基于类型的运行时环境存储管理算法*

张武生+, 杨广文, 郑纬民

(清华大学 计算机科学与技术系,北京 100084)

A Class-Based Garbage Collection Algorithm for the Runtime Environments

ZHANG Wu-Sheng+, YANG Guang-Wen, ZHENG Wei-Min

(Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

+ Corresponding author: Phn: +86-10-62796945, E-mail: zws@tsinghua.edu.cn, http://www.tsinghua.edu.cn

Received 2003-11-25; Accepted 2005-02-03

Zhang WS, Yang GW, Zheng WM. A class-based garbage collection algorithm for the runtime environments. ***Journal of Software*****, 2005,16(8):1506−1512.** DOI: 10.1360/jos161506

Abstract: The class-based garbage collection approach is to improve the efficiency of the runtime environment for server applications. The class-based garbage collection algorithm is intended to reduce the overhead of dynamic object creation through an object-reuse technology, and the unique techniques of thread cache and lease protocol etc. can further reduce the pause time of the server applications and provide them better smoothness and good memory usage. The virtual machines run on top of the class-based garbage collector can be implemented to support fine grade parallelism between the mutator threads and the garbage collection threads.

Key words: virtrual machine; runtime environment; garbage collection; thread cache; class-based

摘 要: 基于类型进行分类管理堆空间的垃圾回收算法通过废弃对象复用来降低运行时环境创建对象所需时间开销,同时还通过线程缓存、租赁等技术进一步增强运行时系统的存储管理效率.运行实验表明,该算法能够在回收线程和工作线程之间实现细粒度的并行性并缩短对象申请和回收时间,进而能够减少工作线程的停顿现象,增加服务器应用的平滑性以及提高堆空间的使用效率.

关键词: 虚拟机;运行时环境;垃圾回收算法;线程缓存;对象分类

中图法分类号: TP316 文献标识码: A

GC(garbage collector,垃圾回收器)是 Java 虚拟机(JVM)的重要组成部分之一.它的性能对运行时环境(runtime environment)的效率有着至关重要的影响,各种优化算法都需要解决并发性(concurrency),防止程序停顿(pause)等现象的发生.

* Supported by the National Natural Science Foundation of China under Grant Nos.60373004, 60373005, 90412006 (国家自然科学基金); the National Grand Fundamental Research 973 Program of China under Grant No.2004CB318000 (国家重点基础研究发展计划(973))

作者简介: 张武生(1972−),男,内蒙古赤峰人,博士,主要研究领域为高性能计算,分布式系统,P2P;杨广文(1967−),男,博士,教授,CCF 高级会员,主要研究领域为网格计算,并行与分布处理,算法设计与分析;郑纬民(1941−),男,教授,博士生导师,CCF 高级会员,主要研究领域为并行处理,分布式系统.

(1) 虚拟机体系结构

基于中间代码的应用程序多采用动态编译以解释方式执行,由虚拟机完成字节码的解释和内存管理.虚拟机是一个抽象的体系结构,一般以线程为单位实现对应用程序的控制,其基本组成部分如图 1 所示.如果应用程序某处使用 new 指令创建新的对象,则执行线程调用堆管理器来获取所需空间.堆管理器(garbage collector,简称GC)负责申请空闲空间,并管理全局堆.对于需要频繁创建及撤销对象的应用程序而言,堆空间的使用效率是影响性能的重要因素之一[1−3].

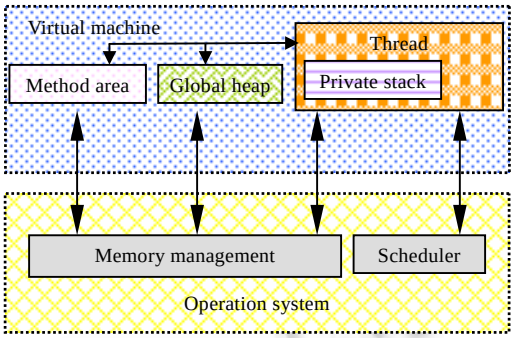

Fig.1 Example architecture of the virtural machine
图 1 虚拟机体系结构示例

(2) 垃圾回收算法

自 20 世纪 80 年代开始,就有许多致力于提高 GC 性能的研究,当时 GC 的应用主要集中在 Lisp 及其他一些函数式语言上.近年来,自动内存管理技术则随着 Java 语言的流行而受到越来越多的重视,应用的成功使人们普遍认识到它应该成为程序设计语言所具备的主要功能之一.为提高 GC 的工作效率,人们从各种角度出发考虑其优化措施,设计了一系列回收算法.这些算法按照工作方式可划分为两大类:基于引用计数的工作方式和基于扫描的工作方式[4−7].

