昵称:平凡希
园龄:8年4个月
粉丝:1071
关注:1
+加关注

< 2019年4月 >

日	一	二	三	四	五	六
31	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	1	2	3	4
5	6	7	8	9	10	11

搜索
找找看
谷歌搜索

常用链接
我的随笔
我的评论
我的参与
最新评论
我的标签

随笔分类
Java IO(5)
Java NIO(7)
java8(3)
Java多线程(20)
java基础(11)
Java集合(8)
Java虚拟机(9)
Linux
Mybatis(9)
mysql(15)
redis(9)
spring(8)
SpringMVC(8)
设计模式(5)

随笔档案
2018年6月 (1)
2018年4月 (1)
2018年1月 (3)
2017年12月 (2)
2017年11月 (10)
2017年10月 (3)
2017年9月 (6)
2017年8月 (9)
2017年7月 (6)
2017年6月 (12)
2017年5月 (3)
2017年4月 (3)
2017年3月 (24)
2017年2月 (14)
2017年1月 (2)
2016年12月 (5)
2016年11月 (4)

Java垃圾回收(GC)机制详解

一、为什么需要垃圾回收

如果不进行垃圾回收，内存迟早都会被消耗空，因为我们在不断的分配内存空间而不进行回收。除非内存无限大，我们可以任性的分配而不回收，但是事实并非如此。所以，垃圾回收是必须的。

二、哪些内存需要回收？

哪些内存需要回收是垃圾回收机制第一个要考虑的问题，所谓"要回收的垃圾"无非就是那些不可能再被任何途径使用的对象。那么如何找到这些对象？

1、引用计数法

这个算法的实现是，给对象中添加一个引用计数器，每当一个地方引用这个对象时，计数器值+1；当引用失效时，计数器值-1。任何时刻计数值为0的对象就是不可能再被使用的。这种算法使用场景很多，但是，Java中却没有使用这种算法，因为这种算法很难解决对象之间相互引用的情况。看一段代码：

```
/**
 * 虚拟机参数:-verbose:gc
 */
public class ReferenceCountingGC
{
    private Object instance = null;
    private static final int _1MB = 1024 * 1024;

    /** 这个成员属性唯一的作用就是占用一点内存 */
    private byte[] bigSize = new byte[2 * _1MB];

    public static void main(String[] args)
    {
        ReferenceCountingGC objectA = new ReferenceCountingGC();
        ReferenceCountingGC objectB = new ReferenceCountingGC();
        objectA.instance = objectB;
        objectB.instance = objectA;
        objectA = null;
        objectB = null;

        System.gc();
    }
}
```

看下运行结果：

```
[GC 4417K->288K(61440K), 0.0013498 secs]
[Full GC 288K->194K(61440K), 0.0094790 secs]
```

看到，两个对象相互引用着，但是虚拟机还是把这两个对象回收掉了，这也说明虚拟机并不是通过引用计数法来判定对象是否存活的。

2、可达性分析法

这个算法的基本思想是通过一系列称为"GC Roots"的对象作为起始点，从这些节点向下搜索，搜索所走过的路径称为引用链，当一个对象到GC Roots没有任何引用链(即GC Roots到对象不可达)时，则证明此对象是不可用的。

