**[jvm系列 (二) ---垃圾收集器与内存分配策略](http://www.cnblogs.com/-new/p/7284368.html)**

<http://www.cnblogs.com/-new/p/7284368.html>

# [jvm系列 (二) ---垃圾收集器与内存分配策略](http://www.cnblogs.com/-new/p/7284368.html)

# 垃圾收集器与内存分配策略

前言：本文基于《深入java虚拟机》再加上个人的理解以及其他相关资料，对内容进行整理浓缩总结。本文中的图来自网络，感谢图的作者。如果有不正确的地方，欢迎指出。

## 目录

* [jvm系列（一）：jvm内存区域与溢出](http://www.cnblogs.com/-new/p/7244460.html)
* [jvm系列（二）：垃圾收集器与内存分配策略](http://www.cnblogs.com/-new/p/7284368.html)

## 回顾

* 上文介绍了jvm的内存区域以及介绍了内存的溢出情况。
* jvm区域分为5个，线程独有：虚拟机栈，本地方法栈，程序计数器。线程共享：方法区，堆
* 两种溢出：栈溢出（StackOverflowError），OutOfMemoryError（OOM）

## 为什么学习垃圾收集

* 看起来jvm好像一切帮你做好，但是当垃圾收集成为系统达到更高并发量的瓶颈时，我们就需要对这种自动化的技术进行监控和调节。
* 根据实际应用需求，选择最优的收集方式才能更高的性能。

## 垃圾收集的区域

* 虚拟机栈，本地方法栈，程序计数器是线程私有的，和线程同生共死，当线程销毁时，内存自然回收，所以这部分不是考虑的重点。
* 所以研究重点应该在方法区和堆，而方法区的回收效率较低，重点在堆。

# 确定回收对象

## 引用计数

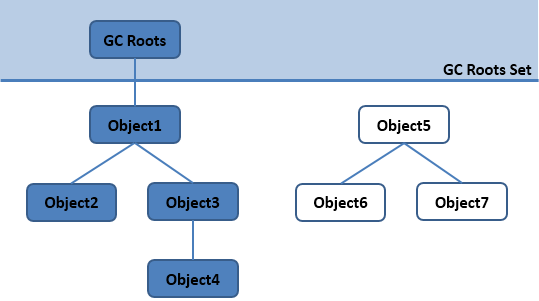
* 对象有一个引用计数器 new String（）
* 有引用，计数器加一，String jiajun=new String（）
* 引用失效，计数器减一，jiajun=null；
* 出现一个问题
* Person jia=new Persoon();
* Person jun=new Person();
* jia.bro=jun;
* jun.bro=jia;
* jia=null;

jun=null;

* 此时虽然我们没有办法用jia和jun这两个对象，也就是这个是垃圾了，但是两个对象又互相引用，如果用这种算法是无法进行回收

## 可达性分析

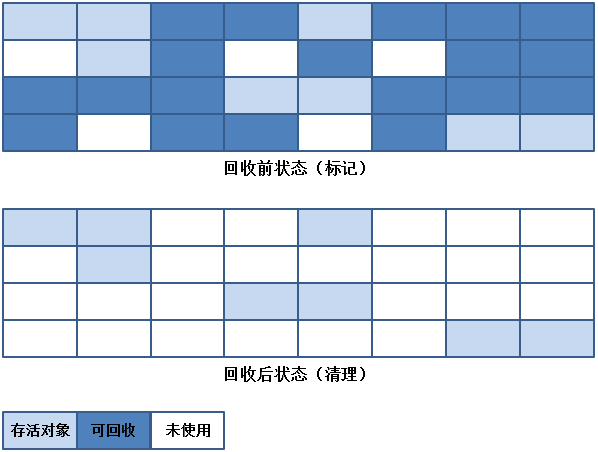
* GC Roots的对象：虚拟机栈中引用的对象（比如方法中定义的对象），本地方法栈中引用的对象，方法区中类静态属性引用的对象（static），方法区中常量应用的对象（final）
* 通过GC Roots对象作为起点，当GC Roots到一个对象没有引用链的话，那么就证明这个而对象不可用