引用计数工作方式的基本原理是对应用程序申请的内存空间设置一个计数域保存其被引用的次数,新申请的空间计数设置为 1,以后每被引用一次计数域加 1,取消一个引用后计数域则减 1.当引用计数变为 0 时,则该内存就自动成为 GC 操作的目标.采取引用计数方式的例子有 UNIX 下的应用程序,如 awk,PERL,早期版本的 Smalltalk 等.该方式最大的缺点在于它的不完备性,因为它难以处理循环引用情况.此外,跟踪引用所需要的时间和空间开销也比较大.

扫描工作方式的基本原理是遍历由对象之间的相互引用所形成的图,在遍历过程中识别有用内存和无用内存并进行标记从而确定应该回收的“垃圾”.其中在实际应用中显示较好效果的有 Marksweep,Generation-Copying,Train Algorithm 等算法.

1 基于类型分类的堆管理算法

根据网络服务系统的特点,本文结合虚拟机体系结构及运行机制提出了一种基于类型来划分空间的堆管理算法(Classbase-GC),该算法利用对象的类型实现堆空间的分类管理,其目的是:

- 随着应用程序长时间运行使得堆空间布局趋于稳定从而可以大大减少新创建对象时扫描可用空闲块的时间,提高响应速度.
- 减少回收工作中对象拷贝工作量,减缓因 GC 执行所造成的应用程序长时间停顿.
- 提高堆管理器与工作线程间工作的并行性.
- 提高堆空间的利用率.

定义(类型空间,**CSp**). 系指由虚拟机维护的一组数据结构,用以记录已加载类的运行时元数据信息.运行中创建的任何对象都按照其类型分别归属各自的类空间.

1.1 运行时刻二进制模型

在面向对象系统中,所有实体都被抽象成对象,每个对象都分别属于某种类型(即类,class),程序的运行时刻系统负责加载并在内存中建立这些类型的静态信息,如同现行许多基于这一概念工作的面向对象系统,如 C++,Java 一样,该概念模型也是本节所讨论的依类型分类的 GC 算法的工作基础.在本文的后面叙述中将把代表对象类型的类(class)的运行时刻数据结构称为元类(MetaClass).

1.1.1 基本数据结构

图 2 为运行时刻表示对象模型的几个数据结构.图 2(a)给出 *MetaClass* 数据结构(相当于方法表),它主要保

存编译后的二进制代码以及域成员等信息.*MetaClass* 数据结构是运行时刻创建对象的模板.

```
MetaClass {
    MethodTable     *methods;
    FieldTable      *members;
    size_t          object_size;
    //other runtime information
    //…
}
```

(a) Structure of meta class (a) 元类结构

```
ObjHdr {
    MetaClass       *proto;
    MethodTable     *methods;
    time_t          creation_time;

    refere_obj {
        int         offset;
        obj_hdr     *ref_obj;
    } *ref_set;
    Obj_Hdr *next_obj_of_this_type;
}
```

(b) Object header (b) 对象头

```
classtable_entry {
    ObjHdr *class;
    ObjHdr *obj_free;
    ObjHdr *obj_end;
    Lock   *lock;
    int    use_count;
}
```

(c) Global class table (c) 全局元类表

Fig.2 Structures of the objects
图 2 对象模型的数据结构

除元数据结构以外,图 2(b)给出了该结构的细节.所有对象(包括元类 *MetaClass* 结构本身)以 *ObjHdr* 结构开始.其中的 *proto* 保存一个指向该对象所属元类数据结构的指针,GC 算法可以通过该指针来访问对象的元数据信息以及同属该类型的其他对象的信息.GC 线程在运行过程中将使用 *create_time* 时间戳决定相应对象是否可以在本次运行中处理或必须留待下次运行.由于 Classbase-GC 把工作过程分为标记和回收两个完全独立的阶段,因而在回收阶段必须能够区分已被标记阶段处理过的对象和自标记开始以后系统新创建的对象.以该时间戳为标记,就能够将两个不同时期创建的对象分开来处理.

