Chapter3 垃圾收集器与内存分配策略

概述

GC的历史比Java久远，1960年诞生于MIT的Lisp语言是第一门真正使用内存动态分配和垃圾收集技术的语言。

PC，虚拟机栈以及本地方法栈3个区域随着线程而生，随着线程而灭。栈中的栈帧分配多少内存在类结构确定的时候就已知了，因此这几个区域的内存分配和回收都具有确定性。在这几个区域就不需要过多考虑回收的问题，因为当方法或者线程结束以后，内存自然就跟随着回收了。

The stack frame generally includes the following components:

- The return address
- Argument variables passed on the stack
- Local variables (in HLLs)
- Saved copies of any registers modified by the subprogram that need to be restored (e.g. $s0-$s8 in MAL).

Java堆和方法区与他们不同，一个interface的实现类需要的内存不一样，每个方法不同的分支需要的内存也不一样，只有在运行期间才能知道会创建哪些对象。这部分内存的分配和回收都是动态的，也是GC主要关注的部分。

哪些对象要被回收

在堆中存放着Java中基本所有的对象实例，在GC进行工作之前需要首先判断哪些对象是存活的，哪些对象是死去的(不可能在被任何途径使用的对象)。

判断方法如下：

1. 引用计数算法

给对象加一个引用计数器，每当一个地方引用它时，计数器值+1，当引用失效时，计数器就-1

任何时刻计数器为0的对象是不可能再被使用的

总结：

实现简单，判定效率高，著名应用案例：微软的COM（Component Object Model）技术，FlashPlayer， Python和Squirrel

主流Java虚拟机中没有选用引用计数算法来管理内存，因为它很难解决对象之间的相互循环引用的问题

Eg: ObjA.instance = ObjB; ObjB.instance = ObjA;

这两个对象都不可能被访问，但是互相引用，无法被回收

2. 可达性分析

主流语言（Java， C#，Lisp）都是使用可达性分析来判定对象是否存活的

基本思路：通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到GC Roots没有任何引用链相连（从GC Roots到这个对象不可达），则证明该对象是不可用的

在Java语言中，可以被认为是GC Roots的对象主要包括以下几种：

- Local variable and input parameters of the currently executing methods
- Active threads
- Static field of the loaded classes
- JNI references

和上面一样：

- 虚拟机栈（栈帧中的本地变量表）中引用的对象
- 方法区中类静态属性引用的对象
- 方法区中常量引用的对象
- 本地方法栈中JNI（native方法）引用的对象

Java引用

在JDK1.2后，Java对引用的概念进行了扩张，分为强引用，软引用，弱引用，虚引用

四种引用强度逐渐降低

- 强引用

  代码中普遍存在的，类似于Object obj = new Object(); 只要强引用存在，GC永远不会回收掉被引用的对象

- 软引用

  一些还有用但是并非必须的对象，在系统将要发生内存溢出异常的时候，会将这些对象列入回收范围进行第二次回收

- 弱引用

  描述非必需对象，但引用强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发生之前

  当垃圾收集器工作时，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，在JDK1.2以后提供了WeakReference类来实现弱引用

- 虚引用

幽灵引用，幻影引用，一个对象是否有虚引用不会影响到其生存时间，也无法使用虚引用来得到一个对象实例

为一个对象设置一个虚引用的唯一目的就是能在这个对象被收集器回收时收到一个系统通知，在JDK1.2以后提供了PhantomReference类来实现虚引用

可达性分析中标记不可达的对象不一定要被回收

宣告一个对象死亡需要两次标记过程：

- 可达性分析后发现没有与GC Roots相连接的引用链，会被第一次标记
- 之后对象有一个拯救自己的机会，VM会对对象进行筛选，判断该对象是否有必要执行finalize()的方法
    - 当对象没有覆盖Object类中的finalize方法或者该方法已经被VM调用过了，则没必要执行finalize方法
    - 如果要执行finalize方法，首先把这个对象放置在F-Queue的队列之中，并在VM建立的低优先级的Finalizer线程中执行，但不会等待它执行结束（防止死循环或者执行缓慢，导致其他对象永久等待，以致内存回收系统崩溃）
    - 执行完finalize后GC对F-Queue中的对象进行第二次标记(如果对象在finalize中拯救自己（eg：把自己this指针赋值给某个类变量或者对象的成员变量，那就会被拯救))

