Java垃圾收集算法与内存泄露 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/645/2021_2022_Java_E5_9E_83_E5_9C_BE_c104_645096.htm 1.垃圾收集算法的核心思想 Java语言建立了垃圾收集机制，用以跟踪正在使用的对象和发现并回收不再使用(引用)的对象。该机制可以有效防范动态内存分配中可能发生的两个危险：因内存垃圾过多而引发的内存耗尽，以及不恰当的内存释放所造成的内存非法引用。垃圾收集算法的核心思想是：对虚拟机可用内存空间，即堆空间中的对象进行识别，如果对象正在被引用，那么称其为存活对象，反之，如果对象不再被引用，则为垃圾对象，可以回收其占据的空间，用于再分配。垃圾收集算法的选择和垃圾收集系统参数的合理调节直接影响着系统性能，因此需要开发人员做比较深入的了解。 2.触发主GC(Garbage Collector)的条件 JVM进行次GC的频率很高,但因为这种GC占用时间极短,所以对系统产生的影响不大。更值得关注的是主GC的触发条件,因为它对系统影响很明显。总的来说,有两个条件会触发主GC: ①当应用程序空闲时,即没有应用线程在运行时,GC会被调用。因为GC在优先级最低的线程中进行,所以当应用忙时,GC线程就不会被调用,但以下条件除外。 ②Java堆内存不足时,GC会被调用。当应用线程在运行,并在运行过程中创建新对象,若这时内存空间不足,JVM就会强制地调用GC线程,以便回收内存用于新的分配。若GC一次之后仍不能满足内存分配的要求,JVM会再进行两次GC作进一步的尝试,若仍无法满足要求,则 JVM将报“ out of memory ”的错误,Java应用将停止。 由于是否进行主GC由JVM根据系统环境

决定,而系统环境在不断的变化当中,所以主GC的运行具有不确定性,无法预计它何时必然出现,但可以确定的是对一个长期运行的应用来说,其主GC是反复进行的。 3.减少GC开销的措施 根据上述GC的机制,程序的运行会直接影响系统环境的变化,从而影响GC的触发。若不针对GC的特点进行设计和编码,就会出现内存驻留等一系列负面影响。为了避免这些影响,基本的原则就是尽可能地减少垃圾和减少GC过程中的开销。具体措施包括以下几个方面: (1)不要显式调用System.gc() 此函数建议JVM进行主GC,虽然只是建议而非一定,但很多情况下它会触发主GC,从而增加主GC的频率,也即增加了间歇性停顿的次数。 (2)尽量减少临时对象的使用 临时对象在跳出函数调用后,会成为垃圾,少用临时变量就相当于减少了垃圾的产生,从而延长了出现上述第二个触发条件出现的时间,减少了主GC的机会。 (3)对象不用时最好显式置为Null 一般而言,为Null的对象都会被作为垃圾处理,所以将不用的对象显式地设为Null,有利于GC收集器判定垃圾,从而提高了GC的效率。 (4)尽量使用StringBuffer,而不用String来累加字符串(详见blog另一篇文章JAVA中String与StringBuffer) 由于String是固定长的字符串对象,累加String对象时,并非在一个String对象中扩增,而是重新创建新的String对象,如Str5=Str1 Str2 Str3 Str4,这条语句执行过程中会产生多个垃圾对象,因为对次作“ ”操作时都必须创建新的String对象,但这些过渡对象对系统来说是没有实际意义的,只会增加更多的垃圾。避免这种情况可以改用StringBuffer来累加字符串,因StringBuffer是可变长的,它在原有基础上进行扩增,不会产生中间对象。 (5)能用基本类型如Int,Long,就不用Integer,Long对象 基本类型变量占用的内存资源比相应对象

占用的少得多,如果没有必要,最好使用基本变量。 (6)尽量少用静态对象变量 静态变量属于全局变量,不会被GC回收,它们会一直占用内存。 (7)分散对象创建或删除的时间 集中在短时间内大量创建新对象,特别是大对象,会导致突然需要大量内存,JVM在面临这种情况时,只能进行主GC,以回收内存或整合内存碎片,从而增加主GC的频率。集中删除对象,道理也是一样的。它使得突然出现了大量的垃圾对象,空闲空间必然减少,从而大大增加了下一次创建新对象时强制主GC的机会。 4.gc与finalize方法 ⑴gc方法请求垃圾回收 使用System.gc()可以不管JVM使用的是哪一种垃圾回收的算法，都可以请求Java的垃圾回收。需要注意的是，调用System.gc()也仅仅是一个请求。JVM接受这个消息后，并不是立即做垃圾回收，而只是对几个垃圾回收算法做了加权，使垃圾回收操作容易发生，或提早发生，或回收较多而已。 ⑵finalize方法透视垃圾收集器的运行 在JVM垃圾收集器收集一个对象之前，一般要求程序调用适当的方法释放资源，但在没有明确释放资源的情况下，Java提供了缺省机制来终止化该对象释放资源，这个方法就是finalize()。它的原型为： protected void finalize() throws Throwable 在finalize()方法返回之后，对象消失，垃圾收集开始执行。原型中的throws Throwable表示它可以抛出任何类型的异常。 因此，当对象即将被销毁时，有时需要做一些善后工作。可以把这些操作写在finalize()方法里。 java 代码 protected void finalize() { // finalization code here } ⑶代码示例 java 代码 class Garbage{ int index. static int count. Garbage() { count . System.out.println("object " count " construct"). setID(count). } void setID(int id) { index=id. } protected void

finalize() //重写finalize方法 { System.out.println("object " index " is reclaimed"). } public static void main(String[] args) { new Garbage(). new Garbage(). new Garbage(). new Garbage(). System.gc(). //请求运行垃圾收集器 } } 5.Java 内存泄漏 由于采用了垃圾回收机制，任何不可达对象(对象不再被引用)都可以由垃圾收集线程回收。因此通常说的Java 内存泄漏其实是指无意识的、非故意的对象引用，或者无意识的对象保持。无意识的对象引用是指代码的开发人员本来已经对对象使用完毕，却因为编码的错误而意外地保存了对该对象的引用(这个引用的存在并不是编码人员的主观意愿)，从而使得该对象一直无法被垃圾回收器回收掉，这种本来以为可以释放掉的却最终未能被释放的空间可以认为是被“ 泄漏了 ”。 考虑下面的程序,在ObjStack类中,使用push和pop方法来管理堆栈中的对象。两个方法中的索引(index)用于指示堆栈中下一个可用位置。push方法存储对新对象的引用并增加索引值,而pop方法减小索引值并返回堆栈最上面的元素。在main方法中,创建了容量为64的栈,并64次调用push方法向它添加对象,此时index的值为64,随后又32次调用pop方法,则index的值变为32,出栈意味着在堆栈中的空间应该被收集。但事实上,pop方法只是减小了索引值,堆栈仍然保持着对那些对象的引用。故32个无用对象不会被GC回收,造成了内存渗漏。 java 代码 public class ObjStack { private Object[] stack. private int index. ObjStack(int indexcount) { stack = new Object[indexcount]. index = 0. } public void push(Object obj) { stack[index] = obj. index . } public Object pop() { index--. return stack[index]. } } public class Pushpop { public static void main(String[] args) { int i = 0. Object tempobj.

//new一个ObjStack对象，并调用有参构造函数。分配stack Obj数组的空间大小为64，可以存64个对象，从0开始存储 ObjStack stack1 = new ObjStack(64). while (i 100Test 下载频道开通，各类考试题目直接下载。详细请访问 www.100test.com