ref_set 域是一个由在 GC 运行时刻建立的表结构,记录对象引用信息.其中的 offset 指明父对象中引用该对象的位置,ref_obj 则保存指向父对象的指针.标记结束后通过该结构所记录的对象之间引用关系就可以获得系统在当时时刻的一个完整的对象引用图.所有属于同一类型的对象通过 next_obj_of_this_type 域构成链接表,并称每个这样的链表为堆空间的一个划分,同类型对象构成的链表所占用的空间称为一个子类空间.

1.1.2 基本数据结构

Classbase-GC 使用一个全局表来保存所有 *MetaClass* 结构构成全局类型空间,全局类表的元素定义为如图 2(c)所示的 classtable_entry.

obj_free 和 *obj_end* 则总是分别指向子空间中第 1 个可用对象和子空间链表的表尾.借助二者,GC 仅通过一次寻址即可获得所取对象的存储空间,从而使得应用程序创建新的对象时间为与堆空间大小以及其使用状态无关的常数.

use_count 是一个统计量,Classbase-GC 用它来记录各种类型对象创建的频率.根据该频率算法可以预测各个子空间平均需要量,进而在整理堆空间时对某些子空间做相应的扩张或收缩调整,通过这种做法可以减少碎片的发生,在较长运行期内维持一种稳定的堆空间布局.

lock 域是维护一致性的锁,为了保证 *obj_free*,*obj_end*,*use_count* 等值被安全修改,每个工作线程在创建新的对象之前必须通过该锁获取相应子空间的控制权.该机制在保证安全性的同时极大地增加了线程之间的并行性.设置 *glb_lock* 来对各个线程的更新操作进行同步,每个线程在向全局类表之中添加新的类型之前必须首先获取 *glb_lock*.

此外,还以全局变量 *mark_time* 作为记录标记操作开始的时间戳,以便与 *ObjHdr* 中的 *create_time* 相比较来确定各个对象应归属哪一轮操作处理.全局变量 *obj_hold* 和 *mem_hold* 分别用来统计系统运行时刻所有线程所创建对象的总和以及占用空间的总和.

2 算法实现

2.1 基本操作

2.1.1 添加子空间

该操作原型 *add_to_class_table*(*ObjHdr *new_class*).线程调用该原子操作将子空间加入到全局类表中.

2.1.2 创建对象

操作的原型为 *gc_alloc*(*ObjHdr *metaClass*).GC 调用该操作为线程申请一个由参数 *metaClass* 指定类型的对象大小的空间,负责初始化其 *ObjHdr*,并将对象添加到相应子空间.

2.1.3 释放对象

释放对象操作的原型定义为 *gc_free*(*obj_hdr *obj*),是上述 *gc_alloc* 的逆操作,采取两种释放策略——浅释放和深释放.浅释放操作把被释放的仍旧保持在它所属的子空间中,操作时只更新相应子空间的 *obj_free* 和 *obj_end* 指针,实质上并不改变子空间大小.在引入线程缓存机制后,该操作将把释放的对象直接保存在线程缓存中.深释放操作则将空闲的对象彻底交还给操作系统,深释放完成后,相应的子空间将被紧缩.针对某子空间执行深/浅释放,由 GC 算法根据运行时刻对象创建情况所收集的统计信息决定.

2.1.4 移动对象

移动对象操作用于减少空间碎片以优化空间布局,原型定义为 *move_obj*(*obj_hdr *obj*).通过拷贝对象的办法对堆空间进行重新布局,以便使空闲空间尽量连续.