任何一个对象的finalize（）只会被调用一次

回收方法区（Hotspot里的永生代）

回收效率比新生代低很多，新生代能达到（70%～95%）

VM可以不回收方法区

主要回收两部分内容：废弃常量和无用的类

废弃常量：

常量池的字面量：比如说有个字符串“abc”，当前系统中没有一个String对象叫做“abc”，可以把“abc”清理出常量池

无用的类判断条件复杂，要同时满足以下三个条件：

- 该类的所有实例都被回收，java堆中不存在该类的实例
- 加载该类的ClassLoader已经被回收

    顾名思义，它是用来加载 Class 的。它负责将 Class 的字节码形式转换成内存形式的 Class 对象。字节码可以来自于磁盘文件 *.class，也可以是 jar 包里的 *.class，也可以来自远程服务器提供的字节流，字节码的本质就是一个字节数组 []byte，它有特定的复杂的内部格式。

    每个 Class 对象的内部都有一个 classLoader 字段来标识自己是由哪个 ClassLoader 加载的。ClassLoader 就像一个容器，里面装了很多已经加载的 Class 对象。

- 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

  在`Java`中每个类实例都有对应的`Class`对象。类对象是由Java虚拟机(JVM)自动构造的。Class是一个java中的泛型类型。

垃圾收集算法

1. **标记清除算法**

   - 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

   不足：

   - 效率低，标记清除两个过程的效率都不高
   - 空间问题，会产生大量不连续的内存碎片，导致当要分配较大对象的时候无法找到足够的连续内存从而触发下一次垃圾回收动作

2. **复制算法（新生代）**

   将内存空间分成两块，每次只使用其中的一块。当这一块的内存用完以后，就将还存活的对象复制到另外一块上面，再把已使用过的内存空间一次清除掉

   优点： 实现简单，运行高效

   不足： 内存缩小成原来的一半

现在的商业虚拟机都采用这种收集算法来回收新生代，IBM研究表明98%的对象是“朝生夕死”的，不需要以1:1的大小来划分内存空间。现在一般是把内存划分成一块Eden空间和两块较小的Survivor空间。每次使用Eden空间和一块Survivor空间。回收的时候把Eden空间和Survivor空间中存活着的对象一次性复制到另外一块Survivor空间中，清理掉Eden空间和那块Survivor空间。 Hotspot中Eden空间和Survivor空间的大小比例为8:1。只有10%的空间会被浪费。

当Survivor空间不够用的时候，这些对象将直接通过分配担保机制进入老年代。

3. **标记整理算法（老年代）**

   复制收集算法在对象存活率较高的情况下要进行较多复制操作，效率变低。如果不想浪费50%空间，还需要有额外的空间进行分配担保，应对内存100%存活的极端情况。

   在老年代不适合用复制收集算法。

   针对老年代提出了标记整理算法

   - 标记：标记出所有要被回收的对象
   - 让所有存货的对象都向一端移动
   - 直接清除掉边界以外的内存

4. 分代收集算法

当前商用虚拟机都采用“分代收集”算法，按照对象存活周期把内存划分成几块，一般是把Java堆划分成新生代和老年代，针对不同代采用不同的收集算法。

新生代：存活率低 --> 复制算法

老圣代：存活率高，没有额外空间对他进行分配担保 --> 标记清理，标记整理算法

HotSpot的算法实现

1. 一个个查找太浪费时间

在安全点利用OopMap，在类加载的时候计算哪些位置是什么类型数据结构，JIT编译在安全点记录下哪些位置是引用

2. 保持一致性

停止所有节点（当分析GC引用链的时候）

3. 每个地方存OopMap浪费空间

只在安全点暂停，安全点一般在指令序列复用的地方

如何暂停?