那么问题又来了，如何选取GCRoots对象呢？在Java语言中，可以作为GCRoots的对象包括下面几种：

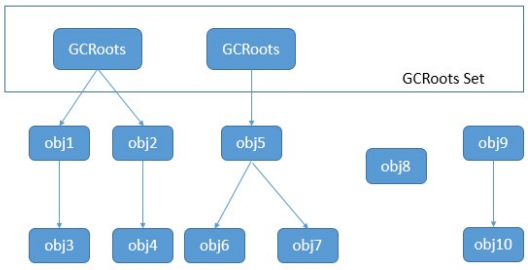
(1). 虚拟机栈(栈帧中的局部变量区，也叫做局部变量表)中引用的对象。

(2). 方法区中的类静态属性引用的对象。

(3). 方法区中常量引用的对象。

(4). 本地方法栈中JNI(Native方法)引用的对象。

下面给出一个GCRoots的例子，如下图，为GCRoots的引用链。



由图可知，obj8、obj9、obj10都没有到GCRoots对象的引用链，即便obj9和obj10之间有引用链，他们还是会被当成垃圾处理，可以进行回收。

三、四种引用状态

在JDK1.2之前，Java中引用的定义很传统：如果引用类型的数据中存储的数值代表的是另一块内存的起始地址，就称这块内存代表着一个引用。这种定义很纯粹，但是太过于狭隘，一个对象只有被引用或者没被引用两种状态。我们希望描述这样一类对象：当内存空间还足够时，则能保留在内存中；如果内存空间在进行垃圾收集后还是非常紧张，则可以抛弃这些对象。很多系统的缓存功能都符合这样的应用场景。在JDK1.2之后，Java对引用的概念进行了扩充，将引用分为强引用、软引用、弱引用、虚引用4种，这4种引用强度依次减弱。

1、强引用

代码中普遍存在的类似"Object obj = new Object()"这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象。

2、软引用

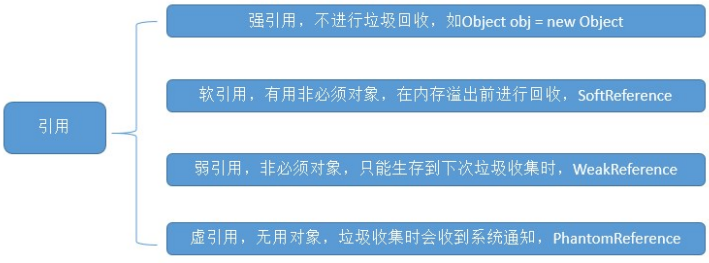
描述有些还有用但并非必需的对象。在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围进行二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。Java中的类SoftReference表示软引用。

3、弱引用

描述非必需对象。被弱引用关联的对象只能生存到下一次垃圾回收之前，垃圾收集器工作之后，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。Java中的类WeakReference表示弱引用。

4、虚引用

这个引用存在的唯一目的就是在这个对象被收集器回收时收到一个系统通知，被虚引用关联的对象，和其生存时间完全没关系。Java中的类PhantomReference表示虚引用。



对象引用分类

对于可达性分析算法而言，未到达的对象并非是“非死不可”的，若要宣判一个对象死亡，至少需要经历两次标记阶段。

1. 如果对象在进行可达性分析后发现没有与GCRoots相连的引用链，则该对象被第一次标记并进行一次筛选，筛选条件为是否有必要执行该对象的finalize方法，若对象没有覆盖finalize方法或者该finalize方法是否已经被虚拟机执行过了，则均视作不必要执行该对象的finalize方法，即该对象将会被回收。反之，若对象覆盖了finalize方法并且该finalize方法并没有被执行过，那么，这个对象会被放置在一个叫F-Queue的队列中，之后会由虚拟机自动建立的、优先级低的Finalizer线程去执行，而虚拟机不必要等待该线程执行结束，即虚拟机只负责建立线程，其他的事情交给此线程去处理。

