第35卷 第10期 计 算 机 工 程 2009年5月
Vol.35 No.10 Computer Engineering May 2009
·软件技术与数据库· 文章编号：1000—3428(2009)10—0086—03 文献标识码：A 中图分类号：TP311.5

基于即时编译器辅助的并行垃圾收集器

吴廷鹏[1,2]，张 昱[1,2]，刘玉宇[1]
(1. 中国科学技术大学计算机科学技术系，合肥 230027；2. 中国科学院计算机系统结构重点实验室，北京 100080)

摘 要：设计一种基于即时编译器辅助的并行垃圾收集器，在编译阶段利用即时编译器安插显式内存回收指令释放无用对象空间，结合自动和显式内存管理的优点，减轻程序员和垃圾收集器的负担，同时提高内存空间的利用率，支持显式对象回收操作，且可以有效重用这些对象空间，从而提高Java应用程序的执行效率，实验结果表明，该设计方案是可行的。

关键词：即时编译器；垃圾收集器；显式内存管理

Parallel Garbage Collector Based on Just-in-time Compiler Assistance

WU Ting-peng[1,2], ZHANG Yu[1,2], LIU Yu-yu[1]
(1. Dept. of Computer Science & Technology, University of Science & Technology of China, Hefei 230027;
2. Key Lab of Computer System & Architecture, Chinese Academy of Sciences, Beijing 100080)

【Abstract】 A parallel Garbage Collector(GC) based on Just-In-Time(JIT) compiler assistance is presented, which can free objects explicitly by inserting the "free" instructions in compilation time. Combined with the merits of automatic and explicit memory management, the burden of both programmer and GC is reduced, and the utilization of memory space is promoted. It supports explicit object operation, and reuses these object spaces, which improves the operating efficiency of Java applications. Experimental results show this design scheme is feasible.

【Key words】 Just-In-Time(JIT) compiler; Garbage Collector(GC); explicit memory management

1 概述

像Java这类面向对象的程序设计语言采用垃圾收集器(Garbage Collector, GC)动态管理对象的分配与回收。垃圾收集器不仅使分配和回收具有灵活性，还简化了程序员的编程，但其本身也要消耗资源，会带来额外的系统开销，是影响Java虚拟机性能的重要因素之一。为提高垃圾收集器自动管理内存的性能，文献[1-2]提出即时编译器(Just-In-Time compiler, JIT compiler)辅助的内存管理技术，即编译阶段在代码中安插显式的内存回收指令，从而减轻垃圾收集器自动回收的负担，现有的研究主要集中在分析技术上，并没有研究如何更好地在垃圾收集器中支持对象的显式回收操作。

本文基于现有的垃圾收集算法，研究垃圾收集器对显式回收操作的支持，给出一个支持对象显式回收操作的并行垃圾收集器的实现方案。

2 垃圾收集器的实现框架

在开源的Java SE平台Apache Harmony[3]上实现即时编译器辅助的垃圾收集，主要涉及到即时编译器和垃圾收集器2个模块，另外，虚拟机的核心模块VMCore负责协调这2个模块的交互。

在即时编译器辅助的垃圾收集框架中，即时编译器的主要工作是：(1)在高级中间表示(HIR)和低级中间表示(LIR)中增加支持对象显式回收的指令；(2)添加一个在HIR上的优化遍，它将分析出程序中的对象生命期信息，选择合适的程序点安插回收指令来释放已经死亡的对象空间；(3)修改代码选择器，使之支持扩展HIR到扩展LIR的变换；(4)修改代码发射器，将显式回收指令发射为运行时的指令调用。其中，对象生命期分析算法是以逃逸分析为基础的。

VMCore部分需要添加新的运行时支持，它能将显式回收指令生成的指令调用映射到垃圾收集器中提供的对象显式回收接口free()。

垃圾收集器部分的主要工作是提供对对象显式回收接口free()的实现，改进对象分配接口allocate()的实现以支持对显式回收空间的及时重用，从而提高内存空间的利用率，减少垃圾收集的调用频率。