- 抢占式中断

  当GC发生时，先把所有线程中断，如果发现有线程中断的地方不在安全点上，恢复线程，让它继续跑到安全点上（几乎不用这种）

  不需要线程的执行代码去主动配合

- 主动式中断

  GC需要中断线程的时候，仅设置一个标志，各个线程主动去轮询这个标志，当这个标志为真的时候就自己终端挂起，轮训标志的地方和安全点的位置是重合的

  把某个内存地址设为不可读，当线程执行到这个地方的时候就产生一个异常信号，在预先注册的异常处理器中暂停线程实现等待

4. 如果线程被blocked或者wait呢，不能等待它走到安全点

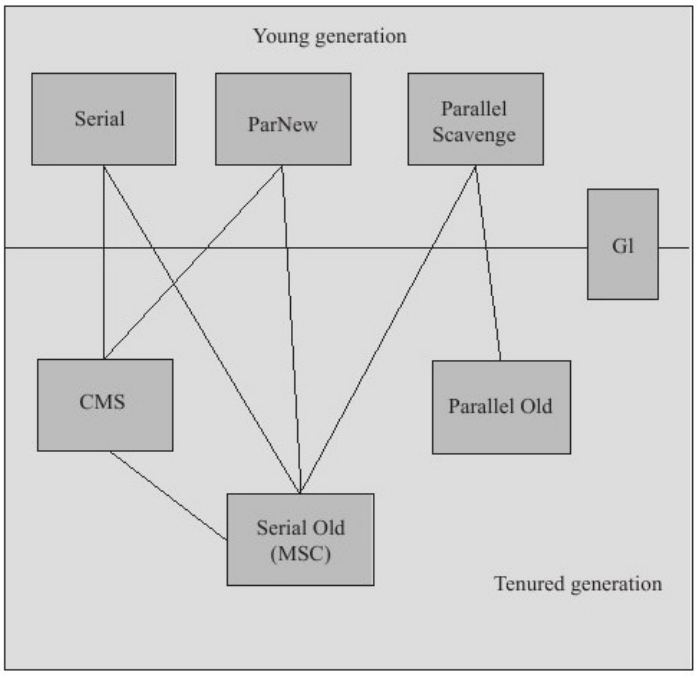
利用安全区域，安全区域在一段代码片段之中引用关系不会发生变化，在这个区域中的任何地方开始GC都是安全的

Safe Region

- 是被扩展了的SafePoint
- 当线程执行到Safe Region中的代码时，首先标记自己已经进入了Safe Region，这样当这段时间中JVM要发起GC时，就不用管标识为Safe Region状态的线程

- 当线程要离开安全区域的时候，它要检查系统是否完成了根节点枚举或者整个GC过程，如果完成了，那线程就继续执行，否则就一直等到可以安全离开Safe Region的信号为止