2.对F-Queue中对象进行第二次标记，如果对象在finalize方法中拯救了自己，即关联上了GCRoots引用链，如把this关键字赋值给其他变量，那么在第二次标记的时候该对象将从“即将回收”的集合中移除，如果对象还是没有拯救自己，那就会被回收。如下代码演示了一个对象如何在finalize方法中拯救了自己，然而，它只能拯救自己一次，第二次就被回收了。具体代码如下：

```
package com.demo;

/*
 * 此代码演示了两点：
 * 1.对象可以再被GC时自我拯救
 * 2.这种自救的机会只有一次，因为一个对象的finalize()方法最多只会被系统自动调用一次
 * */
public class FinalizeEscapeGC {

    public String name;
    public static FinalizeEscapeGC SAVE_HOOK = null;

    public FinalizeEscapeGC(String name) {
        this.name = name;
    }

    public void isAlive() {
        System.out.println("yes, i am still alive :)");
    }

    @Override
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.println("finalize method executed!");
        System.out.println(this);
        FinalizeEscapeGC.SAVE_HOOK = this;
    }

    @Override
    public String toString() {
        return name;
    }

    public static void main(String[] args) throws InterruptedException {
        SAVE_HOOK = new FinalizeEscapeGC("leesf");
        System.out.println(SAVE_HOOK);
        // 对象第一次拯救自己
        SAVE_HOOK = null;
        System.out.println(SAVE_HOOK);
        System.gc();
        // 因为finalize方法优先级很低，所以暂停0.5秒以等待它
        Thread.sleep(500);
        if (SAVE_HOOK != null) {
            SAVE_HOOK.isAlive();
        } else {
            System.out.println("no, i am dead : (");
        }

        // 下面这段代码与上面的完全相同，但是这一次自救却失败了
        // 一个对象的finalize方法只会被调用一次
        SAVE_HOOK = null;
        System.gc();
        // 因为finalize方法优先级很低，所以暂停0.5秒以等待它
        Thread.sleep(500);
        if (SAVE_HOOK != null) {
            SAVE_HOOK.isAlive();
        } else {
            System.out.println("no, i am dead : (");
        }
    }
}
```

运行结果如下：

```
leesf
null
finalize method executed!
leesf
yes, i am still alive :)
no, i am dead : (
```

由结果可知，该对象拯救了自己一次，第二次没有拯救成功，因为对象的finalize方法最多被虚拟机调用一次。此外，从结果我们可以得知，一个堆对象的this（放在局部变量表中的第一项）引用会永远存在，在方法体内可以将this引用赋值给其他变量，这样堆中对象就可以被其他变量所引用，即不会被回收。

四、方法区的垃圾回收

方法区的垃圾回收主要回收两部分内容：1. 废弃常量。2. 无用的类。既然进行垃圾回收，就需要判断哪些是废弃常量，哪些是无用的类。

如何判断废弃常量呢？以字面量回收为例，如果一个字符串“abc”已经进入常量池，但是当前系统没有任何一个String对象引用了叫做“abc”的字面量，那么，如果发生垃圾回收并且有必要时，“abc”就会被系统移出常量池。常量池中的其他类（接口）、方法、字段的符号引用也与此类似。

如何判断无用的类呢？需要满足以下三个条件

1. 该类的所有实例都已经被回收，即Java堆中不存在该类的任何实例。

2. 加载该类的ClassLoader已经被回收。

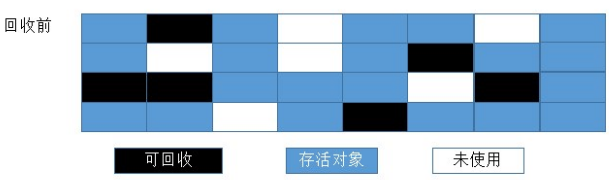
3. 该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

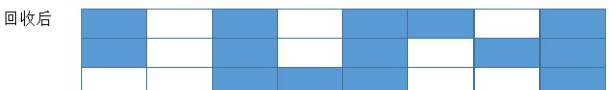
满足以上三个条件的类可以进行垃圾回收，但是并不是无用就被回收，虚拟机提供了一些参数供我们配置。

五、垃圾收集算法

1、标记-清除（Mark-Sweep）算法

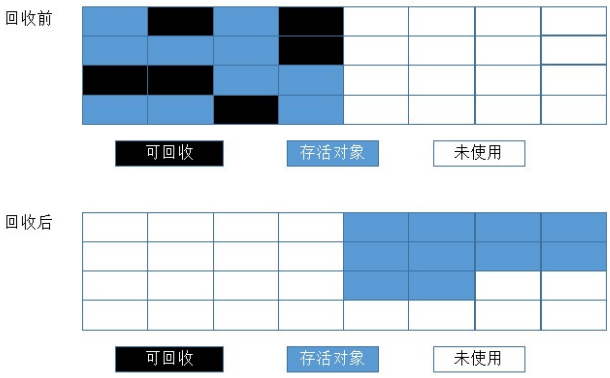
这是最基础的算法，标记-清除算法就如同它的名字样，分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，标记完成后统一回收所有被标记的对象。这种算法的不足主要体现在效率和空间，从效率的角度讲，标记和清除两个过程的效率都不高；从空间的角度讲，标记清除后会产生大量不连续的内存碎片，内存碎片太多可能会导致以后程序运行过程中在需要分配较大对象时，无法找到足够的连续内存而不得不提前触发一次垃圾收集动作。标记-清除算法执行过程如图：