2.1.5 锁操作

这类操作用于保证多线程并发访问互斥资源时的安全性.锁操作总是成对出现,其原型定义为 *gc_lock*(*Lock *lock*),*gc_unlock*(*Lock *lock*).这对操作为 GC 算法提供了对被访问资源施加保护的机制.由于锁的使用需要底层系统调用,因此算法设计需要在增加并行性和减少需保护的资源之间做出折衷选择.Classbase-GC 算法通过把相同类型的对象合成一组,使得锁操作每次仅针对同一组对象进行,从而各个线程可以同时操作不同的组,这既增加了并行性又避免了锁操作的频繁使用.同时,线程缓存机制可以使得不同线程对相同子空间进行并行操作,也可进一步减少锁的使用次数.

2.1.6 租赁操作

租赁操作由 *gc_alloc* 调用,它提供了一种跨越子空间的存储区间共享机制,主要用于缓解应用程序因堆管理器运行所造成的停顿现象,原型为 *lease*(*ObjHdr *MetaClass*).当某个进程申请创建新对象过程中发生线程缓存、子空间以及系统缺失时,可从与当前申请对象大小相似类型的子空间中借取对象空间到当前空间中来满足线程运行所需.借取空闲开销较全面运行 GC 要小得多,并对各个子空间所占空间的大小进行平衡,以利于稳定运行.

2.2 实现技术

2.2.1 线程缓存

某线程在运行过程中调用 new 指令申请创建新的对象时,堆管理器使用 *gc_alloc* 操作为其定位所需的存储空间.这个过程中,发出请求的线程必须首先获取其将要创建类型的对象所属子空间的互斥锁.这不但削弱了线程之间的并行性,而且增加了系统的锁操作频率.为此,在 Classbase-GC 中设计了线程缓存,缓存对于每个线程而言都是私有的,因此在申请创建新的对象时,如果缓存中有足够的空间就不需要与其他线程进行同步来访问相应的子空间,它是提高工作线程并行性的重要手段.

2.2.2 合并子空间

在定义良好的面向系统中出于建模的需要往往会针对抽象的概念定义许多抽象的类型,由于抽象类不能实例化,因此对抽象子空间可做特殊处理而不必维护专门的访问锁.在实现时设计了一些特殊的子空间组,如“System”组管理所有的单一实例对象,“Abstract”组统一管理所有的抽象类,“ZeroSize”组则综合管理了所有大小为 0 的对象.这些组在运行时刻可以排除在扫描标记和整理操作之外.

2.2.3 租赁协议

该协议的实现有一个非常重要的特征——从借取者角度来看,被借的对象空间仅属于自身类型,即屏蔽了其原来所属子空间信息.

为使租赁协议足够精简,鉴于其主要操作是搜索合适的子空间,通过对全局类表按照 *object_size* 进行排序来加快搜索速度,在搜索过程中主要测试两个条件:①*object_size* ≥请求对象大小;②子空间的 *obj_free* ≠ *obj_end*.采用该协议的主要缺点是,无法保证每次搜索全局类表时都能够找到满足上述条件的子空间,当这一过程失败时仍然需要重新启动强制 GC,此时会造成额外开销.

2.2.4 运行时机

Classbase-GC 算法可选择懒惰型、周期型和不确定型等 3 种运行方式.懒惰型 GC 总是尽量保持休眠状态,直到它被某些工作线程唤醒.虽然实现简单,堆管理器在程序运行期间所占开销也比较小,但应用程序会因深释放操作而出现显著停顿.周期型实现方式通过一个预定义的计时器来控制算法运行.由于不同类型的应用程序其使用内存的模式各不相同,因而很难确定有效的计时周期,运行过程中往往仍需要工作线程与 GC 线程进行同步来等待深释放的空间.在不确定型实现方式中,每次 *gc_alloc* 被调用后不论成功与否都执行一个测试操作,当预定义的条件满足时就唤醒堆管理器线程运行.考虑到效率问题,需尽量减小检查开销.

3 测试与分析

测试环境采用 4CPU 的 XEON700MHzSMP,1MB 片内 Cache,主存储空间 1GB.GC 模块和工作线程之间构成一个排队系统,为观察其处理创建对象请求的服务时间、等待时间、滞留时间、等待队长等参数的分布特征,测试过程共创建对象数目约为 1.85×10^7,并以创建对象所需执行的指令数量作为计时单位.表 1 分别给出堆空间大小为 512KB 和 1024KB 条件下各参数的对比.