HotSpot中的垃圾收集器（7种）

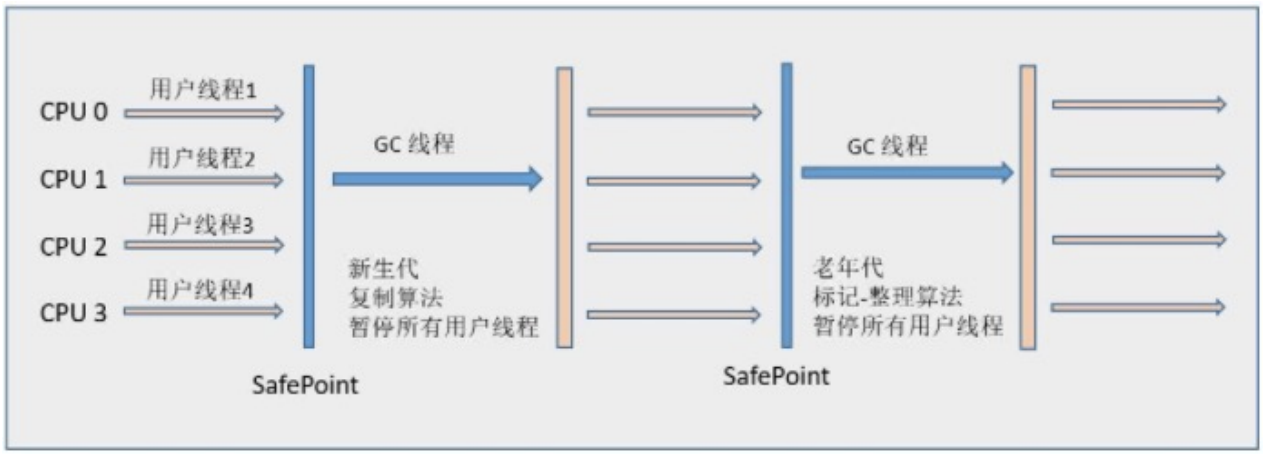


新生代收集器

1. Serial收集器

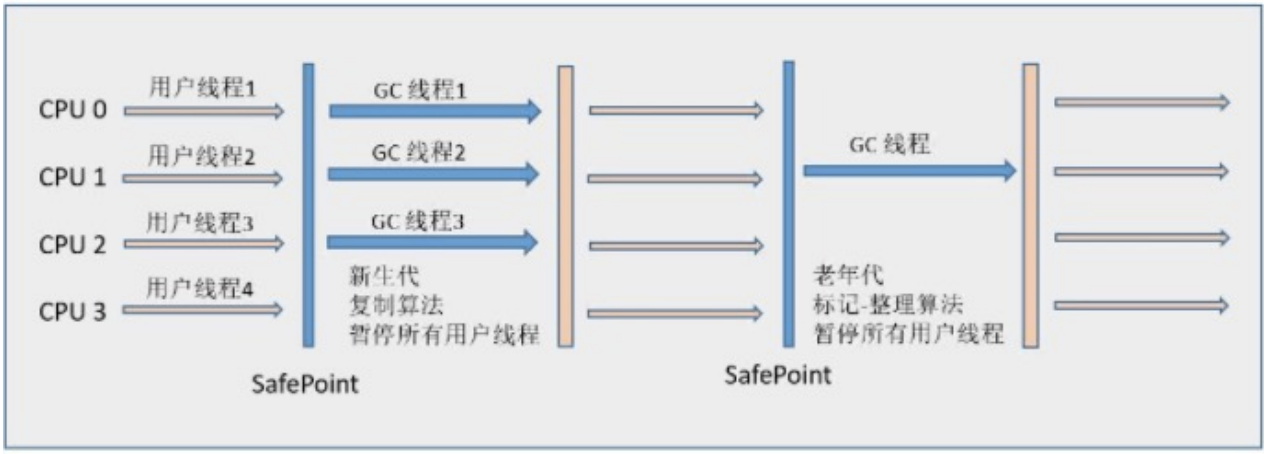
- 最基本，最悠久的垃圾收集器，曾经是新生代收集的唯一选择

- 单线程收集器
- 只使用一个CPU或一条收集线程
- 需要暂停其他所有的工作线程
- 虚拟机在后台自动暂停所有线程，在用户不可见的情况下把用户正常工作的线程全部停掉
- 虚拟机运行在Client模式下的默认新生代收集器，简单而高效，对于限定单个CPU的环境来说，没有现成交互的开销，由最高的单线程收集效率



2. ParNew收集器

- Serial收集器的多线程版本
- 在新生代采取复制算法的时候采用多线程
- 是许多运行在Server模式下的虚拟机的首选新生代收集器
- **除了Serial收集器之外，目前只有它能和CMS收集器一起工作**
- 在单CPU环境下效果没有Serial好



并行与并发

- 并行：多条垃圾收集线程并行工作，但用户线程还处在等待状态
- 并发： 用户线程与垃圾收集线程同时执行（不一定是并行的，可能会交替执行），用户程序在继续运行，而垃圾收集程序运行在另外一个CPU上

如果某个系统支持两个或者多个动作（Action）**同时存在**，那么这个系统就是一个**并发系统**。如果某个系统支持两个或者多个动作**同时执行**，那么这个系统就是一个**并行系统**。并发系统与并行系统这两个定义之间的关键差异在于**“存在”**这个词。

“并行”概念是“并发”概念的一个子集

3. Parallel Scavenge收集器

- 新生代收集器
- 并行多线程收集器
- 复制算法
- CMS的关注点 是缩短垃圾收集是用户线程的停顿时间，而Parrallel Scavenge的目的是达到一个可控制的吞吐量
    - 吞吐量 Throughput：CPU用于运行 用户代码时间/CPU总消耗时间的比值（CPU总消耗时间为运行用户代码时间+垃圾收集时间）
    - eg：虚拟机总运行了100分钟，其中垃圾收集花掉了1分钟，吞吐量为99/100 = 99%
    - 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
    - 高吞吐量可以高效率的利用CPU时间，尽快完成程序的运算任务，主要是和在后台运算 而不需要太多交互的任务
- 两个参数：最大垃圾收集停顿时间，吞吐量大小