2、复制(Copying)算法

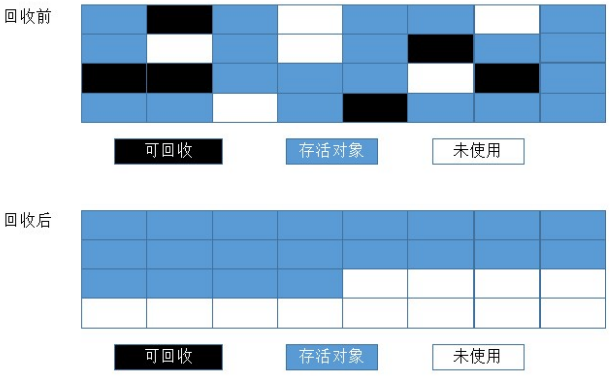
复制算法是为了解决效率问题而出现的，**它将可用的内存分为两块，每次只用其中一块，当这一块内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已经使用过的内存空间一次性清理掉**。这样每次只需要对整个半区进行内存回收，内存分配时也不需要考虑内存碎片等复杂情况，只需要移动指针，按照顺序分配即可。复制算法的执行过程如图：



不过这种算法有个缺点，**内存缩小为了原来的一半，这样代价太高了**。现在的商用虚拟机都采用这种算法来**回收新生代**，不过研究表明1:1的比例非常不科学，因此**新生代的内存被划分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块Survivor**。每次回收时，将Eden和Survivor中还存活着的对象一次性复制到另外一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认Eden区和Survivor区的比例为8:1，意思是每次新生代中可用内存空间为整个新生代容量的90%。当然，我们没有办法保证每次回收都只有不多于10%的对象存活，当Survivor空间不够用时，需要依赖老年代进行分配担保(Handle Promotion)。

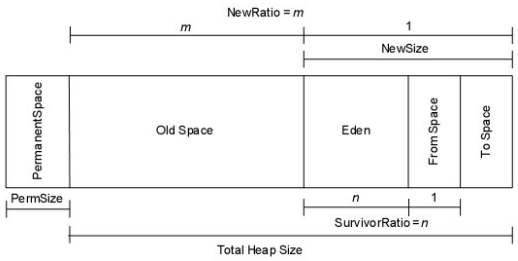
3、标记-整理(Mark-Compact)算法

复制算法在对象存活率较高的场景下要进行大量的复制操作，效率很低。万一对象100%存活，那么需要有额外的空间进行分配担保。老年代都是不易被回收的对象，对象存活率高，因此一般不能直接选用复制算法。根据老年代的特点，有人提出了另外一种标记-整理算法，过程与标记-清除算法一样，不过不是直接对可回收对象进行清理，而是让所有存活对象都向一端移动，然后直接清理掉边界以外的内存。标记-整理算法的工作过程如图：



4、分代收集算法

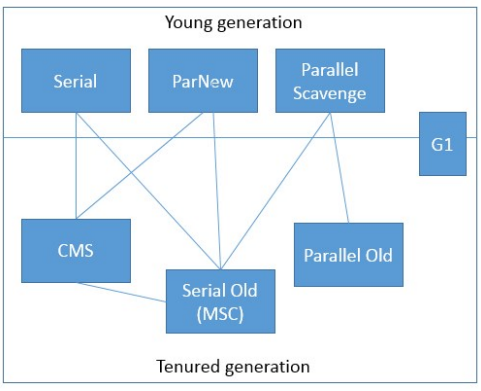
根据上面的内容，用一张图概括一下堆内存的布局



现代商用虚拟机基本都采用**分代收集算法**来进行垃圾回收。这种算法没什么特别的，无非是上面内容的结合罢了，根据对象的生命周期的不同将内存划分为几块，然后根据各块的特点采用最适当的收集算法。**大批对象死去、少量对象存活的(新生代)，使用复制算法，复制成本低；对象存活率高、没有额外空间进行分配担保的(老年代)，采用标记-清理算法或者标记-整理算法。**