# 垃圾收集算法

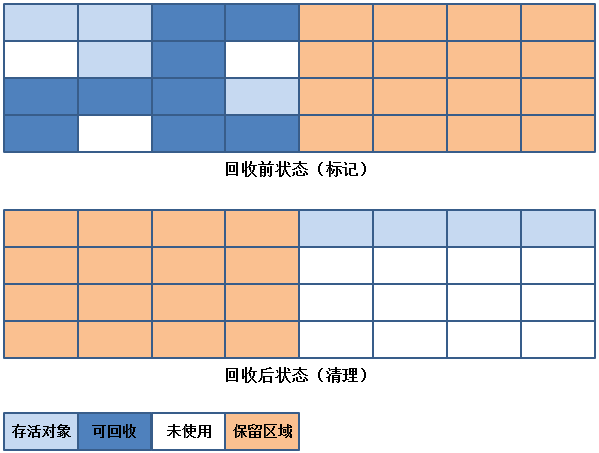
## 标记清除

* 确定回收对象，进行标记，然后回收标记的对象
* 一个问题是标记清除效率不高
* 一个问题是标记清除后产生大量不连续的内存碎片，但需要分配较大对象的时候，因为没有足够大的连续空间分配给此对象，此时会触发另一次垃圾收集



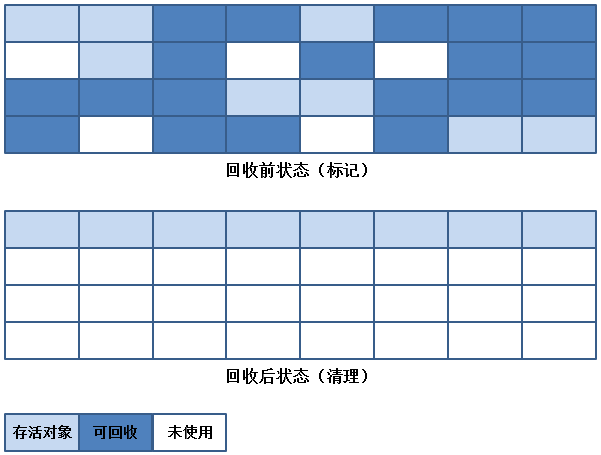
## 复制

* 解决内存碎片的问题
* 将内存分为等大的A区和B区，创建对象时，存放于A区，当A区空间用完之后，将A区存活的对象复制到B区，然后A区清空，这样的话，就避免了内存碎片的问题
* 但是又有一个问题，这个时候我们只用到了一半的空间，所以我们要向办法提高空间利用率
* 研究发现，新生代对象大多都是朝生夕死，也就是存活的对象不多，那么我们就没必要分配一半的空间用于“粘贴”
* 所以虚拟机将新生代分为Eden和Surivior两个区，比例为8:1:1，可用的内存为一个伊甸区个一个存活区，一个存活区用于“粘贴,”，也就是新生代可用内存空间为容量的90%，这样的话就提高了空间了利用率
* 但是又有一个问题，如果这一个存活区存放不了复制的的对象，那么怎么办？于是，如果这些对象放不下，将直接进入另一块区域老年代



## 标记整理

* 对于朝生夕死的新生代来说，复制算法是不错的选择。但是对于存活率高的对象不是很好的选择，因为要进行较多的复制操作，效率会降低
* 过程：确定回收对象，进行标记，让存活的对象向一端移动，然后清理掉端边界以外的内存
* 那么这样的话，不会有内存碎片的问题，也没有复制过多效率降低的问题



## 分代收集

* 上面几种方法的综合利用
* 根据新生代和老年代的特点，分别选用适用的算法。
* 新生代存活率，那么可以选择复制算法
* 老年代存活率高，可以采用标记清理或标记整理算法

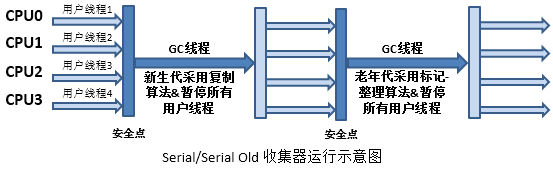
# 垃圾收集器

## Stop the world（STW）

* 意思指GC时，停顿所有java执行线程
* 因为在进行可达性分析时候，如果同时对象引用进行变化，那么这样可达性分析就不正确。这是stop the world 的重要原因

