要分配到堆上。但是这种方法和 java 程序员不需要关心对象的内存空间从哪里分配相矛盾。本文把 Java 栈式分配对象的识别工作交给编译器，编译器通过对 Java 源代码进行分析自动识别出栈式分配对象，并用扩展指令 snew 代替堆式分配的 new 指令来区别。这种方法对程序员是透明的。目前，针对 Java 栈式分配对象的识别有很多方法，其中逃逸分析是一种主要方法。逃逸分析[7]是一种静态分析方法，用以确定数据的生存时间是否超过它的静态域的生存时间。如果对象 O 没有逃逸出方法 M，那么 O 必然也没有逃逸出创建它的线程 T，这样 O 就可以储存在 M 方法的栈帧中，而不必储存在堆空间中。

3.2 Java 栈帧改造

本节对 Java 栈帧进行改造，使其除了支持基本类型和预用类型数据的存储外，还能支持栈式分配对象的直接存储。以图 1 中的 Java 程序为例，程序在运行时，改进后的Java 栈帧内存空间变化情况如图 3 所示，不用显式的指令释放 Obj2 所占据的空间，Obj3 会复用这 Obj2 的内存空间，这些指令在编译时被确定。与图 2 所示传统 JVM 的内存分配相比，栈式分配对象没有涉及堆空间的分配，栈帧中没有为引用变量 S1、S2 和 S3 分配内存空间，对栈式分配对象的访问不通过引用进行，所有通过引用变量对栈式分配对象进行的访问都转化成直接访问。显然，改进后的 Java 栈帧优化了内存分配和对象访问。

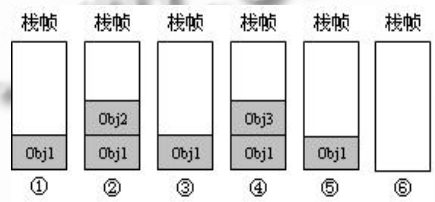

图 3　栈帧模型

3.3 内存分配与回收算法

在 Java 虚拟机解释执行时，若遇到 snew 指令则从当前栈帧上为创建的 Java 对象分配内存空间，栈帧的大小在编译时被确定，不需要判断是否会溢出；若遇到 new 指令则仍然从堆上分配内存空间，分配时判断堆上是否还有空闲内存，若没有则启动垃圾收集器进行收集。内存分配算法如图 4 所示：算法中的第三行，表示为栈式分配对象直接从栈帧顶分配空间，不用判断栈帧空间是否足够。算法中的第六行是针对堆式分配对象的垃圾收集过程。算法中的第八行，表示为创建对象从堆上分配内存空间。

```
(1)New(N) =
(2)    if N == snew
(3)        allocate from object stack
(4)    else if N == new
(5)        if free_pool is empty
(6)            garbage_collection()
(7)        endif
(8)        allocate from heap
(9)    endif
```

图 4　内存分配与回收算法

栈空间的回收以栈帧为单位，当线程离开一个方法时，Java 虚拟机就会弹出一个栈帧。栈帧空间的回

收在编译阶段被确定，不需要显式的回收指令，当一个对象的生命周期结束时，下一个栈式分配对象就会复用这个内存空间。堆空间的回收是由垃圾收集器负责，当堆没有足够的空闲内存时，虚拟机才会启动垃圾器进行回收。

4 实验

4.1 实验环境搭建

为了验证基于栈式分配的JVM 垃圾收集算法的有效性，开发实现了一个 Java 虚拟机的内存分配和垃圾收集的模拟系统。每个指针占用 4 个字节，每个模拟的对象的大小是实际大小加上 4 字节的对象头。对象头的高 24 位用来表示对象的实际大小，第 0 位在垃圾收集时用作标记位，第 1~6 不用，在本实验中为空。堆上的空闲空间用链表来组织，每个空闲块前面都会加上一个空闲链表头，这个链表头包括两个成员 size 和 next，size 占用 4 字节，表示该空闲块的大小(不包括 size)，next 占用 4 个字节，表示下一个空闲块。

本实验只涉及内存分配语句，不涉及其它的程序逻辑，代码由图 5 中的字节码文法生成。先按给定的语法树的深度随机产生 10 棵语法树(深度为 20 的完全二叉树)，每个结点代表一条语句。结点如图 6 所示，parent 指向父结点或为 NULL；lChild 指向左子树或为 NULL；rChild 指向右子树或为 NULL ；snew 表示该语句是否为栈式分配语句，其值为 true 或 false；MS 表示要分配内存的大小，其值随机产生(本实验取 20)；ref 指令表示该结点从堆上分配的内存空间的首地址。以图 1 中的 Java 程序为例，依次应用图 5 中①，③，①，③，②，②，①，③，②，②产生式即可得到。

① S→{LSS}
② S→ε
③ L→E
④ E→new
⑤ E→snew

图 5 字节码生成文法

parent	lchild	rchild	snew	MS	ref

图 6 树结点结构

代码的执行过程就是深度优先先序遍历语法树的过程：访问树结点时若 snew 为 true 则在栈帧上为其分配大小为 MS 的内存空间，若 snew 为 false 则在堆上为其分配大小为 MS 的内存空间，当堆上空闲内存不够时进行垃圾收集。