六、垃圾收集器

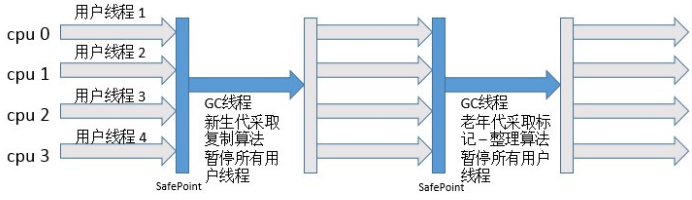
垃圾收集器就是上面讲的理论知识的具体实现了。不同虚拟机所提供的垃圾收集器可能会有很大差别，我们使用的是HotSpot，HotSpot这个虚拟机所包含的所有收集器如图：



上图展示了7种作用于不同分代的收集器，如果两个收集器之间存在连线，那说明它们可以搭配使用。虚拟机所处的区域说明它是属于新生代收集器还是老年代收集器。多说一句，我们必须明确一个观点：**没有最好的垃圾收集器，更加没有万能的收集器，只能选择对具体应用最合适的收集器**。这也是HotSpot为什么要实现这么多收集器的原因。OK，下面一个一个看一下收集器。

1、Serial收集器

最基本、发展历史最久的收集器，这个收集器是一个**采用复制算法的单线程的收集器，单线程一方面意味着它只会使用一个CPU或一条线程去完成垃圾收集工作，另一方面也意味着它进行垃圾收集时必须暂停其他线程的所有工作，直到它收集结束为止**。后者意味着，在用户不可见的情况下要把用户正常工作的线程全部停掉，这对很多应用是难以接受的。不过实际上到目前为止，**Serial收集器依然是虚拟机运行在Client模式下的默认新生代收集器**，因为它简单而高效。用户桌面应用场景中，分配给虚拟机管理的内存一般来说不会很大，收集几十兆甚至一两百兆的新生代停顿时间在几十毫秒最多一百毫秒，只要不是频繁发生，这点停顿是完全可以接受的。Serial收集器运行过程如下图所示：

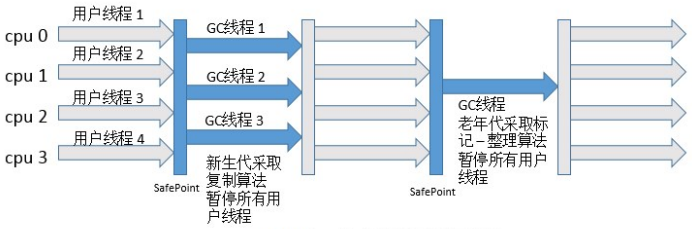


Serial/Serial Old 收集器运行示意图

说明：1. 需要STW（Stop The World），停顿时间长。2. 简单高效，对于单个CPU环境而言，Serial收集器由于没有线程交互开销，可以获取最高的单线程收集效率。

2、ParNew收集器

ParNew收集器其实就是Serial收集器的多线程版本，除了使用多条线程进行垃圾收集外，其余行为和Serial收集器完全一样，包括使用的也是复制算法。ParNew收集器除了多线程以外和Serial收集器并没有太多创新的地方，**但是它却是Server模式下的虚拟机首选的新生代收集器**，其中有一个很重要的和性能无关的原因是，除了Serial收集器外，目前只有它能与CMS收集器配合工作（看图）。CMS收集器是一款几乎可以认为有划时代意义的垃圾收集器，因为它第一次实现了让垃圾收集线程与用户线程基本上同时工作。ParNew收集器在单CPU的环境中绝对不会有比Serial收集器更好的效果，甚至由于线程交互的开销，该收集器在两个CPU的环境中都不能百分之百保证可以超越Serial收集器。当然，随着可用CPU数量的增加，它对于GC时系统资源的有效利用还是很有好处的。它默认开启的收集线程数与CPU数量相同，在CPU数量非常多的情况下，可以使用-XX:ParallelGCThreads参数来限制垃圾收集的线程数。ParNew收集器运行过程如下图所示：