  GC听段时间缩短是以牺牲吞吐量和新生代空间（GC更频繁）来换取的
- 自适应调节策略

老年代收集器

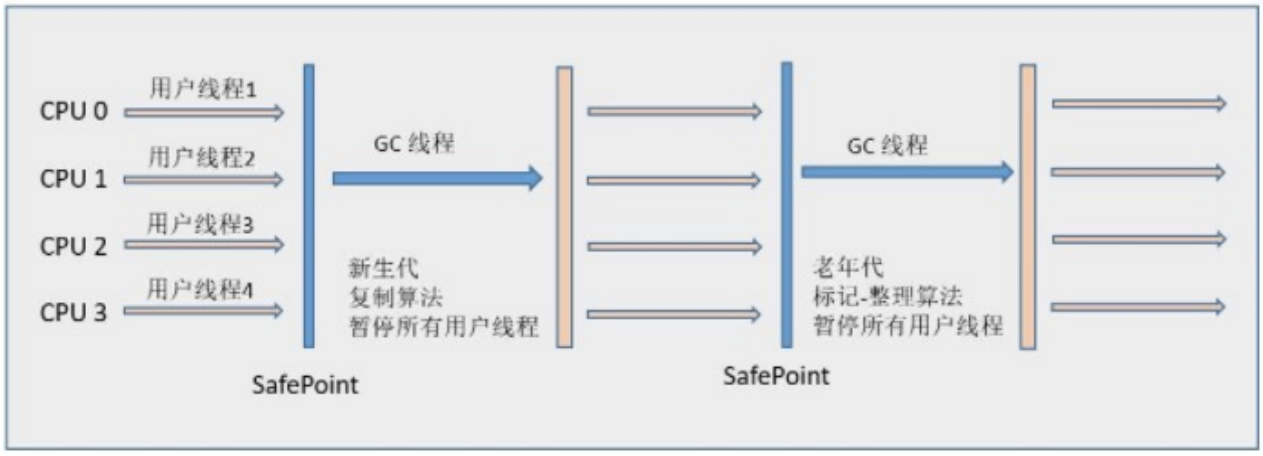
4. Serial Old

是Serial的老年版，单线程收集器，使用“标记--整理”算法

主要给Client模式下的虚拟机使用

在Server模式下有两大用途：

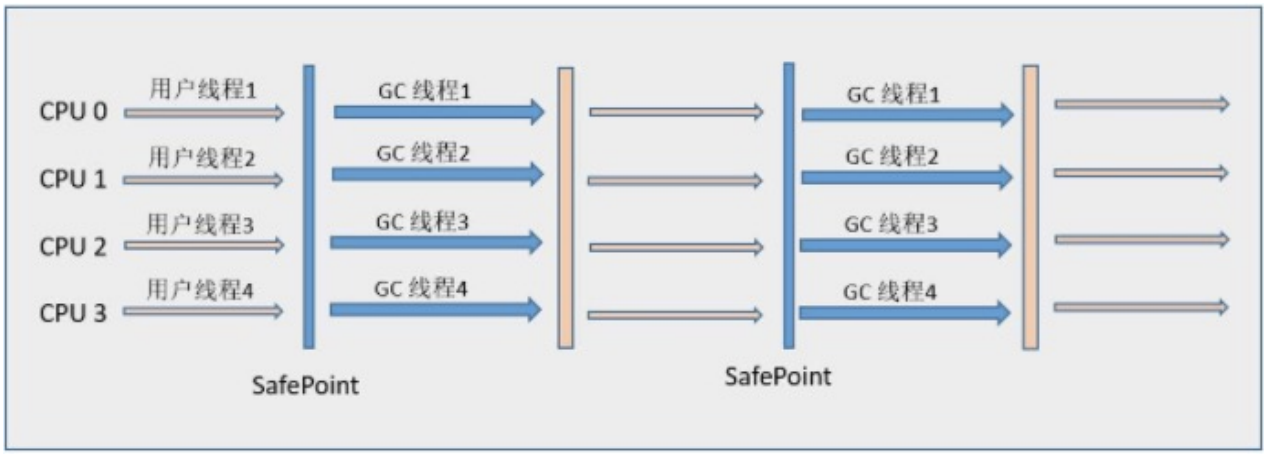
- 在JDK1.5以前与Parallel Scavenge搭配使用
- 作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure的时候使用