Tabel 1 Comparation of the MarkSweep and ClassBase algorithm

表 **1** MarkSweep 和 ClassBase 算法测试数据比较

	Heap size (Kbytes)			
	512		1 024	
	MarkSweep	ClassBase	MarkSweep	ClassBase
Maximum serve time	30 237	127 693	52 479	156 808
Average serve time	3 842.828 6	4.230 578	7 725.943 4	3.061 058
Maximum wait time	21 880	55 052	44 387	6
Average wait time	1.871 459	0.002 968	7.376 219	1.509×10^{-6}
Maximum stay time	30 237	127 693	59 197	156 808
Average stay time	3 844.699 7	4.233 547	7 733.319	3.061 059
Maximum queue length	2	1	2	1
Average queue length	$7.187\ 2\times10^{-4}$	7.544×10-7	0.001 427 29	$7.005\ 1\times10^{-7}$
Ttidy times	2 319	422	1 173	101
Empty rate	---	0.045 3	---	0.027 981
Reuse rate (%)	---	97.984 2	---	99.258 3

图 3 显示的是排队系统各参数指标的比较,横坐标为堆空间大小(单位:Mbytes),纵坐标为按照执行字节码指令计算的计时单位.其中,(a)为最大服务时间,(b)为最大等待时间,(c)为最大滞留时间,(d)为平均服务时间,(e)为平均等待时间,(f)为平均滞留时间,(g)是最大队长,(h)显示 Classbase 算法中浅释放对象的重复利用率.图 4 给出两种算法在不同堆空间大小条件下执行整理操作次数比较.

图 4 和图 5 分别给出 Marksweep 和 Classbase 两种算法在堆空间为 1024KB 条件下创建 100 万和 480 万个对象的平均服务时间,随着对象数目(系统运行时间)的增加,二者均以一定的振荡频率趋于一个恒定值.Classbase 算法由于 98%地重复使用被废弃的对象,恰好避免了回收时的对象移动和拷贝又省略了创建对象时刻的线性搜索操作,故显示出极高的效率.

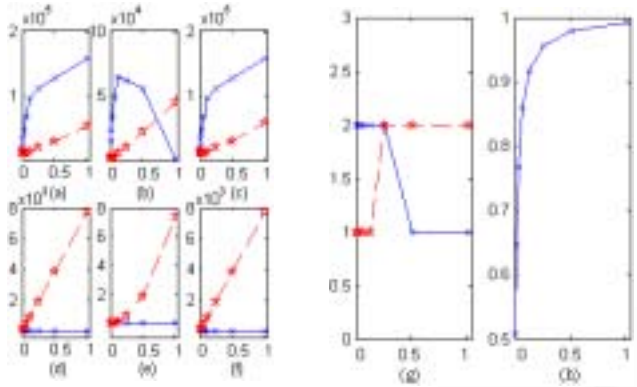

Fig.3

图 3

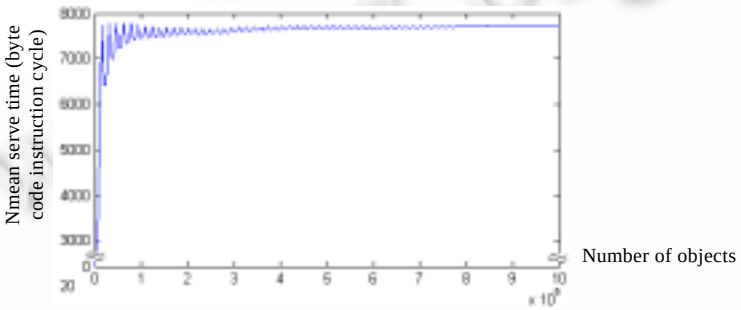

Fig.4 Time of creating 1million objects under Marksweep algorithm with 1MB heap size