ParNew /Serial Old 收集器运行示意图

3、Parallel Scavenge收集器

Parallel Scavenge收集器也是一个新生代收集器，也是用复制算法的收集器，也是并行的多线程收集器，但是它的特点是它的关注点和其他收集器不同。介绍这个收集器主要还是介绍**吞吐量**的概念。**CMS等收集器的关注点是尽可能缩短垃圾收集时用户线程的停顿时间，而Parallel Scavenge收集器的目标则是打到一个可控制的吞吐量。**所谓吞吐量的意思就是CPU用于运行用户代码时间与CPU总消耗时间的比值，即**吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间）**，虚拟机总运行100分钟，垃圾收集1分钟，那吞吐量就是99%。另外，**Parallel Scavenge收集器是虚拟机运行在Server模式下的默认垃圾收集器。**

停顿时间短适合需要与用户交互的程序，良好的响应速度能提升用户体验；高吞吐量则可以高效率利用CPU时间，尽快完成运算任务，主要适合在后台运算而不需要太多交互的任务。

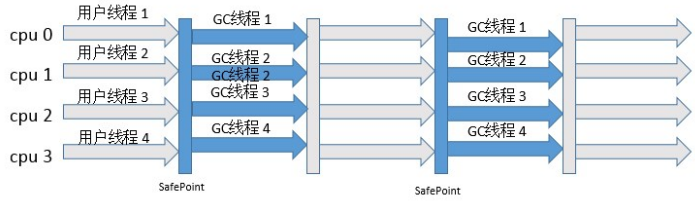
虚拟机提供了-XX:MaxGCPauseMillis和-XX:GCTimeRatio两个参数来精确控制最大垃圾收集停顿时间和吞吐量大小。不过不要以为前者越小越好，GC停顿时间的缩短是以牺牲吞吐量和新生代空间换取的。由于与吞吐量关系密切，**Parallel Scavenge收集器也被称为"吞吐量优先收集器"**。Parallel Scavenge收集器有一个-XX:+UseAdaptiveSizePolicy参数，这是一个开关参数，这个参数打开之后，就不需要手动指定新生代大小、Eden区和Survivor参数等细节参数了，虚拟机会根据当前系统的运行情况手机性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量。**如果对于垃圾收集器运作原理不太了解，以至于在优化比较困难的时候，使用Parallel Scavenge收集器配合自适应调节策略，把内存管理的调优任务交给虚拟机去完成将是一个不错的选择。**

4、Serial Old收集器

Serial收集器的老年代版本，同样是一个单线程收集器，使用"标记-整理算法"，这个收集器的主要意义也是在于给Client模式下的虚拟机使用。

5、Parallel Old收集器

Parallel Scavenge收集器的老年代版本，使用多线程和"标记-整理"算法。这个收集器在JDK 1.6之后的出现，"吞吐量优先收集器"终于有了比较名副其实的应用组合，在注重吞吐量以及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge收集器+Parallel Old收集器的组合。运行过程如下图所示：