5. Parallel Old

是Parallel Scavenge的老年代版本，使用多线程和“标记--整理”算法

配合Parallel Scavenge使用，“吞吐量优先”以及CPU资源敏感



6. CMS收集器

Concurrent Mark Sweep

以最短回收停顿时间为目标的收集器

现在很多Java应用在互联网上或者B/S系统（browser/server）的服务器端，重视响应速度

基于标记清除算法

- 初始标记（CMS initial mark） **STW， stop the world**

  标记GC Roots能直接关联到的对象

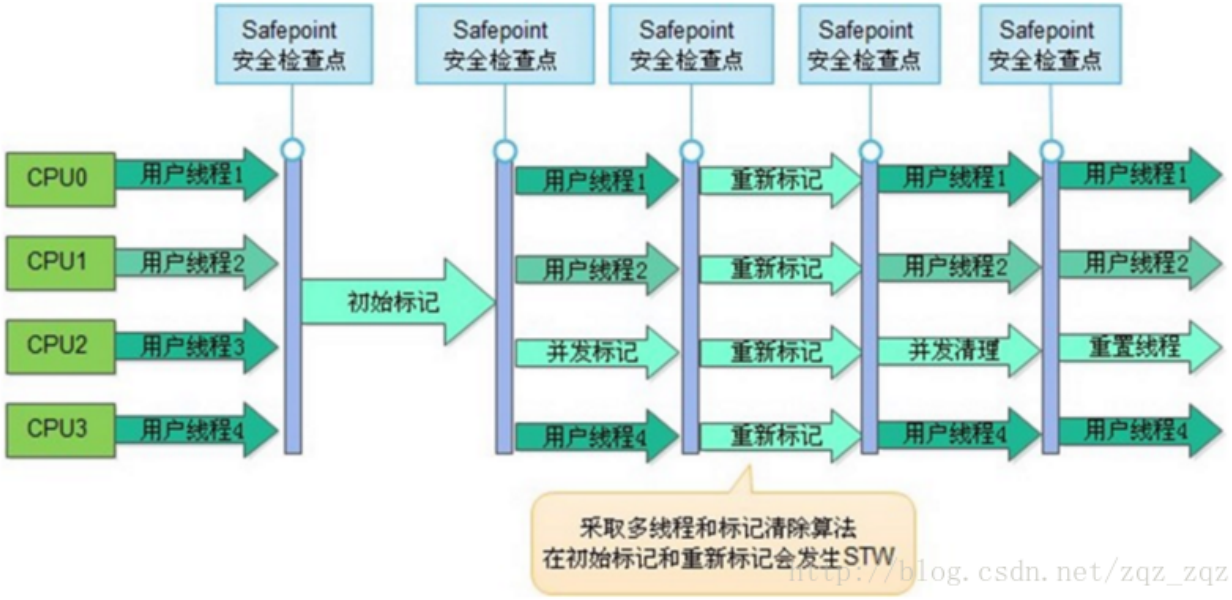
  - 标记老年代中所有的GC Roots对象
  - 标记年轻代中活着的对象引用到的老年代的对象
- 并发标记（CMS concurrent mark）

  从“初始标记”阶段标记的对象开始找出所有存活的对象
- 重新标记（CMS remark） **STW， stop the world**

  修正并发标记过程中因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间会比初始标记的停顿时间稍长一些，但远比并发标记的时间短
- 并发清除（CMS concurrent sweep）

  清除那些没有标记的对象并且回收空间

由于整个过程中耗时最长的**并发标记**和**并发清除**过程收集器线程都可以与用户进程一起工作，因此可以总体上说CMS收集器的内存回收过程是与用户线程一起并发执行的。



CMS缺点：

- 对CPU资源十分敏感
  - 面对并发设计的程序对CPU资源比较敏感，在并发阶段，它虽然不会导致用户线程停顿，但会占用一些CPU资源而使应用程序变慢，总吞吐量会降低
  - 默认启动的回收线程数是（CPU个数+3）/4，意味着在CPU数量在4个以上的时候，并发回收时垃圾收集占不少于25%的CPU资源，并且随着CPU数量的增加而下降
  - 当CPU个数小于2个的时候，CMS对用户程序的影响很大，如果本来CPU负载比较大，还分出一半的运算能力去执行收集器线程，就可能导致用户程序的执行速度降低了50%
    - 为了解决这个问题，出现了i-CMS，在并发标记、清理的时候让GC线程和用户线程交替运行，尽量减小GC线程独占资源的时间，这样会使垃圾收集的过程更长，但对用户程序的影响会更小一些，速度下降没有那么明显（然而实际效果一般，deprecated）
- 无法处理浮动垃圾，可能出现Concurrent Mode Failure而导致另一次Full GC
  - 由于在CMS并发清理阶段用户线程还在运行着，伴随程序自然就会产生新的垃圾，这部分垃圾出现在标记过程之后，因此CMS无法在当次收集中清理掉它们，只能等待下一次GC时再清理掉，他们被称为“浮动垃圾”
  - 由于垃圾清理阶段用户线程还需要运行，不能在老年代完全被填满以后再进行收集，需要预留足够的内存空间给用户线程使用
  - JDK1.5中默认情况下CMS在老年代使用了68%以后就会被激活，在JDK1.6中，CMS的启动阈值已经提升到92%。要是CMS在运行期间预留的内存无法满足程序需要，就会出现“Concurrent Mode Failure”，这样虚拟机启动后备预案，临时启用Serial Old收集器来进行老年代的垃圾收集
- CMS基于“标记--清除”算法实现，会产生大量空间碎片，老年代还会有很大空间剩余，但无法找到足够大的连续空间来分配当前对象，不得不提前出发一次Full GC。为此提供了FullGC开启时内存碎片的合并整理的开关，解决了空间碎片的问题，但是停顿时间会增加，此外还有执行多少次不压缩FullGC后执行一次带压缩的Full GC的选项