整个框架如图1所示。

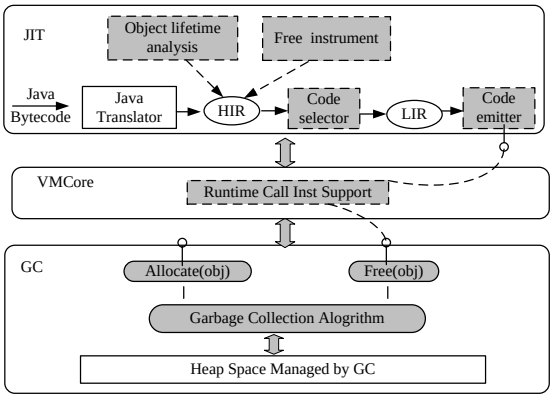

图1 即时编译辅助的垃圾收集框架

本文主要阐述垃圾收集器侧的设计与实现，这项工作是

基金项目：中国科学院计算机系统结构重点实验室开放课题基金资助项目；Intel公司研究基金资助项目
作者简介：吴廷鹏(1983－)，男，硕士研究生，主研方向：Java内存管理；张 昱，副教授、博士；刘玉宇，硕士研究生
收稿日期：2008-11-20 **E-mail：**yuzhang@ustc.edu.cn

以Harmony的GC-MS垃圾收集器(一种并行的标记-清扫算法的实现)为基础的。

3 即时编译器辅助的垃圾收集器实现

3.1 Harmony 中的 GC-MS 算法

GC-MS 是种并行标记-清扫算法的实现，它采用高效的多线程分配，利用不同的数据结构组织管理不同大小的对象在堆中的分配。其中，小于 1 KB 的对象(简称小对象)分配使用块(Block)结构；大于等于 1 KB 的对象(简称大对象)分配使用空闲空间池(Free Area Pool)，如图 2 所示。

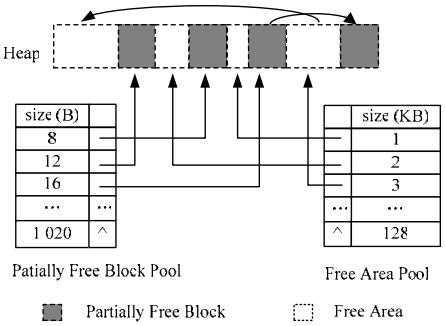

图 2 GC-MS 中的堆空间管理

块结构包含块头和数据区 2 个部分，块头包含该块中的相关分配信息，数据区用于分配对象。块的大小是固定的，当前设为 32 KB，每个块只能分配特定大小的对象。GC-MS 引入块结构的目的是为解决多线程并行分配带来的同步问题。当前在虚拟机上运行的并行应用程序的每个线程在 GC-MS 中对应有一个分配线程(Mutator)，每个 Mutator 在本地持有若干个块，分别用于不同大小的小对象分配。当一个 Mutator 需要分配一个大小为 size(按 4 Byte 对齐)的小对象时，直接在该 size 对应的本地块中分配，由于一个块只被一个 Mutator 专有，因此分配无须同步操作。

空闲空间池管理所有大对象的分配，它包含 128 条链表，依次管理大小为 1 KB,2 KB,…,127 KB 以及大小大于等于 128 KB 的空闲空间，每条链上的空闲空间大小相同。当一个 Mutator 需要分配一个大小为 size(按 1 KB 对齐)的大对象时，到该 size 在空闲空间池中对应的链表上获得一个空闲空间用于对象的分配。由于多个 Mutator 共用空闲空间池，因此大对象的分配需要同步操作，但由于大对象的数目较少，同步操作带来的性能开销不是很大，另外，空闲空间池还用来为 Mutator 提供块结构空间，这相当于分配给 Mutator 一个大小为 32 KB 的大对象。

当堆中无可用空间时，GC-MS 需要暂停应用程序的执行，启动垃圾收集来回收堆中已经死亡的对象空间，在收集完成后，对于每个块结构，如果块中无活对象，则将块空间归还给空闲空间池(Free Space Pool)；如果块中还有活对象，则将该块插入到部分空闲空间池(Partially Free Block Pool)，做法是根据块中已分配的对象大小插入到该大小在部分空闲空间池里对应的链表上。

垃圾收集完成后，应用程序继续执行。当分配线程 Mutator 需要分配一个大小为 size 的对象时，如果该 size 对应的本地块已经用完，则 Mutator 先到部分空闲块池中获得一个该 size 对应的部分空闲块到本地用于该 size 的对象的分配。如果部分空闲空间池中无该 size 的部分空闲块，则到空闲空间池中申请一个 32 KB 大小的空闲空间作为新的块，用于该 size 的对象的分配，如果堆中没有空闲空间，则需要再次触发垃圾收集来回收死亡对象的空间。

3.2 对象显式回收操作的实现

为支持对象的显式回收操作，垃圾收集器需要提供新的接口 free()来支持对象空间的显式回收，因为应用程序中的线程和垃圾收集器上的分配线程 Mutator 相对应，所以应用程序上的对象回收操作也是通过 Mutator 来完成。