Parallel Scavenge/Parallel Old 收集器运行示意图

6、CMS收集器

CMS（Conrrurent Mark Sweep）收集器是以获取最短回收停顿时间为目标的收集器。使用标记 - 清除算法，收集过程分为如下四步：

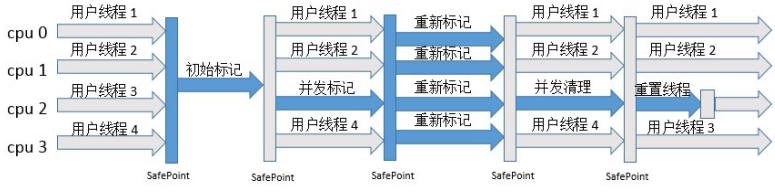
(1). 初始标记，标记GCRoots能直接关联到的对象，时间很短。

(2). 并发标记，进行GCRoots Tracing（可达性分析）过程，时间很长。

(3). 重新标记，修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，时间较长。

(4). 并发清除，回收内存空间，时间很长。

其中，并发标记与并发清除两个阶段耗时最长，但是可以与用户线程并发执行。运行过程如下图所示：



Concurrent Mark Sweep收集器运行示意图

说明：1. 对CPU资源非常敏感，可能会导致应用程序变慢，吞吐率下降。2. 无法处理浮动垃圾，因为在并发清理阶段用户线程还在运行，自然就会产生新的垃圾，而在此次收集中无法收集他们，只能留到下次收集，这部分垃圾为浮动垃圾，同时，由于用户线程并发执行，所以需要预留一部分老年代空间提供并发收集时程序运行使用。3. 由于采用的标记 - 清除算法，会产生大量的内存碎片，不利于大对象的分配，可能会提前触发一次Full GC。虚拟机提供了-XX:+UseCMSCompactAtFullCollection参数来进行碎片的合并整理过程，这样会使得停顿时间变长，虚拟机还提供了一个参数配置，-XX:+CMSFullGCsBeforeCompaction，用于设置执行多少次不压缩的Full GC后，接着来一次带压缩的GC。

7、G1收集器

G1是目前技术发展的最前沿成果之一，HotSpot开发团队赋予它的使命是未来可以替换掉JDK1.5中发布的CMS收集器。与其他GC收集器相比，G1收集器有以下特点：

(1). 并行和并发。使用多个CPU来缩短Stop The World停顿时间，与用户线程并发执行。

(2). 分代收集。独立管理整个堆，但是能够采用不同的方式去处理新创建对象和已经存活了一段时间、熬过多次GC的旧对象，以获取更好的收集效果。

(3). 空间整合。基于标记 - 整理算法，无内存碎片产生。

(4). 可预测的停顿。能简历可预测的停顿时间模型，能让使用者明确指定在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒。

在G1之前的垃圾收集器，收集的范围都是整个新生代或者老年代，而G1不再是这样。使用G1收集器时，Java堆的内存布局与其他收集器有很大差别，它将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分（可以不连续）Region的集合。

8、常用的收集器组合

七、理解GC日志

每种收集器的日志形式都是由它们自身的实现所决定的，换言之，每种收集器的日志格式都可以不一样。不过虚拟机为了方便用户阅读，将各个收集器的日志都维持了一定的共性，来看下面的一段GC日志：

```
[GC [DefNew: 310K->194K(2368K), 0.0269163 secs] 310K->194K(7680K), 0.0269513 secs] [Times: user=0.00 sys=0.00, real=0.03 secs]
[GC [DefNew: 2242K->0K(2368K), 0.0018814 secs] 2242K->2241K(7680K), 0.0019172 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (System) [Tenured: 2241K->193K(5312K), 0.0056517 secs] 4289K->193K(7680K), [Perm : 2950K->2950K(21248K)], 0.0057094 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
 def new generation   total 2432K, used 43K [0x00000000052a0000, 0x0000000005540000, 0x0000000006ea0000)
  eden space 2176K,   2% used [0x00000000052a0000, 0x00000000052aaeb8, 0x00000000054c0000)
  from space 256K,    0% used [0x00000000054c0000, 0x00000000054c0000, 0x0000000005500000)
  to   space 256K,    0% used [0x0000000005500000, 0x0000000005500000, 0x0000000005540000)
 tenured generation   total 5312K, used 193K [0x0000000006ea0000, 0x00000000073d0000, 0x000000000a6a0000)
   the space 5312K,   3% used [0x0000000006ea0000, 0x0000000006ed0730, 0x0000000006ed0800, 0x00000000073d0000)
 compacting perm gen  total 21248K, used 2982K [0x000000000a6a0000, 0x000000000bb60000, 0x000000000faa0000)
   the space 21248K,  14% used [0x000000000a6a0000, 0x000000000a989980, 0x000000000a989a00, 0x000000000bb60000)
No shared spaces configured.
```