7. G1垃圾收集器

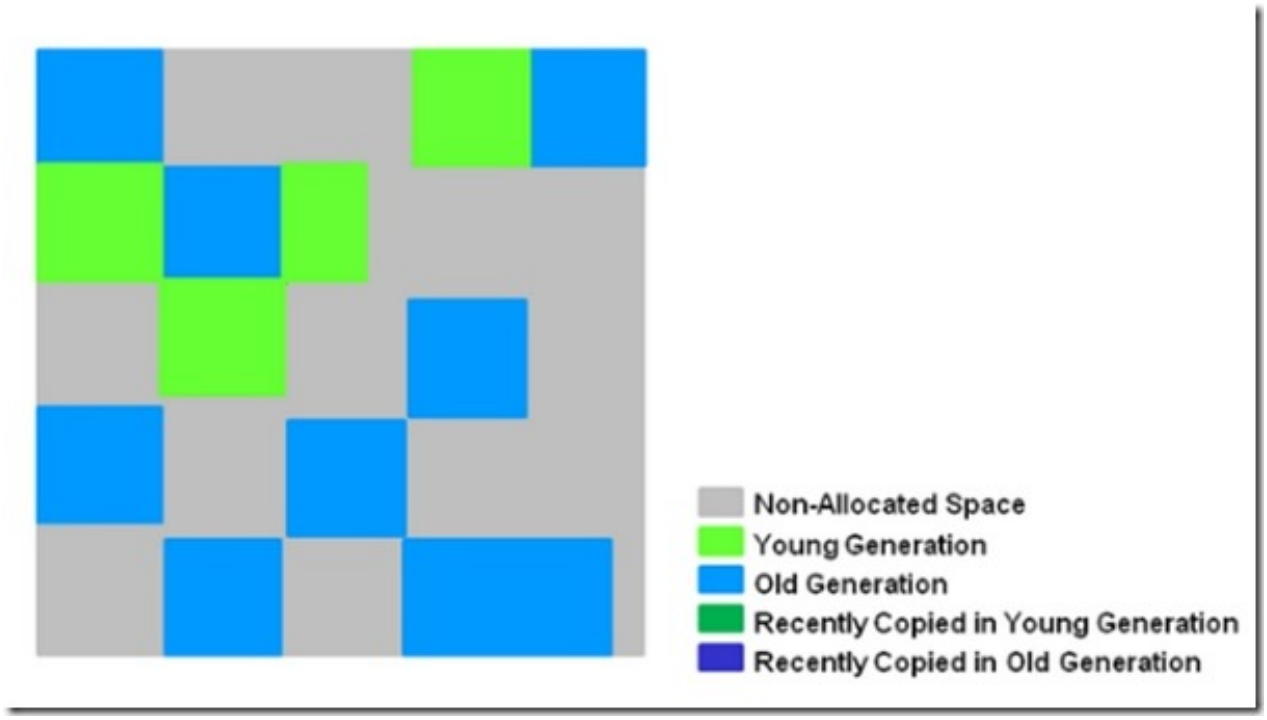
是当今收集器技术发展的前沿成果之一

面向服务器端应用的垃圾收集器，HotSpot开发团队目标是未来用它替换掉JDK1.5中发布的CMS垃圾收集器

- 并发与并行： 能充分利用多CPU、多核环境下的硬件优势，使用多个CPU来缩短Stop the world的停顿时间
- 分代收集： 对不同对象采用不同的方式收集
- 空间整合： 与CMS收集器的“标记--清理”算法不同，G1从整体来看是基于“标记--整理”算法实现的收集器，从局部上来看是基于复制算法实现的，都不会产生空间碎片，收集后可以提供规整的可用内存，有利于程序长时间运行
- 可预测的停顿： CMS和G1都关注降低停顿时间，但G1还可以建立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毫秒的时间片段中，消耗在垃圾收集的时间不得超过N毫秒，这已经几乎是实时Java（RTSJ）的垃圾收集器的特征
    - 它可以有计划的避免在整个Java堆中进行全区域的垃圾收集，它跟踪各个区域里面垃圾堆积的价值大小（回收所获得的空间大小以及回收时间的经验值），在后台维护一个优先列表，优先回收价值最大的Region（Garbage-First）区域名称的来由。
    - 这种使用Region划分内存空间以及有优先级的区域回收方式，保证了G1收集齐在有限的时间内可以获得尽量高的收集效率