对于大对象的显式回收，可以直接将其插入到空闲空间池中，当后面再需要分配相同大小的对象时，该对象空间就会被重新分配出去，该操作需要一个同步操作，因为可能出现多个 Mutator 同时在空闲空间池中的同一条链表上操作，但由于这部分的对象数目较少，因此同步带来的性能开销可以忽略。

支持对象显式回收操作的难点在于对小于 1 KB 的对象的分配，这里引入新的数据结构 Obj_Space, Obj_Space_Pool 来支持对象的显式回收操作，如图 3 所示。

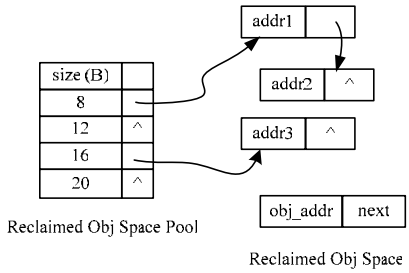

图 3 显式回收的对象的空间管理

Obj_Space 包含 2 个域，addr 用来记录显式回收的对象地址，next 用来指向下一个 Obj_Space 元素。具有相同大小的对象空间被放在同一条链表上，各个大小的链表以数组结构组织在一起构成 Obj_Space_Pool，同时每个 Mutator 在本地持有一个 Obj_Space_Pool 用于管理其对应的应用程序线程显式回收的对象空间。当 Mutator 接收到一个对象显式回收的调用时，只需要根据对象的大小将它插入到本地的 Obj_Space_Pool 中，由于每个 Mutator 有自己的 Obj_Space_Pool，因此不需要同步操作。另外，对于被显式回收的对象空间，可以利用它自身来构造 Obj_Space，这样就不用开辟新的空间构建 Obj_Space，所以，该策略不会带来很大的空间开销。

下面给出对象显式回收方法 free()的伪代码：

```
void free(void* addr, unsigned int size){
1: if (size < 1KB) {
2: Obj_Space* space = (Obj_Space*)addr;
3: space->addr = addr;
4: add_obj_into_local_obj_space_pool(space, size);}
5: else {
6: Free_Space* space = (Free_Space*)addr;
7: add_obj_into_free_space_pool (space, size);}
}
```

其中，第 2 步~第 4 步完成小对象的回收，即第 2 步利用被回收的对象空间自身来构造 Obj_Space，第 3 步设置其 addr 域为自身的地址，第 4 步将该对象对应的 Obj_Space 插入到本地的 Obj_Space_Pool 中。第 6 步~第 7 步完成大对象的回收，直接将其插入到图 2 的空闲空间池中。

为尽可能地重用显式回收的对象空间，这里修改了对象的分配策略 allocate()。当分配线程 Mutator 需要分配一个大小为 size 的对象时，先在其本地的 Obj_Pool_Space 中查看是

否有该 size 的对象空间，这样显式回收的对象空间就能够尽快地被分配出去，另外，由于在每次垃圾收集中会统一回收所有的空闲空间，因此垃圾收集过程结束需要清空每个 Mutator 本地的 Free_Space_Pool。综上所述，该算法以较小的时间和空间开销支持了对象的显式回收操作，同时这些回收的空间能够尽快地被重新分配出去，提高了内存空间的利用率。

4 实验结果及分析

笔者在开源的 Java SE 平台 Apache Harmony 上实现即时编译器辅助的垃圾收集器 JIT_GC-MS，并做了相关的性能测试。实验平台的操作系统是 Windows XP，CPU 为 AMDX2，主频为 2.0 GHz，内存为 1 GB。测试用例为 Jolden 中的基准程序 Health 和 TSP。

为比较原有的 GC-MS 和本文 JIT_GC-MS 在性能上的差异，考核在不同堆大小情况下运行 Health 和 TSP 时 GC 的停顿时间，结果如图 4、图 5 所示。