1、日志的开头“GC”、“Full GC”表示这次垃圾收集的停顿类型，而不是用来区分新生代GC还是老年代GC的。如果有Full，则说明本次GC停止了其他所有工作线程(Stop-The-World)。看到Full GC的写法是“Full GC(System)”，这说明是调用System.gc()方法所触发的GC。

2、“GC”中接下来的“[DefNew”表示GC发生的区域，这里显示的区域名称与使用的GC收集器是密切相关的，例如上面样例所使用的Serial收集器中的新生代名为“Default New Generation”，所以显示的是“[DefNew”。如果是ParNew收集器，新生代名称就会变为“[ParNew”，意为“Parallel New Generation”。如果采用Parallel Scavenge收集器，那它配套的新生代称为“PSYoungGen”，老年代和永久代同理，名称也是由收集器决定的。

3、后面方括号内部的“310K->194K(2368K)”、“2242K->0K(2368K)”，指的是**该区域已使用的容量->GC后该内存区域已使用的容量(该内存区总容量)**。方括号外面的“310K->194K(7680K)”、“2242K->2241K(7680K)”则指的是**GC前Java堆已使用的容量->GC后Java堆已使用的容量(Java堆总容量)**。

4、再往后“0.0269163 secs”表示该内存区域GC所占用的时间，单位是秒。最后的“[Times: user=0.00 sys=0.00 real=0.03 secs]”则更具体了，user表示用户态消耗的CPU时间、内核态消耗的CPU时间、操作从开始到结束经过的墙钟时间。后面两个的区别是，墙钟时间包括各种非运算的等待消耗，比如等待磁盘I/O、等待线程阻塞，而CPU时间不包括这些耗时，但当系统有多CPU或者多核的话，多线程操作会叠加这些CPU时间，所以如果看到user或sys时间超过real时间是完全正常的。

5、“Heap”后面就列举出堆内存目前各个年代的区域的内存情况。

分类: Java虚拟机

平凡希
关注 - 1
粉丝 - 1071
+加关注

« 上一篇:Mybatis学习总结(九)——查询缓存
» 下一篇:jvm调优-从eclipse开始

posted @ 2017-03-30 21:20 平凡希 阅读(64788) 评论(5) 编辑 收藏

评论列表

#1楼 2018-02-07 16:37 yxlaisj

请问从哪里看垃圾回收的日志，比如第一块代码：

[GC 4417K->288K(61440K), 0.0013498 secs]
[Full GC 288K->194K(61440K), 0.0094790 secs]

支持(4) 反对(0)

#2楼 2018-07-18 09:05 炽热的火

[GC 4417K->288K(61440K), 0.0013498 secs]
[Full GC 288K->194K(61440K), 0.0094790 secs]
这个你是在哪里看见的？

支持(1) 反对(0)

#3楼 2019-01-08 15:29 miralily_swing

[GC 4417K->288K(61440K), 0.0013498 secs]
[Full GC 288K->194K(61440K), 0.0094790 secs]

请问，垃圾回收的日志，是从哪里看的？

支持(0) 反对(0)

#4楼 2019-01-16 11:51 漫步ing

@ miralily_swing
https://blog.csdn.net/u013613428/article/details/53763925

支持(0) 反对(0)

#5楼 2019-02-23 11:20 czwbig

@ yxlaisj
通过java命令参数可以输出，例如
-XX:+PrintGCDetails 输出GC的详细日志；
其他的可以上网查一下，这篇文章的内容基本上参考的是周志明的《深入理解Java虚拟机》，很好的书

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

相关博文：
· 【原创】GC/垃圾回收简介
· GC垃圾回收
· 垃圾回收(GC)
· GC垃圾回收——有用的函数和类
· Java垃圾回收(GC)机制详解

最新新闻:
· 国家航天局:中国2030年或将实现载人登月
· TCL:彩电在北美销量首次第一, 超越三星
· 港媒:马云仍然希望阿里巴巴能够在香港上市
· 15秒不够用?抖音全面开放1分钟视频权限
· 日媒:华为在芯片设计领域已逼近苹果
» 更多新闻...