图 4 堆空间 1MB,MarkSweep 算法创建 100 万个对象期间平均服务时间

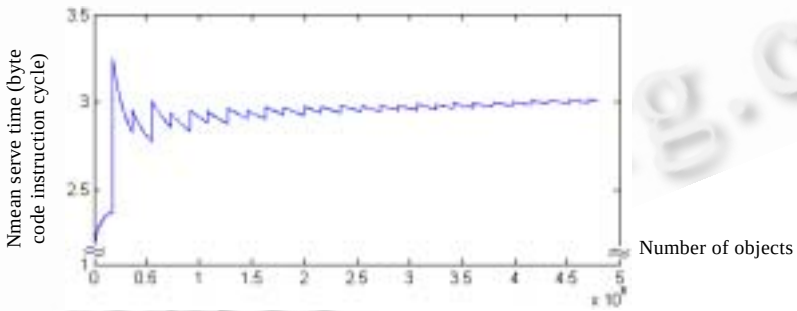

Fig.5 Time of creating 4.8 million objects under Classbase algorithm with 1MB heap size

图 5 堆空间 1MB,ClassBase 算法创建 480 万个对象期间平均服务时间

4 总 结

本文提出并实现了一种基于对象类型进行分类的堆空间管理算法,旨在提高 Web 应用服务系统运行时环境的效率.讨论了包括从算法所采用的数据结构到实现技术等各方面的细节.提出了废弃对象复用、线程缓存、子空间租赁等技术,并就基于统计信息的 GC 线程工作时机进行了分析.测试结果表明,在堆空间超过 1MB 时,对象的复用率可达 98%以上,这在改善运行时环境动态创建对象效率的同时,也减少了内存碎片的发生,能够获得较好的存储空间利用率.Classbase 算法更适合于系统资源相对丰富,但需要快速、频繁地创建、取消对象的信息服务应用领域.

References:

[1] Gosling J, Joy B, Steele G. The Java Language Specification. Addison-Wesley, 1998.

[2] Endo T, Taura K, Yonezawa A. A scalable mark-sweep garbage collector on large-scale shared-memory machines. Technical Report, Department of Information Science, University of Tokio, 2001.

[3] Appel AW. Simple generational garbage collection and fast allocation. Software Practice and Experience, 2002,19(2):171−183.

[4] Dijkstra EW, Lamport L, Martin AJ, Scholten CS, Steffens EFM. On-the-Fly garbage collection: An exercise in cooperation. Communications of the ACM, 1978,21(11):965−975.

[5] Doligez D, Leroy X. A concurrent generational garbage collector for a multi-threaded implementation of ML. In: Proc. of the Conf. Record of the 20th Annual ACM Symp. on Principles of Programming Languages. ACM SIGPLAN Notices, ACM Press. 1999. 113−123.

[6] Domany T, Kolodner E, Petrank E. A Generational On-the-Fly Garbage Collector for Java. In: Proc. of the SIGPLAN 2000 Conf. on Programming Language Design and Implementation, 2001. 274−284

[7] Hölzle U. A fast write barrier for generational garbage collectors. In: Moss E, Wilson PR, Zorn B, eds. Proc. of the OOPSLA/ECOOP'93 Workshop on Garbage Collection in Object-Oriented Systems. 1993. 92−97.

****

中国计算机学会设立创新奖

中国计算机学会创新奖是一项完全由社会力量设立的在计算机领域的科学技术创新奖。中国计算机学会负责该奖项评选和颁奖，每年评选一次。2005 年是首次。

中国计算机学会设立此奖的目的是为推动中国计算机及相关领域的科技创新和进步，促进科研成果的转化，促进 IT 产业的发展，推动科技界学术共同体评价体系的建立，发现和激励创新型科技人才。

中国计算机学会创新奖的评奖活动严格按照公开、公正的原则，根据《中国计算机学会创新奖评奖条例》执行。创新奖评委机构由 CCF 设立，评委会分为初评委员会和终评委员会，成员均为业内专家。为保证评选的公正性不受干扰，评奖委员会成员名单在最终评奖结果宣布以前不对外公布。主办单位对入围项目将予以公布，接受社会对入围项目的署名投诉。此外，对查实的不符合条例的行为将予以处罚。

申报中国计算机学会创新奖不需要缴纳任何费用，拥有科技成果的团体或个人均可申报参评此奖项，此外，获奖者还可获得证书和奖金。

2005 年度中国计算机学会创新奖的申报工作已经开始。

相关信息请登陆中国计算机学会网站 http://www.ccf.org.cn 了解。