## Serial收集器

* 用一条线程去完成垃圾收集工作
* 垃圾收集时，需要暂停其他工作线程，显然这是不好，所以缩短线程停顿的时间是一个研究重点
* 采用复制算法，用于新生代

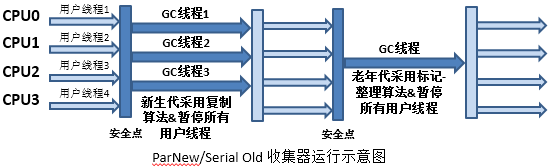


## Serial Old收集器

* 与Serial收集器类似
* 采用标记整理算法，用于老年代

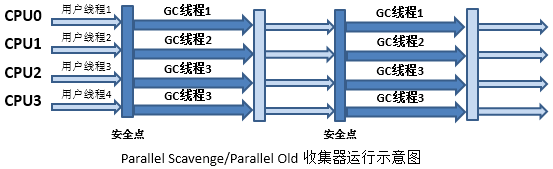
## ParNew收集器

* 用多条线程去完成收集工作
* 垃圾收集时，需要暂停其他工作线程
* 默认开启的收集线程数和cpu数量一样，ParallelGCThreads参数可以用来设置线程数
* 用于新生代，复制算法



## Parallel Scavenge收集器

* 新生代收集器，使用复制算法
* 关注吞吐量，吞吐量=代码运行时间/(代码运行时间+垃圾收集时间),也就是高效率利用cpu时间，尽快完成程序的运算任务
* MaxGCPauseMillis参数设置最大停顿时间，GCTimeRatio设置吞吐量大小

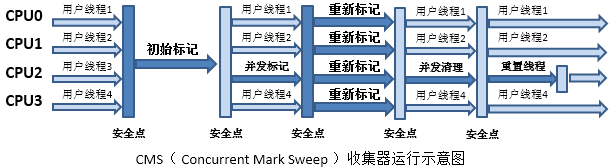


## Parllel Old收集器

* Parallel Scavenge的老年代版
* 使用标记整理算法

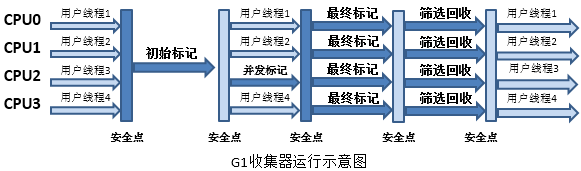
## CMS收集器

* 用于老年代
* 关注停顿时间，希望获取最短回收停顿时间
* 基于标记清除算法，会产生内存碎片
* 总体来说和用户线程一起并发执行
* 对cpu资源敏感，也就是会占用较多cpu资源
* 无法处理浮动垃圾，由于是和用户线程并发执行，所以并发时用户线程产生的的新垃圾（浮动垃圾）无法收集。



## G1收集器

* 使用多个cpu来缩短STW（stop the world ）的停顿时间
* 分代收集
* 不会产生内存空间碎片
* 可以建立可预测的停顿时间模型，能够让使用者指定M毫秒时间段内，垃圾收集的时间不超过N毫秒



## 流行的收集器组合

| **新生代** | **老年代** |
| --- | --- |
| Serial | Serial Old |
| Serial | CMS |
| ParNew | CMS |
| ParNew | Serial Old |
| Parallel Scavenge | Serial Old |
| Parallel Scavenge | Parallel Old |
| G1 | G1 |

