编译原理 第10次作业

Exercise 10.1

- Surf the Internet and write a short paper to compare GC on Java platform with GC on Microsoft .Net platform.

答：Java和.NET中均采用分代垃圾回收方法，但具体实现上存在差异。

1 划分方式

1.1 .Net platform

.NET框架中的垃圾回收器被称为分代的垃圾回收器（Generational GarbageCollector），被分配的对象划分为3个“代”。0代：从没有被标记为回收的新分配的对象；1代：在上一次垃圾回收中没有被回收的对象；2代：在一次以上的垃圾回收后仍然没有被回收的对象。0、 1、 2代对应的托管堆的初始化大小分别是256K， 2M和10M。

垃圾回收器在发现改变大小能够提高性能的话，会改变托管堆的大小。例如当应用程序初始化了许多小的对象,并且这些对象会被很快回收的话，垃圾回收器就会将第0代的托管堆变为128K，并且提高回收的频率。

如果情况相反，垃圾回收器发现在第0代的托管堆中不能回收很多空间时，就会增加托管堆的大小。

1.2 Java platform

新生代：几乎所有新生成的对象首先都是放在年轻代的。新生代内存按照8:1:1的比例分为一个Eden区和两个Survivor(Survivor0,Survivor1)区。大部分对象在Eden区中生成。当新对象生成， Eden Space申请失败（因为空间不足等），则会发起一次GC(Scavenge GC)。回收时先将Eden区存活对象复制到一个Survivor0区，然后清空Eden区，当这个Survivor0区也存放满了时，则将Eden区和Survivor0区存活对象复制到另一个Survivor1区，然后清空Eden和这个Survivor0区，此时Survivor0区是空的，然后将Survivor0区和Survivor1区交换，即保持Survivor1区为空， 如此往复。当Survivor1区不足以存放Eden和Survivor0的存活对象时，就将存活对象直接存放到老年代。当对象在Survivor区躲过一次GC的话，其对象年龄便会加1，默认情况下，如果对象年龄达到15岁，就会移动到老年代中。若是老年代也满了就会触发一次FullGC，也就是新生代、老年代都进行回收。新生代大小可以由-Xmn来控制，也可以用-XX:SurvivorRatio来控制Eden和Survivor的比例。

老年代：在年轻代中经历了N次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。内存比新生代也大很多(大概比例是1:2)，当老年代内存满时触发Major GC即Full GC， Full GC发生频率比较低，老年代对象存活时间比较长，存活率标记高。一般来说，大对象会被直接分配到老年代。所谓的大对象是指需要大量连续存储空间的对象，最常见的一种大对象就是大数组。

2 处理过程

2.1 .Net platform

在应用程序初始化的之前，所有等级的托管堆都是空的。当对象被初始化的时候，他们会*按照初始化的先后顺序被放入第0代的托管堆中*。最近被分配内存空间的对象被放置于第0代，因为第0代很小，小到足以放进处理器的二级（L2）缓存，所以*第0代能够为我们提供对其中对象的快速存取*，但是没有被使用的内存不会被自动释放。

当第0代中没有可以分配的有效内存时，就会在第0代中触发一轮垃圾回收，在这轮垃圾回收中将删除所有不再被引用的对象，并将当前正在使用中的对象移至第1代。针对第0代的垃圾回收是最常见的回收类型，而且速度很快。

在第0代的垃圾内存回收不能有效的请求到充足的内存时，就启动第1代的垃圾内存回收。

第2代的垃圾内存回收要作为最后一种手段而使用，当且仅当第1代和第0代的垃圾内存回收不能被提供足够内存时进行。

如果各代都进行了垃圾回收后仍没有可用的内存，就会引发一个OutOfMemeryException异常 。

另外，对于不同的代采用的是不同的算法。

对新生代/0代采用复制算法。复制算法将内存按容量按一定的比例分为对象面和空闲面。对象在对象面上创建；存活的对象被从对象面复制到空闲面，这个时候空闲面变为对象面；将对象面中所有的对象全部清空，这个时候对象面变为空闲面。

对老年代/1代、 2代采用标记-清除/标记-整理算法。标记-清除：从根集合进行扫描，对存活的对象进行标记，然后对堆内存从头到尾进行线性遍历，回收不可达对象内存。标记-整理：从根集合进行扫描，对存活的对象进行标记，然后移动所有存活的对象， 且按照内存地址次序依次排列，然后将末端内存地址以后的内存全部回收。

2.2 Java platform

新生代GC（Minor GC/Scavenge GC）：发生在新生代的垃圾收集动作。因为Java对象大多都具有朝生夕灭的特性，因此Minor GC非常频繁(不一定等Eden区满了才触发)，一般回收速度也比较快。在新生代中，每次垃圾收集时都会发现有大量对象死去，只有少量存活，因此可选用复制算法来完成收集。

老年代GC（Major GC/Full GC）：发生在老年代的垃圾回收动作。 Major GC，经常会伴随至少一次Minor GC。由于老年代中的对象生命周期比较长，因此Major GC并不频繁，一般都是等待老年代满了后才进行Full GC，而且其速度一般会比Minor GC慢10倍以上。另外，如果分配了Direct Memory，在老年代中进行Full GC时，会顺便清理掉Direct Memory中的废弃对象。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用标记—清除算法或标记—整理算法来进行回收。

3 内存管理模式

3.1 .Net platform

内存中的资源分为"托管资源"和"非托管资源"。托管资源必须接受.NET Framework的CLR的管理，而非托管资源则不必接受.NET Framework的CLR管理，需要手动清理垃圾（显式释放）。

托管资源又分别存放在"堆栈"和"托管堆"。所有的值类型（包括引用和对象实例）和引用类型的引用都存放在"堆栈"中，而所有引用所代表的对象实例都保存在托管堆中。

3.2 Java platform

所有资源都是托管资源，不存在需要手动清理垃圾的情况。