本实验中的垃圾收集器采用标记-清除算法[8]模拟实现，垃圾收集算法的标记过程就是从当前结点到根结点的遍历过程，若是 new 指令分配的对象则标记 ref 指令的对象。运行环境：戴尔 OptiPlex GX520 台式电脑，处理器：英特尔 Pentium(R) D 2.8GHz，内存：1GB(海力士 DDR2 533MHz)。

4.2 结果分析

在设定 JVM 堆值情况下进行本实验，可以得到 JVM 垃圾收集次数与栈式分配对象比例变化的关系。实验中取堆值分别为 130048b、260096b 和 520192b(其中 260096b 为 KVM[8]中堆大小的默认值)，由图 7 中的实验结果表明，对于任意给定的堆，垃圾收集次数随栈式分配对象比例成线性递减关系，栈式分配比例越大，垃圾收集次数越少。当栈式分配对象比例为 0 时，垃圾收集算法即为传统的标记-清除垃圾收集算法，垃圾收集次数最多。堆值越小，不但节省内存空间，而且垃圾收集次数随栈式分配对象比例增加而下降的速度越快，表明基于栈式分配的垃圾收集算法在堆小的情况下优势更明显。但并不是堆值越小越好，因为堆越小，垃圾收集频繁，影响系统的总体开销。在实际应用中应结合系统的运行时间开销、栈式分配对象比例等因素进行研究取最佳的堆值。

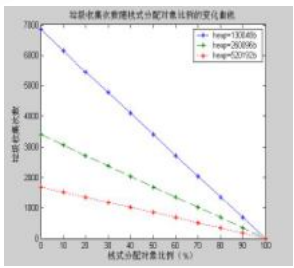

图 7 垃圾收集次数比较

图 8 中的 3 条曲线表示堆值分别为 130048b、260096b 和 520192b 的实验运行时间与栈式分配对象比例的关系。实验结果表明，对于任意给定的堆，运行时间随栈式分配对象比例成递减关系，栈式分配对象比例越大，运行时间越少，即表明内存分配更快，性能

更优。当栈式分配对象所占比例为 0 时，即为垃圾收集为传统的标记-清除垃圾收集算法，运行时间最长。

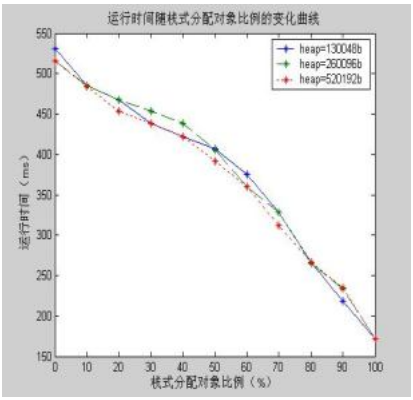

图 8　运行时间比较

图 8 中三条曲线出现交叉，说明不存在某条曲线绝对好于另一条曲线，但是在一个确定的区间范围内，性能上可能存在优劣。在实际应用中，客户端程序和服务端程序的对象生命周期是不同的，不同的逃逸分析方法识别出的栈式分配对象的数量也是不同的，堆的取值也可能不一样，可以结合本文来研究确定最佳的内存分配与垃圾回收方案。

5　结论

本文介绍了一种基于栈式分配策略的 JVM 垃圾收集算法。该算法在编译阶段通过对源代码进行分析确定栈式分配对象，并用扩展指令表示这些对象的栈式内存分配；改进了 Java 栈帧使其支持栈式分配对象的存储；在 JVM 运行时，栈式分配对象被直接分配到改进过的栈上，当程序离开其作用域时，它们所占用的空间立即得到释放，其它对象则被分配到堆上，由垃圾回收器进行回收。实验结果表明，与传统的垃圾收集算法相比，该算法优化了 JVM 的内存分配和回收，减轻了垃圾收集器的负担，提高了运行性能。

参考文献

1 Collins GE. A method for overlapping and era sure of lists. Communications of the ACM, 1960,3(12):655－657.

2 McCarthy J. Recursive functions of symbolic expressions and their computation by machine. Communications of the ACM,1960,3(4):184－195.

3 Fenichel R, Yochelson JC. A Lisp gar bage collector for virtual memory computer systems. Communications of the ACM, 1969,12(11):611－612.

4 Lieberman H, Hewitt CB. A real-time garbage collector based on the lifetimes of objects. Communications of the ACM, 1983,26(6):419－29.

5 Corry E. Optimistic stack allocation for java-like languages. Erez Petrank. Proc. of the 5th International Symposium on Memory Management.Ottaw, Canada: ACM, 2006,162－173.

6 文纳斯.深入 Java 虚拟机.北京:机械工业出版社, 2003.102－103.

7 Blanchet B. Escape analysis for javaTM: Theory and practice. ACM Transactions on Programming Languages and System, 2003,25(6):713－775.

8 探矽工作室.深入嵌入式 Java 虚拟机.北京:中国铁道出版社, 2003.

9 琼斯,林斯.垃圾收集.北京:人民邮电出版社, 2004.