## 垃圾收集参数总结

| **参数** | **描述** |
| --- | --- |
| UseSerialGC | 虚拟机运行在Client 模式下的默认值，打开此开关后，使用Serial +Serial Old 的收集器组合进行内存回收 |
| UseParNewGC | 打开此开关后，使用ParNew + Serial Old 的收集器组合进行内存回收 |
| UseConcMarkSweepGC | 打开此开关后，使用ParNew + CMS + Serial Old 的收集器组合进行内存回收。Serial Old 收集器将作为CMS 收集器出现Concurrent Mode Failure失败后的后备收集器使用 |
| UseParallelGC | 虚拟机运行在Server 模式下的默认值，打开此开关后，使用ParallelScavenge + Serial Old（PS MarkSweep）的收集器组合进行内存回收 |
| UseParallelOldGC | 打开此开关后，使用Parallel Scavenge + Parallel Old 的收集器组合进行内存回收 |
| SurvivorRatio | 新生代中Eden 区域与Survivor 区域的容量比值， 默认为8， 代表Eden ：Survivor=8∶1 |
| PretenureSizeThreshold | 直接晋升到老年代的对象大小，设置这个参数后，大于这个参数的对象将直接在老年代分配 |
| MaxTenuringThreshold | 晋升到老年代的对象年龄。每个对象在坚持过一次Minor GC 之后，年龄就加1，当超过这个参数值时就进入老年代 |
| UseAdaptiveSizePolicy | 动态调整Java 堆中各个区域的大小以及进入老年代的年龄 |
| HandlePromotionFailure | 是否允许分配担保失败，即老年代的剩余空间不足以应付新生代的整个Eden 和Survivor 区的所有对象都存活的极端情况 |
| ParallelGCThreads | 设置并行GC 时进行内存回收的线程数 |
| GCTimeRatio | GC 时间占总时间的比率，默认值为99，即允许1% 的GC 时间。仅在使用Parallel Scavenge 收集器时生效 |
| MaxGCPauseMillis | 设置GC 的最大停顿时间。仅在使用Parallel Scavenge 收集器时生效 |
| CMSInitiatingOccupancyFraction | 设置CMS 收集器在老年代空间被使用多少后触发垃圾收集。默认值为68%，仅在使用CMS 收集器时生效 |
| UseCMSCompactAtFullCollection | 设置CMS 收集器在完成垃圾收集后是否要进行一次内存碎片整理。仅在使用CMS 收集器时生效 |
| CMSFullGCsBeforeCompaction | 设置CMS 收集器在进行若干次垃圾收集后再启动一次内存碎片整理。仅在使用CMS 收集器时生效 |

# 内存分配与回收策略

## GC种类

* Minor GC，新生代GC，回收速度快
* Major GC/ Full GC , 收集整个堆，包括yong gen，old gen，perm gen，回收速度慢

## 对象优先分配在新生代中的Eden

* 大多数情况下，对象分配在Eden区，如果Eden区空间不够，将发起一次MinorGC

## 大对象直接进入老年代

* 大对象指需要大量连续内存空间的对象
* 经常出现大对象，由于需要连续的空间，容易导致内存还有不少空间就提前触发垃圾收集
* 通过PretenureSizeThreshold参数设置限制，超过这个限制的对象直接分配在老年代

## 长时间存活的对象将进入老年代

* 每个对象有一个对象年龄计数器，经过第一次MinorGC进入存活区，年龄为一，以后经过MinorGC还能继续在存活区的话，年龄加一，当通过限制（MaxTenuringThreshold）后会晋升到老年代

## 动态对象年龄判定

* 存活区空间相同年龄所有对象大小的总和大于存活区的空间一半，大于等于该年龄的可以直接进入老年代

## 空间分配担保

* Minor GC之前，虚拟机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，如果可以，可以安全进行
* 如果不行，看是否允许担保失败，不允许的话进行Full GC
* 允许的话，看老年代的最大可用连续空间是否大于历次晋升老年代对象的平均大小（将之前的作为经验值），如果大于，进行MinorGC，否则进行FullGC

# 方法区回收

## 回收内容

* 废弃常量
* 无用的类
* 回收的效率不高

## 废弃常量

* 没有任何地方引用常量池的字符串常量，必要的话，这个字符串会被清除常量池。常量池中的类 方法 字段的符号引用也是类似

## 无用的类

* 不存在该类的任何对象
* 加载该类的ClassLoader已经被回收
* 该类的对应的Class对象没有被引用，无法在任何地方通过反射访问该类的方法

# gc触发条件

## ygc

* 当eden空间不足的时候会触发ygc

## full gc

* 创建大对象时，eden区空间不足，会尝试分配到老年代，如果老年代空间也不足就会触发full gc
* 触发yong gc的时候，会检查老年代的最大连续空间是否大于新生代所有对象总空间，如果大于的话，说明这次yong gc是安全的。如果小于的话，看是否允许担保，不允许担保的话，那么进行full gc。如果允许担保的话，那么参考一下历次晋升到老年代的对象的平均大小，如果老年代连续空间大于这个参考值，那么尝试进行ygc。如果小于这个参考值的话，没办法只能进行full gc。
* 当系统要加载的类，反射的类和调用的方法较多，并且永久代空间不足的话，进行full gc
* 当to区的空间不足的时候，会把对象复制到老年代，如果老年代空间不足会进行full gc