链滴

JVM|04 垃圾回收

作者：qq692310342

原文链接：https://ld246.com/article/1566015861889

来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20180914.jpg?imageView2/1/w/960/h/540/interlace/1/q/100">
<h2 id="什么是垃圾回收机制">什么是垃圾回收机制</h2>
<h3 id="背景知识">背景知识</h3>
程序的运行必然需要申请内存资源，无效的对象资源如果不及时处理就会一直占有内存资源，最
将导致内存溢出，所以对内存资源的管理是非常重要了。
<h3 id="Java语言的垃圾回收">Java 语言的垃圾回收</h3>
为了让程序员更专注于代码的实现，而不用过多的考虑内存释放的问题，所以，在 Java 语言中
有了自动的垃圾回收机制，也就是我们熟悉的 GC。
有了垃圾回收机制后，程序员只需要关心内存的申请即可，内存的释放由系统自动识别完成。
换句话说，自动的垃圾回收的算法就会变得非常重要了，如果因为算法的不合理，导致内存资源一直
有释放，同样也可能会导致内存溢出的。
当然，除了 Java 语言，C#、Python 等语言也都有自动的垃圾回收机制。
<h2 id="Java的垃圾回收常用算法">Java 的垃圾回收常用算法</h2>
<h3 id="引用计数法">引用计数法</h3>
<h4 id="原理">原理</h4>
假设有一个对象 A，任何一个对象对 A 的引用，那么对象 A 的引用计数器 +1，当引用失败时，
象 A 的引用计数器就-1，如果对象 A 的计数器的值为 0，就说明对象 A 没有引用了，可以被回收。<
p>
<h4 id="优劣分析">优劣分析</h4>

优势
实时性较高，无需等到内存不够的时候，才开始回收，运行时根据对象的计数器是否为 0，就可以直
回收。
在垃圾回收过程中，应用无需挂起。如果申请内存时，内存不足，则立刻报
outofmember 错误。
区域性，更新对象的计数器时，只是影响到该对象，不会扫描全部对象。
劣势
每次对象被引用时，都需要去更新计数器，有一点时间开销。
浪费 CPU 资源，即使内存够用，仍然在运行时进行计数器的统计。
无法解决循环引用问题。（最大的缺点）
循环问题演示：

<pre><code class="highlight-chroma"><span class="highlight
cl">class TestA {
    public TestB b ;
}
class TestB{
    public TestA a;
}

public class Main {
    public static vo
d main(String[] args) {
        TestA a = ne
 TestA();
        TestB b = ne
 TestB();
        a.b = b;
        b.a = a;
        a = null;
        b = null;
//   虽然被设置为
ull，但是a与b之间依旧存在着循环引用的问题

    }
}
</code></pre>
<h3 id="标记-清除算法">标记-清除算法</h3>
<h4 id="原理-">原理</h4>
标记清除算法，是将垃圾回收分为 2 个阶段，分别是标记和清除。
标记：从根节点开始标记引用的对象。
清除：未被标记引用的对象就是垃圾对象，可以被清理。

初始状态下，所有的目标对象都是为 0（未被标记）
待 jvm 出现有效内存耗尽，就会挂起线程，执行 GC 线程，进行标记

从根节点进行标记到最后，然后回收未被标记的对象。
清理完毕之后挂起 gc 线程，重新执行原先被挂起的线程。
而被标记的对象会被重新置 0；

<h4 id="优劣分析-">优劣分析</h4>

优势
很明显的解决了循环应用导致的不能被回收的问题
缺点
缺点也很明显
效率较低，标记和清除两个动作都需要遍历所有的对象，并且在 GC 时，需要停止应用程序，对于交性要求比较高的应用而言这个体验是非常差的。
通过标记清除算法清理出来的内存，碎片化较为严重，因为被回收的对象可能存在于内存的各个角落所以清理出来的内存是不连贯的

<h3 id="标记-压缩算法">标记-压缩算法</h3>
<h4 id="原理--">原理</h4>
标记压缩算法是在标记清除算法的基础之上，做了优化改进的算法。和标记清除算法一样，也是根节点开始，对对象的引用进行标记，在清理阶段，并不是简单的清理未标记的对象，而是将存活的象压缩到内存的一端，然后清理边界以外的垃圾，从而解决了碎片化的问题。

<h4 id="优劣分析--">优劣分析</h4>

优点
在标记清除算法的基础上解决了产生碎片的问题
缺点
算法多出一步压缩，所以在性能上也会有所影响

<h3 id="复制算法">复制算法</h3>
<h4 id="原理---">原理</h4>
复制算法的核心就是：将原有的内存空间一分为二，每次只用其中的一块，在垃圾回收时，将正使用的对象复制到另一个内存空间中，然后将该内存空间清空，交换两个内存的角色，完成垃圾的回。
典型的复制算法的落地实现就是：jvm 中堆内存的年轻代的 gc 策略（具体可以看我 jvm 系列的博客<a href="https://ld246.com/forward?goto=https%3A%2F%2Fwww.jeffcc.top%2Farticles%2F219%2F08%2F16%2F1565950696850.html" target="_blank" rel="nofollow ugc">内存模型</a的那一部分内容）

在 GC 开始的时候，对象只会存在于 Eden 区和名为“From”的 Survivor 区，Survivor 区“To 是空的。
紧接着进行 GC，Eden 区中所有存活的对象都会被复制到“To”，而在“From”区中仍存活的象会根据他们的年龄值来决定去向。年龄达到一定值(年龄阈值，可以通过- XX:MaxTenuringThresho d 来设置)的对象会被移动到年老代中，没有达到阈值的对象会被复制到“To”区域。
经过这次 GC 后，Eden 区和 From 区已经被清空。这个时候，“From”和“To”会交他们的色，也就是新的“To”就是上次 GC 前的“From”，新的“From”就是上次 GC 前的“To”。不怎样，都会保证名为 To 的 Survivor 区域是空的。
GC 会一直重复这样的过程，直到“To”区被填满，“To”区被填满之后，会将所有象移动到年代中。

<h4 id="优劣分析---">优劣分析</h4>

优势
在垃圾对象多的情况下，效率较高
清理后，内存无碎片
劣势
在垃圾对象少的情况下，不适用，如：老年代内存
分配的 2 块内存空间，在同一个时刻，只能使用一半，内存使用率仅有 50%

<h3 id="分代算法">分代算法</h3>
前面介绍了多种回收算法，每一种算法都有自己的优点也有缺点，谁都不能替代谁，所以根据垃回收对象的特点进行选择，才是明智的选择。
分代算法其实就是这样的，根据回收对象的特点进行选择，在 jvm 中，年轻代适合使用复制算法，老代适合使用标记清除或标记压缩算法。
<hr>
END
2019 年 8 月 17 日 12:23:50