G1之前的垃圾收集器进行收集的范围都是整个新生代或者老年代，而G1中新生代与老年代不再是物理隔离的了，就是将整个Java堆划分成多个大小相等的独立区域，它们都是一部分（不需要连续的）区域的集合

G1将Java堆空间分割成了若干个Region，即年轻代/老年代是一系列Region的集合，这就意味着在分配空间时不需要一个连续的内存区间，即不需要在JVM启动时决定哪些Region属于老年代，哪些属于年轻代。因为随着时间推移，年轻代Region被回收后，又会变为可用状态了



存在的难点

一个对象分配在某个Region中，并非只能被本Region中的其他对象使用，而是与整个Java堆任意的对象发生引用关系-----要扫描整个Java堆才能保证准确性?

解决方案：

使用Remembered Set避免全堆扫描

G1中每个Region都有一个Remembered Set，虚拟机发现程序对Reference类型的数据进行写操作的时候，会产生一个Write Barrier暂时中断写操作，检查Reference引用的对象是否处于不同的Region之中，如果是，则通过CardTable把相关引用信息记录到被引用对象所属Region的Remembered Set中

G1将Heap划分为多个固定大小的region，这也是G1能够实现控制GC导致的应用暂停时间的前提，region之间的对象引用通过remembered set来维护，每个region都有一个remembered set，remembered set中包含了引用当前region中对象的region的对象的pointer，由于同时应用也会造成这些region中对象的引用关系不断的发生改变，G1采用了Card Table来用于应用通知region修改remembered sets

在进行内存回收的时候，在GC根节点枚举范围中加入Remembered Set就可以保证不对全堆扫描也不会有遗漏

G1收集器的运作流程（除了Remembered Set）

- 初始标记
  - stop the world,标记GC Roots能直接关联到的对象
  - 修改TAMS，next top at mark start，让下一阶段的用户程序并发运行时能在正确可用的Region中创建新的对象，耗时较短
- 并发标记
  - 从GC Roots开始对堆中的对象进行可达性分析，耗时较长，但是可以与用户程序并发执行
- 最终标记
  - 修正在并发标记的时候引用户运作而导致标记发生变动的那一部分标记记录
  - JVM把这段时间对象变化记录在线程Remembered Set Logs里面，在最终标记阶段吧Remembered

  Set Logs中的数据合并到Remembered Set中
  - 这里需要STW
- 筛选回收
  - 最终根据每个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来制定回收计划
  - 这个阶段可以并发执行，但是由于只回收一部分Region，时间是用户可控制的，停顿用户进程将大幅提高收集效率

内存分配策略

对象的内存分配，从大方向将，就是在堆上分配的（但也可能根据JIT编译后被拆散为标量类型并间接的栈上分配）。对象主要分配在新生代的Eden区，少数情况下也会直接分配在老年区

1. 对象优先在Eden分配

大多数情况下，对象在新生代Eden区中分配，当Eden区没有足够空间进行分配时，虚拟机将发起一次Minor GC

2. 大对象直接进入老年代

所谓的大对象指的是需要大量连续内存空间的Java对象，最典型的是那种需要很长的字符串以及数组

碰到大对象容易导致内存还有不少空间的时候就提前触发垃圾收集以获取足够的连续空间来安置它们

3. 长期存活的对象将进入老年代

虚拟机给每个对象定义了一个对象年龄计数器，如果对象在Eden出生并经过第一次Minor GC后仍然存活，并且能被Survivor容纳的话，将会被移动到Survivor空间中，并且对象年龄设为1，当他的年龄增加到一定数量（15岁）时，就会晋升到老年代

为了更好地适应不同程序的内存状况，如果在Survivor中相同年龄所有对象大小的综合大于Survivor的一半，年龄大于或等于该年龄的对象就可以直接进入老年代

空间分配担保

在发生Minor GC之前，虚拟机会检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那么Minor GC可以保证是安全的

原因：

新生代使用复制收集算法，为了提高内存利用率，只使用一个Survivor空间来作为轮换备份，因此当出现大量对象Minor GC之后仍然存活的情况（最极端的情况就是内存回收之后新生代的所有对象都存活），就需要老年代进行分配担保，把Survivor中不能容纳的对象直接进入老年代

Minor GC

- 从年轻代空间（包括 Eden 和 Survivor 区域）回收内存被称为 Minor GC
- 所有的 Minor GC 都会触发（stop-the-world)
- 每次 Minor GC 会清理年轻代的内存

Full GC

清理整个堆空间—包括年轻代和老年代。