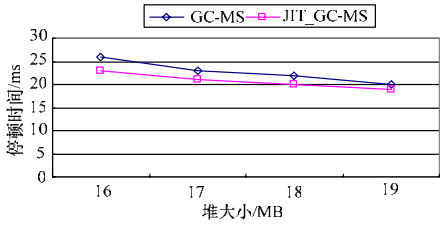

图 4 Health 的性能对比

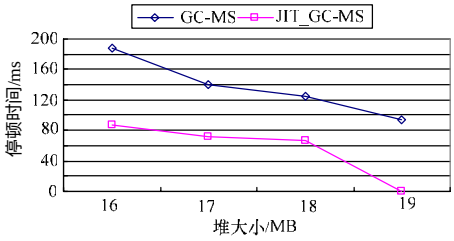

图 5 TSP 的性能对比

从图 5、图 6 可以看出，对于 Health 和 TSP，JIT_GC-MS 在停顿时间上都少于 GC-MS，提高了应用程序的执行效率。

随着堆的大小变大，性能提升的幅度变小，这是由于可以显式回收的对象数目固定，而当堆变大时，同样大小的重用空间带来的性能提升相对减少。同时，统计了由 JIT_GC-MS 显式回收的对象数量及重用情况，结果如表 1 所示。

表 1 显式释放的对象及被重用的对象空间数目

基准程序	总计回收的对象数目	堆大小/MB	被重用的对象空间的总数目	被重用对象空间的总大小/MB
Health	100 004	16	100 003	1
		17	100 002	1
		18	100 002	1
		19	100 003	1
TSP	524 287	16	524 284	13
		17	524 284	13
		18	524 284	13
		19	524 285	13

从表 1 可以看出，JIT_GC-MS 能够及时重用几乎所有的、显式回收的对象空间，可以显著提高内存空间的利用率。

5 结束语

本文提出一种基于即时编译器辅助的并行垃圾收集器的实现，该收集器支持显式的对象回收操作，且可以及时有效地重用这些对象空间。实验结果表明，使用即时编译器辅助的垃圾收集器，提高了内存空间的利用率，从而提高 Java 程序的执行效率。下一步工作将改进即时编译器中分析算法的精确性，使之能够分析出更多可以回收的对象，从而获得更好的性能提升。

参考文献

[1] Samuel Z. A Static Analysis for Automatic Individual Object Reclamation[C]//Proc. of the ACM Conf. on Programming Language Design and Implementation. Ottawa, Canada: [s. n.], 2006.

[2] Sigmund C. Inference for Compile-time Object Deallocation [C]// Proc. of the 6th Int'l Symp. on Memory Management. Montreal, Canada: [s. n.], 2007.

[3] Brendon C. Apache Software Foundation[Z]. (2005-07-12). http://harmony.apache.org/index.html.

编辑 陈 文

(上接第 85 页)

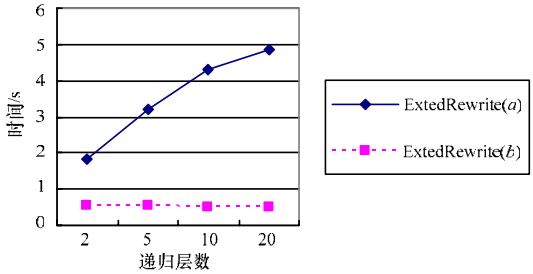

图 6 基于递归层数的 *Q2* 查询结果

6.4 实验结果分析

从实验数据可以看出，随着文档尺寸的增大，重写一个查询需要的时间变长，递归安全视图的递归层数越多，重写查询需要的时间越长。处理递归安全视图的查询重写算法可以应用到非递归视图，且其查询时间与应用处理非递归视图的查询重写算法相差不多。

参考文献

[1] 瞿裕忠, 张剑锋, 陈 峥, 等. XML 语言及相关技术综述[J]. 计算机工程, 2000, 26(12): 4-6.

[2] Miklau G, Suciu D. Containment and Equivalence for an XPath Fragment[C]//Proc. of Conf. on Principles of Database Systems. Madison, Wisconsin, USA: [s. n.], 2002: 65-76.

[3] Balmin A, Ozcan F, Beyer K, et al. A Framework for Using Materialized XPath Views in XML Query Processing[C]//Proc. of Conf. on Very Large Data Bases. Toronto, Canada: [s. n.], 2004: 60-71.

[4] Xu Wanghong, Ozsoyoglu Z M. Rewriting XPath Queries Using Materialized Views[C]//Proc. of Conf. on Very Large Databases. Trondheim, Norway: [s. n.], 2005: 121-132.

[5] Fan Wenfei, Chan Chee-Yong, Garofalakis M. Secure XML Querying with Security Views[C]//Proc. of ACM SIGMOD International Conference on Management of Data. Paris, France: [s. n.], 2004: 13-18.

[6] 王竞原, 胡运发, 葛家翔. XPath 中的文本查询研究[J]. 计算机工程, 2007, 33(11): 70-72.

编辑 陈 晖